



9 Awesome Projects
written especially for young people!

Adventures in Minecraft®

```
# use FindPointOnCircle Function to find a point on a circle
hourHandX, hourHandY = FindPointOnCircle(clockMiddleX, clockMiddleY, 100, 0)

# draw hour hand
mcDrawing.drawLine(clockMiddleX, clockMiddleY, hourHandX, hourHandY, block.DIRT.id)

#minute hand
# what angle would a minute hand point to?
minHandAngle = (360 / 60) * minutes
# use FindPointOnCircle Function to find a point on a circle
minHandX, minHandY = FindPointOnCircle(clockMiddleX, clockMiddleY, 100, minHandAngle)

# draw minute hand
mcDrawing.drawLine(clockMiddleX, clockMiddleY, minHandX, minHandY, block.WOOD_PLANKS.id)

#second hand
# what angle would a second hand point to?
secHandAngle = (360 / 60) * seconds
# use FindPointOnCircle Function to find a point on a circle
secHandX, secHandY = FindPointOnCircle(clockMiddleX, clockMiddleY, 100, secHandAngle)
```



**Martin O'Hanlon
David Whale**

WILEY

Adventures in Minecraft®

Adventures in Minecraft®

Martin O'Hanlon and David Whale

WILEY

This edition first published 2015

© 2015 John Wiley and Sons, Ltd.

Registered office

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book. This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

All photographs used in this work courtesy of S K Pang, 2014.

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and/or other countries, and may not be used without written permission. Minecraft is a registered trademark of Mojang Synergies AB Corporation. All other trademarks are the property of their respective owners. John Wiley & Sons, Ltd. is not associated with any product or vendor mentioned in the book.

A catalogue record for this book is available from the British Library.

ISBN 978-1-118-94691-6 (paperback); ISBN 978-1-118-94685-5 (ePub); 978-1-118-94684-8 (ePDF)

Set in Chaparral Pro 10\12.5 by TCS\SPS

Printed and bound in Great Britain by Bell & Bain

For my wife Leonie, without you, this would never have been.

—Martin.

For my wife Gail, for putting up with me while I constantly played Minecraft.

—David.

Publisher's Acknowledgements

Some of the people who helped bring this book to market include the following:

Editorial

Publisher: Barry Pruett
Associate Publisher: Jim Minatel
Executive Commissioning Editor: Craig Smith
Acquisitions Editor: Aaron Black
Project Editor: Sydney Argenta
Copy Editor: Grace Fairley
Technical Editor: Cliff O'Reilly
Editorial Manager: Mary Beth Wakefield
Senior Project Editor: Sara Shlaer
Editorial Assistant: Jessie Phelps
Illustrator: Sarah Wright

Marketing

Marketing Manager: Lorna Mein
Marketing Assistant: Polly Thomas

Minecraft Consultants

Zachary Igielman
Lauren Trussler
Sam Whale
Ben Foden
Ben Ramachandra
Ria Parish

About the Authors

MARTIN O'HANLON has been designing and programming computer systems for all of his adult life. His passion for programming and helping others to learn led him to create the blog <Stuff about="code" /> (www.stuffaboutcode.com) where he shares his experiences, skills and ideas. Martin regularly delivers presentations and workshops on programming Minecraft to coders, teachers and young people with the aim of inspiring them to try something new and making programming fun.

DAVID WHALE writes computer programs for devices you wouldn't imagine have computers inside them. He was bitten by the computer programming bug aged 11 when he was at school, and still thoroughly enjoys writing software and helping others to learn programming. He runs a software consultancy business in Essex, but also regularly volunteers for The Institution of Engineering and Technology (The IET) helping in schools, running weekend computing clubs, judging schools competitions, and running programming workshops for young people at community events all around the UK. You can follow his adventures on his blog at <http://blog.whaleygeek.co.uk>.

Acknowledgments

Many people are involved in producing a book, too many to mention in this small space. We would both like to give our special thanks to the following people:

- The staff at Mojang, for designing such a great game, and their genius and insight in making the game programmable. Without this insight, this book would not have been possible.
- The Raspberry Pi Foundation and the open source community, without which there wouldn't be a Raspberry Pi or a Bukkit server, both of which are vital platforms that enabled this book to be written for a wide audience.
- Our testers and young Minecraft experts, Zachary Igielman, Lauren Trussler, Sam Whale, Ben Foden, Ria Parish, who tried our programs and provided really useful feedback, without which we would never have known if we were pitching the book correctly to the target age group.
- Mr S.K.Pang, for all his advice and help with selecting the right electronic components for our projects, and for helping us to make it possible to easily and cheaply control electronic circuits from the PC and the Mac.
- Cliff O'Reilly, for making sure everything was technically right, and for testing everything 3 times for us (once each on the 3 different computer platforms).
- Sarah Wright, for the truly amazing illustrations throughout this book. They are beautiful pieces of visual artwork, and cleverly and perfectly capture the concepts being presented in each adventure.
- Ben Ramachandra, the young lad at the Christmas 2013 Fire Tech Camp event at Imperial College, London: You were so determined to follow the Python course entirely in Minecraft, which was the moment that caused the idea for this book to spark into existence!
- Roma Agrawal, structural engineer for The Shard, UK: for her suggestions and links to inspiring tall buildings in Adventure 4 and the Bonus Adventure – let's hope we see some amazing creations from our readers!
- Last, but not least, we would like to thank Carrie-Anne Philbin, for having the vision and determination to write her first book *Adventures in Raspberry Pi*, without which the Adventures series of books would not exist—now, see what you've started, Carrie-Anne?!

Contents

Introduction 1

What Is Minecraft?	1
The Virtual World	2
How Did Minecraft Come About?	2
What Is Minecraft Programming?	2
Who Should Read This Book?	3
What You Will Learn	4
What We Assume You Already Know	5
What You Will Need for the Projects	5
A Note for Parents and Teachers	6
How This Book Is Organised	7
The Companion Website	8
Other Sources of Help	9
Conventions	9
Reaching Out	11

Adventure 1

Hello Minecraft World 13

Setting up Your Raspberry Pi to Program Minecraft	15
Installing Minecraft on Your Raspberry Pi	16
Starting Minecraft on Your Raspberry Pi	17
Setting up Your PC or Apple Mac to Program Minecraft	19
Installing the Starter Kit and Python on Your Windows PC	20
Installing the Starter Kit and Python on Your Apple Mac	22
Starting Minecraft on Your Windows PC or Apple Mac	24
Stopping Bukkit	27
Creating a Program	28
Running a Program	30
Stopping a Program	33

Adventure 2

Tracking Your Players as They Move 35

Sensing Your Player's Position	36
Getting Started	38
Showing Your Player's Position	38
Tidying Up Your Position Display	41

Using postToChat to Change Where Your Position Displays.....	43
Introducing a Game Loop	43
Building the Welcome Home Game.....	45
Using if Statements to Make a Magic Doormat.....	46
Checking if Your Player Is at a Particular Location	47
Building a Magic Doormat	48
Writing the Welcome Home Game	49
Using Geo-Fencing to Charge Rent.....	53
Working out the Corner Coordinates of the Field.....	54
Writing the Geo-Fence Program.....	56
Moving Your Player	58
Further Adventures in Tracking Your Player.....	61

Adventure 3

Building Anything Automatically **63**

Creating Blocks.....	64
Building More than One Block.....	66
Using for Loops	67
Building Multiple Blocks with a for Loop	67
Building a Huge Tower with a for Loop	69
Clearing Some Space	71
Using setBlocks to Build Even Faster.....	71
Reading Input from the Keyboard	72
Building a House	74
Building More than One House.....	79
Using Python Functions	80
Building a Street of Houses with a for Loop.....	83
Adding Random Carpets.....	85
Generating Random Numbers	85
Laying the Carpets	86
Further Adventures in Building Anything	89

Adventure 4

Interacting with Blocks **91**

Finding Out What You Are Standing On	92
Finding out if Your Feet Are on the Ground.....	92
Building Magic Bridges	95
Using Python Lists as Magic Memory	98
Experimenting with Lists	98
Building Vanishing Bridges with a Python List	101

Sensing that a Block Has Been Hit	105
Writing a Treasure Hunt Game	108
Writing the Functions and the Main Game Loop	109
Placing Treasure in the Sky	110
Collecting Treasure when It Is Hit	111
Adding a Homing Beacon	112
Adding Your Bridge Builder	113
Further Adventures in Interacting with Blocks.....	115

Adventure 5

Interacting with Electronic Circuits 117

What You Will Need for this Adventure	118
Prototyping Electronics with a Breadboard	121
Building a Circuit that Lights an LED	123
Connecting Electronics to Your Computer	124
Setting Up the PC or Mac to Control Electronic Circuits.....	125
Configuring the Drivers	126
Finding the Serial Port Number.....	127
Controlling an LED	128
Lighting Up an LED from your Computer	129
Flashing the LED	132
Running a GPIO Program	134
Writing the Magic Doormat LED Program.....	137
Using a 7-Segment Display.....	138
What is a 7-Segment Display?	138
Wiring Up the 7-Segment Display.....	140
Writing Python to Drive the 7-Segment Display.....	142
Using a Python Module to Control the Display.....	144
Making a Detonator.....	145
Wiring Up a Button	146
Writing the Detonator Program.....	148
Further Adventures in Electronic Circuits	152

Adventure 6

Using Data Files 155

Reading Data from a File.....	155
Interesting Things You Can Do With Data Files.....	156
Making a Hint-Giver	156

Building Mazes from a Data File	160
Understanding CSV Files.....	160
Building a Maze.....	162
Building a 3D Block Printer	168
Hand-Crafting a Small Test Object to 3D Print.....	169
Writing the 3D Printer.....	171
Building a 3D Block Scanner	174
Building a Duplicating Machine.....	178
Writing the Framework of the Duplicating Machine Program	178
Displaying the Menu	182
Building the Duplicator Room	183
Demolishing the Duplicator Room	185
Scanning from the Duplicator Room.....	186
Cleaning the Duplicator Room	187
Printing from the Duplicator Room.....	187
Listing Files	189
Further Adventures in Data Files	191

Adventure 7

Building 2D and 3D Structures..... 193

The minecraftstuff Module.....	194
Creating Lines, Circles and Spheres.....	194
Drawing Lines	196
Drawing Circles	197
Drawing Spheres.....	199
Creating a Minecraft Clock.....	200
Drawing Polygons.....	206
Pyramids	209
Further Adventures with 2d and 3d Shapes	213

Adventure 8

Giving Blocks a Mind of Their Own..... 215

Your Block Friend.....	215
Using Random Numbers to Make Your Block Friend More Interesting	222
Bigger Shapes	225
Alien Invasion.....	228
Further Adventures in Simulation	235

Adventure 9

The Big Adventure: Crafty Crossing 237

A Game within a Game	237
Part 1—Building the Arena	239
Part 2—Creating the Obstacles	243
The Wall	243
Running More Than One Obstacle	246
Building the River	249
Creating the Holes	252
Part 3—Game Play	256
Starting the Game	257
Collecting Diamonds	259
Out of Time	261
Tracking the Player	263
Setting the Level as Complete and Calculating Points	264
Adding the Game Over Message	265
Part 4—Adding a Button and Display	266
What You Will Need	266
Set Up the Hardware	267
Diamond Countdown	269
Time-Left Indicator	270
Further Adventures in Your Continuing Journey with Minecraft	271

Appendix A

Where to Go from Here 273

Websites	273
Minecraft	273
Python	275
Bukkit	275
Other Ways to Make Things Happen Automatically	276
Projects and Tutorials	276
Videos	277
Books	278

Index 279

FOREWORD

It was another busy IT lunch club at school. I've always been happy to give up my lunch break to allow the pupils an opportunity to catch up on homework, research on the web, email their teachers and print out assignments. As I walked along the rows of computers I saw the "hard work" in progress: multi-coloured birds being catapulted into towers, aliens being zapped into submission and one pupil trying to reverse a car round a car park—at least that one might have some educational benefit!

My philosophy has always been that the more time kids spend on the computers, the more natural the computers become to them; even a game will improve their hand/eye coordination and familiarisation with the keyboard. They become adept at logging in, starting the browser, and finding the shortest number of key terms possible for the search engine. They can be into the room and onto a game in the time it takes me to prop the door to the IT lab open.

As I patrolled the aisles, ever eager to find the one child in a dozen that was actually doing something productive, I spotted a pupil diligently stacking what looked like little 3D cubes on top of each other in their game window. This pupil seemed enthralled, and as I paused, curious as to why this comparatively basic endeavour had captured this child's attention, a house emerged from the little collection of blocks that had been stacked.

This was my introduction to Minecraft: a little architect in the making, a pupil who could hide himself away in the chaos of an IT room at lunch and disappear into a world of his own creation: a virtual home he designed, created and explored. Kids' games are infectious little things and once it reaches a critical mass, once enough have bought into an idea, they're all at it. Once it gains momentum, they need to get on board to keep up with the rest of the crowd and by summer they were all on Minecraft.

However, unlike a lot of the passing fads I soon discovered that there was a lot more to this game than the rest. I found out students can automate their constructions. By wiring up their houses they can create their own lighting systems and elevators. They can devise hidden staircases that appear from inside walls. In fact they can create practically any contraption they can imagine. A history lesson overspills into lunch club as they build the castles they just learned about, biology class means they're all competing to build the best virtual skeleton, and a geography lesson destroys all that hard work when their simulated volcano erupts lava over both.

When our school purchased a MinecraftEdu account, my pupils could work collaboratively. They've built giant replicas of the school logo, they devised a maze, and held a school contest to escape it. They measured the classrooms and built the replica school inside Minecraft.

More importantly though, this has given me a virtual classroom; a strange state of affairs where I fly above the avatars of my class, instructing them on the finer details of

logic. I start by building an exclusive-OR gate using the blocks of this world, and then set them a team-work task of building a calculator by wiring several of these circuits together.

To my delight I learned that Minecraft is also available on the Raspberry Pi. But more importantly, much more importantly, it can be programmed in the Python programming language. My pupils began work on it immediately. Through for loops and block placement we created platforms and trampolines, fireballs and drag races—games within a game that did things even the original game designers had not imagined—they had made the world their own.

One of my pupils, left to his own devices, created an auto-house script that allows the user to generate a home around themselves; another created a mine-sweeper puzzle. Writing computer programs was suddenly something they were all desperate to learn, driven by the need to build and automate larger and more complex tasks inside the Minecraft world, and to compete and keep up with what all their friends were inventing.

Our school now has a purpose-fitted Minecraft network with its own server, built by the pupils, for the pupils. With the help of their network they can submit work for grading in exactly the same way that their coursework is assessed. They can compete against each other to code individual parts of a larger world and then explore it together.

This book is a first step into that amazing world. It teaches coding, and it teaches it using one of the most popular games on the planet! Children are gradually drawn outwards from the Minecraft world they are used to, and into the world of computer coding; they have a real purpose to learn coding, to achieve things that they are desperate to learn quicker than their friends can.

It is the ability to see an instant visualisation of the results of their code that makes it so powerful. Programming can be dry and uninteresting to pupils, unable to relate to the outcome of their code when the results are often just text on a screen. Minecraft enables them to see the outcome of their code in a context they understand and enjoy.

I genuinely believe that Minecraft has the power to influence an entire generation; to encourage coding for what it should be—fun and hugely rewarding. So, don't waste any time, get started on your own Adventures in Minecraft with Martin and David to guide you along the way. Be creative, stay excited, build something awesome, and learn how to code along the way!

Ben Smith BSc (Hons)

Head of Computing,

Arnold Keqms, Lytham St. Annes.

Introduction

ARE YOU AN adventurer? Do you like to try new things and learn new skills? Are you a huge fan of Minecraft? And would you like to push the boundaries of what you can do in Minecraft by learning how to write computer programs that interact with your game, and amaze your friends with your creativity and magic? If the answer is a resounding “Yes!” then this is the book for you.



What Is Minecraft?

Minecraft is a sandbox indie game, where you build structures, collect items, mine minerals and fight monsters in order to survive. It appears to you as a 3D virtual world made of different types of blocks, each block having its own place inside the grid layout of the 3D virtual world. Figure 1 shows an example of the Minecraft world.

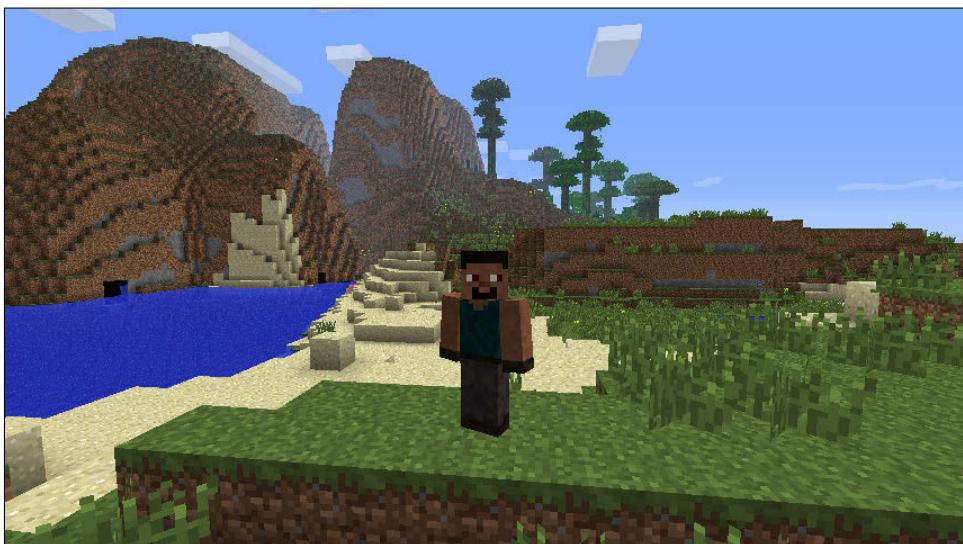


FIGURE 1 The Minecraft world

The Virtual World

In a sandbox game, you are a player inside a virtual world (a sandbox with very distant edges, like a playpen filled with sand). Instead of being offered levels in a pre-set order, you roam around the virtual world and make your own choices about what goals you want to achieve and how to set about them. Because you are making your own choices right from the start, sandbox games have limitless possibilities. You make up your own stories and move through the 3D world, learning new skills and features by discovering them by chance and experimentation.

In Minecraft your player, or avatar, is called Steve. You direct Steve through the sandbox virtual world to achieve whatever mission you decide. If you are successful in surviving your first night against the monsters, you can follow your own enthralling missions to interact with other participants of the game and build huge structures limited only by your imagination.

A sandbox game allows you, the player, to make your own decisions about playing the game, rather than being forced down a specific route by the game designers. You can read more about this type of game design here: http://en.wikipedia.org/wiki/Open_world. There is a little bit of mystery about why the player is called Steve, but you can read more about it here: http://minecraft.gamepedia.com/The_Player.

How Did Minecraft Come About?

Indie games are “independent video games”, created by individuals or small teams. They are often developed without any funding or support from a games publisher. As a result of their independent nature, indie games are often more innovative than other, more mainstream games. According to Wikipedia, Minecraft was created by the Swedish computer programmer Markus Persson, who is known by the gamer tag “Notch”. He first demonstrated Minecraft as an early version in 2009, and the first official release of the game took place in 2011. Notch founded a Swedish company called Mojang AB, which continues to develop the Minecraft game on many computer platforms, including PC, Mac, Raspberry Pi, Linux, iOS, Android, Xbox 360, Playstation and Wii.

You can find out more about the fascinating Minecraft story in a documentary film called *Minecraft: The Story of Mojang* (http://en.wikipedia.org/wiki/Minecraft:_The_Story_of_Mojang).

What Is Minecraft Programming?

This a book about computer programming—it uses Minecraft as a way to teach you about computer programming. If you are looking for some helpful tips on how to build structures and fight combat, there are some other great books on the market listed in Appendix A that will help.

By programming Minecraft, you make your gaming experiences even more exciting, creative, and individual. As you play the normal game, you follow the basic rules of the Minecraft game as set out by the game designers. By writing programs that interact with the Minecraft game world, you can make complex and repetitive tasks—like building huge streets of houses and large structures—automatic. You can make the game and the objects inside it behave in new ways, and invent new things that even the original creators of the game didn’t think of. But most of all, you will learn a general skill—how to program using the Python programming language. You will then be able to apply this to all sorts of other things, not just Minecraft. Figure 2 shows a huge street of houses that was built automatically by a short Python program.

In a recent video about why all children should learn programming (www.youtube.com/watch?v=nKlu9yen5nc), Will.i.am is quoted as saying “great coders are today’s rock stars”. The new skills you learn while following the adventures in this book will make your Minecraft experiences more personal, more creative, more ambitious. Your new wizardry with programming will amaze your friends and fellow gamers and inspire them to ask you what magic you used to achieve such amazing feats. The answer, of course, is the magic of computer programming.



FIGURE 2 A huge street of houses, built by a 20-line Python program

Who Should Read This Book?

Adventures in Minecraft is for any young person who loves playing Minecraft, and would like to learn to program and do more with it. The Adventures series of books is aimed at readers in the age range 11–15, but some of the more challenging later adventures might be appropriate for older readers too. The earlier chapters have also been tested with readers as young as 8.

You might already be an expert in playing the game but find yourself getting frustrated by the length of time it takes to build new structures. Or you might want to find ways to extend the game by adding some additional intelligence and automation to the world. Whatever your reasons, this book will be your guide for a journey through Minecraft programming; and as every adventurer knows, your guidebook is the most important item in your backpack. Your trek will take you from simple beginnings, such as posting messages to the Minecraft chat, through learning the basics of programming Minecraft using the Python programming language, to discovering how to use your new computer programming skills to program your own exciting game inside Minecraft. By the end of your adventures you will have learned the skills you need to become a pioneer in Minecraft programming!

What You Will Learn

You will learn about many aspects of the Minecraft game and how to interact with Minecraft features through the Python programming language. You will discover how blocks are addressed in the 3D world using coordinates, how to sense the position of your player, how to create and delete blocks in the Minecraft world, and how to sense that a block has been hit by the player.

If you are using a Raspberry Pi, you will learn how to install and use the Minecraft programming interface that comes bundled with Minecraft Pi edition. If you are using a PC or a Mac, you will learn how to set up and run your own local Minecraft server using the community developed craft-bukkit server, and how to program it using the same Minecraft programming interface as the Raspberry Pi through the Raspberry Juice plug-in.

You will learn how to write programs in the Python programming language, from the very beginnings of a Hello Minecraft World program to the creation of and interaction with huge 3D objects that, thanks to your new Python programming skills, you can stamp with your own personality.

Using the free MinecraftStuff module of pre-written Python helper code, you will be able to enhance your ability to create both 2D and 3D objects out of blocks, lines, polygons and text.

Your adventures will not be limited to the virtual world of Minecraft though! We will introduce you to ways to connect Minecraft to electronic components, meaning that your Minecraft world will be able to sense and control objects in the real world. Thus, we give you a valuable secret: how to break out of the boundaries of the virtual sandbox world!

Minecraft has two main modes of working: Survival mode and Creative mode. You will be using Creative mode throughout this book. We won't be covering Survival mode (mainly because it's extremely frustrating when a creeper kills you just as you are watching your program running). There are many good books already on the market that explain how to survive the night in Minecraft, and we give links to those and other resources in Appendix A at the back of this book. However, any programs you create in Creative mode will also work in Survival mode.



What We Assume You Already Know

Because this is a book about programming with Minecraft and we want to focus on learning the programming aspects of Minecraft, we have to assume a few things about you the reader and what you already know:

1. You have a computer (a Raspberry Pi running Raspbian, a PC running Microsoft Windows, or an Apple Mac running Mac OS X), which meets the minimum requirements for running Minecraft, and is already set up and working.
2. You have a basic understanding of how to use your computer, such as using a keyboard and a mouse, using the menu system to start programs, and using application menus like File→New→Save.
3. You have a working connection to the Internet, and you know how to use a web browser to download files from the Internet.
4. If you are using a PC or a Mac, you already have a Minecraft user ID and a working copy of Minecraft installed.
5. You know how to play the Minecraft game, such as how to start it, how to move around, how to choose items from the inventory, and how to create and delete blocks in the world.

Because this is a book about programming Minecraft, we don't assume you have any prior knowledge about how to program. As you progress through your adventures, we will lead you through the steps needed to learn programming.

What You Will Need for the Projects

We have written this book to work on three commonly available computers: the Raspberry Pi running Raspbian, a PC running Microsoft Windows, and an Apple Mac

running MacOS X. Minecraft is supported on other platforms too, such as a PC running various flavours of Linux, but we don't cover the set-up of those platforms in this book.

To make the set-up of the various parts simpler, we have prepared three starter kits, one for each of the supported computer platforms. You can download the correct starter kit for your computer from the Wiley website, and in your first adventure we provide step-by-step instructions about how to download and install these and get everything working. These starter kits include everything you need, except the actual Minecraft game itself. You'll be up and running in no time!

You will need an Internet connection on your computer in order to download the starter kits. Almost everything you need for the adventures is included in the starter kits. A few of the adventures have special requirements and we note these at the start of the adventure so you can get everything prepared before you start.

In Adventures 5 and 9, we show you how to connect small electronic circuits together to link the Minecraft virtual world to the real world. For this you will need to buy a small collection of electronic components, which are available from most electronics components stockists. (We provide some links to these in Appendix A.)

The Raspberry Pi has built-in input/output pins, so you can connect your electronic components directly to these. Because PCs and Macs don't include input/output pins, we have chosen a small affordable plug-in board that works via the USB connection of your computer for these projects. Again, there are links in Appendix A to outlets where you can buy this.

The most important things you need on this journey are your own excitement and enthusiasm for Minecraft, and some curiosity and willingness to experiment with your own ideas and push the boundaries of what you already know!

A Note for Parents and Teachers

We have split this book into separate self-contained adventures that you can treat as individual standalone projects, each of which focuses on one specific feature of Minecraft programming. The Python language is introduced gradually and progressively throughout each adventure; the early adventures are aimed chiefly at beginners, with the later adventures becoming more challenging and introducing more Python, stretching the reader a bit more.

Each adventure presents a practical project with step-by-step instructions (that readers can tick off as they complete them), delivered in a descriptive style, very much like a well-commented program listing. Detailed explanations appear in Digging into the Code sidebars that students can read later, meaning that they are not distracted from the progress of typing in and trying the programs.

Each adventure will probably take more than one session to complete, but they are all split into sections, with subheadings at logical points that could be used to provide a goal for an individual lesson, or an activity to be stretched over a number of sessions.

The Python language uses indents on the left-hand side of the program to represent code structure, and it is a case-sensitive language. Extra guidance from an adult may be useful sometimes with very young readers, to make sure they are being careful to use case and indents correctly, thus avoiding the possibility of them introducing errors into their programs. All of the programs are downloadable from the companion website, so if you have problems with indentation you can check our versions of the programs to see where you might have gone wrong.

Many schools have Python version 3 installed. At the time of writing, Mojang have not released a version of the Minecraft programming interface that works with Python version 3, so you should use Python version 2 as explained in Adventure 1.

How This Book Is Organised

Every chapter of the book is a separate adventure, teaching you new skills and concepts as you program and test the projects. The book is organised so that each adventure is a standalone project, but you might find it easier to work through them in order, as we build up your understanding of the programming concepts gradually throughout the book.

It is vital that you do Adventure 1 before doing anything else. This is because it shows you how to download and install everything you need, and to check that it all works properly. We introduce some basic steps in this adventure that you need to know how to do in all the other adventures, but will give you some reminders in the earlier adventures as you get started.

The first three adventures are written for beginners who have little or no programming knowledge, and we explain all the jargon and concepts as you work through them. In Adventures 2, 3 and 4, you cover the key parts of any good Minecraft game. These include *sensing* things that happen in the Minecraft world, doing some *calculations* with some simple maths, and making your programs *behave* differently, for example by displaying a message on the chat or automatically creating blocks in the world. You will use these three concepts of *sensing*, *calculating* and *behaving* throughout the book to build bigger and more exciting Minecraft programs!

Adventures 5 and 6 build on what you learned in the earlier adventures and explore some exciting ways of linking the Minecraft virtual world to the real world. You will experiment with the exciting topic of physical computing by building some small electronic circuits that cause things to happen inside Minecraft and respond to things that happen in Minecraft. There is an abundance of exciting ideas and games you could create using

this as a basis! Adventure 6 looks at ways you can bring in large amounts of data from data files to save and duplicate large structures with a 3D “duplicating machine”.

Adventures 7 and 8 introduce the free MinecraftStuff module, which makes it possible to use blocks to build lines, circles and other 2D shapes, and also some fantastic 3D spheres and pyramids. These can form the beginnings of huge structures that would be very hard to build by hand. Adventure 8 shows how you can add personalities to moving objects to give them their own intelligence. With these techniques, you can write some exciting “games inside a game” that will amaze your friends.

Adventure 9 draws on all the programming concepts and skills from the earlier adventures to create one final big project—an awesome game with scoring, and moving objects that you have to avoid or carry around with you. In this adventure, you also have the option to experiment with physical computing by using electronic components, allowing you to do things in the game by pressing buttons in the real world.

Appendix A (“Where Next”) suggests a whole range of resources that you can use to extend and enhance your adventures, learn more about programming in Python and create even more awesome Minecraft programs based on what you have learnt throughout this book.

In Appendix B (“Quick Reference”) we have included a comprehensive reference guide to the programming features used throughout the book, along with a reference to the programming statements that are specific to Minecraft, and a table of block types that you can build with. You’ll find this is an invaluable reference section to help with all your own projects and inventions as well!

The glossary provides a handy quick reference to all the jargon and terminology we have introduced throughout the book, and is a collection of all definitions from each adventure.

The Companion Website

Throughout this book you’ll find references to the Adventures in Minecraft companion website at www.wiley.com/go/adventuresinminecraft. The website is where you’ll find the starter kits you will need to start programming in Minecraft, together with a collection of video tutorials we have put together to help you if you get stuck. Code files for some of the bigger projects can also be found on the website.

You will also find on the companion website a complete extra bonus adventure! With this adventure you will build a fully functional passenger lift simulation inside Minecraft allowing you to whizz up and down through the world. This bonus adventure is quite challenging and should exercise all of the skills that you have learned and practiced throughout this book, including controlling your lift with electronic circuits.

Our publisher (Wiley) has kindly allowed us to provide a reference appendix in a PDF format that you can download from the companion website. Keep it by your side as a handy reference as you work through these Minecraft adventures. You can also use it in any programming projects you embark on in the future. The Wiley website also includes a glossary. Although definitions are included in the adventures themselves, anytime you want to look up a word, you can go online.

Other Sources of Help

Computers are complex devices, and operating systems and software are changing all the time. We have tried to protect you and your adventures from future changes as much as possible by providing a downloadable starter kit in Adventure 1 that should give you most of what you need. However, if you run into problems or need specific help, here are some useful places to go:

- Sign up for a user ID and downloading and installing Minecraft: <http://minecraft.net>
- Play the Minecraft game: http://minecraft.gamepedia.com/Minecraft_Wiki
- Raspberry Pi: www.raspberrypi.org
- Microsoft Windows: <http://support.microsoft.com>
- Apple Mac and Mac OS X: www.apple.com/support
- The Python language: www.python.org
- The IDLE programming IDE: <https://docs.python.org/2/library/idle.html>
- Minecraft Pi edition: <http://pi.minecraft.net>
- Craft-bukkit server: <http://wiki.bukkit.org>
- Raspberry Juice bukkit plug in: <http://dev.bukkit.org/bukkit-plugins/raspberryjuice>

Conventions

You'll notice that there are special boxes throughout this book, to guide and support you. Here is what they look like:

These boxes explain concepts or terms you might not be familiar with.





These boxes give you hints to make your computer-programming life easier.



These boxes contain important warnings to keep you and your computer safe when completing a step or a project.



These boxes feature quick quizzes for you to test your understanding or make you think more about a topic.



These boxes allow us to explain things or give you extra information we think you'll find useful.



These boxes point you to videos on the companion website that will take you through the tasks, step by step.

You will also find two sets of sidebars in the book. Challenge sidebars give you extra tasks you can accept if you want to take the project a bit further, perhaps by making changes or adding new features. Digging into the Code sidebars explain in a bit more detail some concept or feature of the program, to give you a better understanding of

the programming language Python. These sidebars mean you can focus on getting the programs working first, and then read in more detail about how they work and ways you can extend them further once they are working.

When you are following our steps or instructions using code, you should type the code in exactly as we have described it in the instructions. Python is a language where the amount of space at the start of the line (the indent) is important to the meaning of the program, so take extra special care to make sure you put enough spaces at the left of each line. We have coloured the code listing boxes for you so that it makes it easier to see how much each line needs to be indented. Don't worry too much about it—we explain indenting in the early adventures when you first need to use it.

Sometimes you need to type a very long line of code, longer than will fit on a single line in this book. If you see ↵ at the end of a line of code, it means that line and the following line are part of a single line of code, so you should type them as one line, not separate lines. For example, the following code should be typed on one line, not two:

```
print("Welcome to Adventures in Minecraft by ↵  
Martin O'Hanlon and David Whale")
```

If you are viewing this book on an e-reader, to make sure that the programs you type in are correctly laid out, please set your e-reader font size smaller. This is so that the program listings are not unnecessarily wrapped around the page margins, and to prevent errors being introduced into your programs.



Most adventures include a Quick Reference Table at the end to sum up the main programming statements or concepts. You can refer to these guides when you need a refresher. There is also a reference section in Appendix B, which shows you the most important programming statements for Minecraft and Python. We hope you will find this handy to refer to as you progress through your adventures.

Whenever you complete an adventure, you unlock an achievement and collect a new badge. You can collect the badges to represent these achievements from the Adventures in Minecraft companion website (www.wiley.com/go/adventuresinminecraft).

Reaching Out

In Appendix A you will find ways to take your Minecraft programming knowledge further, with lists of websites, organisations, videos and other resources. Many of these resources include forums where you can ask questions or get in touch with other Minecraft programmers.

You can also contact the authors by sending us a message through our websites:

Martin: www.stuffaboutcode.com

David: <http://blog.whaleygeek.co.uk>

Time to start your adventures!





Adventure 1

Hello Minecraft World

IN THIS BOOK you are going to learn how to write programs that interact with your Minecraft world, allowing you to do some very exciting things. You will be using a programming language called **Python** to do it. This way of controlling the Minecraft world from a Python program was first created for Minecraft: Pi Edition on the Raspberry Pi. If you don't have a Raspberry Pi but have Minecraft for Windows or Apple Mac instead, that's ok—you will just need to do some extra work on the set-up before you get started, which you'll be shown how to do.

Python is the programming language used in this book.



This book is full of adventures that teach you how to write programs for the Minecraft game. It's packed with all sorts of things you can do with Minecraft to entertain your friends and make the game even more fun to play. You will discover some pretty flashy ways to move your player around and before long you'll be finding it easy to build whole cities and Minecraft creations that have never been seen before.

The Python programming language comes with a code editor called IDLE, and you will be using this to create, edit and run the programs created in these adventures.



The Python programming language is used throughout the world in business and education. It is extremely powerful but also easy to learn you can find out more about Python at www.python.org.

When computer programmers learn a new programming language or a new way of doing something, they always start by writing a “hello world” program. This is a really simple program that displays “hello world” on the screen, to make sure everything is installed and working properly.

In this first adventure, you will set up your computer to allow you to write a program that displays the text “Hello Minecraft World” on the Minecraft chat (see Figure 1-1).

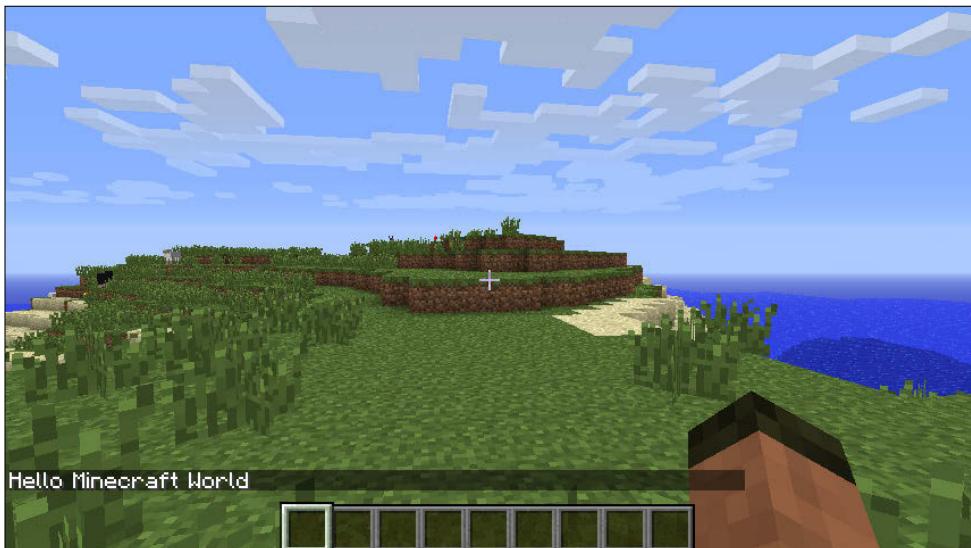


FIGURE 1-1 Hello Minecraft World

To do the Minecraft programming in this book, you need one of these three types of computer: a PC running Microsoft Windows; an Apple Mac running MacOS X; or a Raspberry Pi running Raspbian. The way you set up your computer will be different depending on which sort you have but, once you have set it up, you program Minecraft in exactly the same way on all of them. To make it easier for you to set up your computer, you can download a starter kit from the book’s companion website (www.wiley.com/go/adventuresinminecraft). The starter kits have been tested to make sure all the Adventures in this book work properly. You’ll see that your starter kit contains a **README** file, which you should have a look at. It describes what the kit contains and

how it was created; you could conceivably use this information to set up your own computer from scratch, although this is not recommended. You'll get a lot more out of it by following the instructions in this book.

Make sure you follow the instructions for your type of computer, either “Setting Up Your Raspberry Pi for Programming Minecraft” or “Setting Up Your PC or Apple Mac for Programming Minecraft”.

It's essential that you to set up your computer right, otherwise you could get yourself into quite a muddle. So please make sure that you follow the instructions very carefully.



Setting up Your Raspberry Pi to Program Minecraft

If you are using Raspberry Pi, there are two steps to get your computer setup before you can create your first Minecraft program:

1. Download and install Minecraft: Pi Edition—this is a special version of Minecraft just for the Raspberry Pi.
2. Download and extract the starter kit for Raspberry Pi. This contains everything you need to complete *Adventures in Minecraft* in a folder called MyAdventures; this is where you will also save your Minecraft programs.

To see a video of how to set up your Raspberry Pi, visit the companion website at www.wiley.com/go/adventuresinminecraft.



The Raspberry Pi's graphical user interface (GUI), known as x windows, is used throughout *Adventures in Minecraft*. The GUI is installed on Raspbian, but depending on how you have set up your Raspberry Pi, it may not load the GUI when it boots up. You may instead start with a login and command prompt.

If your Raspberry Pi is set up to start at a command prompt, you will need to login then type **startx** and press Enter to load the GUI when the command prompt appears.



Start *Adventures in Minecraft* with a new installation of Raspbian so you can be sure that your Raspberry Pi is set up correctly. Visit www.raspberrypi.org/help/ for information on setting up your Raspberry Pi and installing Raspbian.

Installing Minecraft on Your Raspberry Pi

Once your Raspberry Pi has booted up and the GUI has started, you can install Minecraft: Pi Edition. The Minecraft: Pi Edition installation file can be downloaded from <http://pi.minecraft.net> where you can also find instructions on how to extract the game from the file and run it.

Follow these steps to download and extract Minecraft: Pi Edition:

1. Double-click the LXTerminal icon (it looks like a black computer screen) on the desktop. The LXTerminal window opens so you can type.
2. Type the following into the LXTerminal window one at a time, pressing Enter after each statement to download Minecraft: Pi Edition installation file:

```
cd ~  
wget https://s3.amazonaws.com/assets.minecraft.net/pi←  
/minecraft-pi-0.1.1.tar.gz
```

A progress bar will appear and show the percentage of the file downloaded.

3. To extract Minecraft: Pi Edition, type:

```
tar -zxvf minecraft-pi-0.1.1.tar.gz
```

Minecraft: Pi Edition will be extracted to a folder called `mcpi`.

Next you need to download the starter kit for Raspberry Pi and extract the `MyAdventures` folder by following these steps:

1. Open your web browser (e.g. Midori), go to the companion website (www.wiley.com/go/adventuresinminecraft) and download the starter kit for Raspberry Pi.
2. A window opens, asking whether you want to open or download. Click Open to open a program called Xarchiver that will extract the files.
3. Click Action→Extract on the Xarchiver menu.
4. Type `/home/pi` in the Extract to: text box (as shown in Figure 1-2).

- Click Extract. Your files will be extracted and saved in the `/home/pi` folder, where you can access them any time you want to use them.

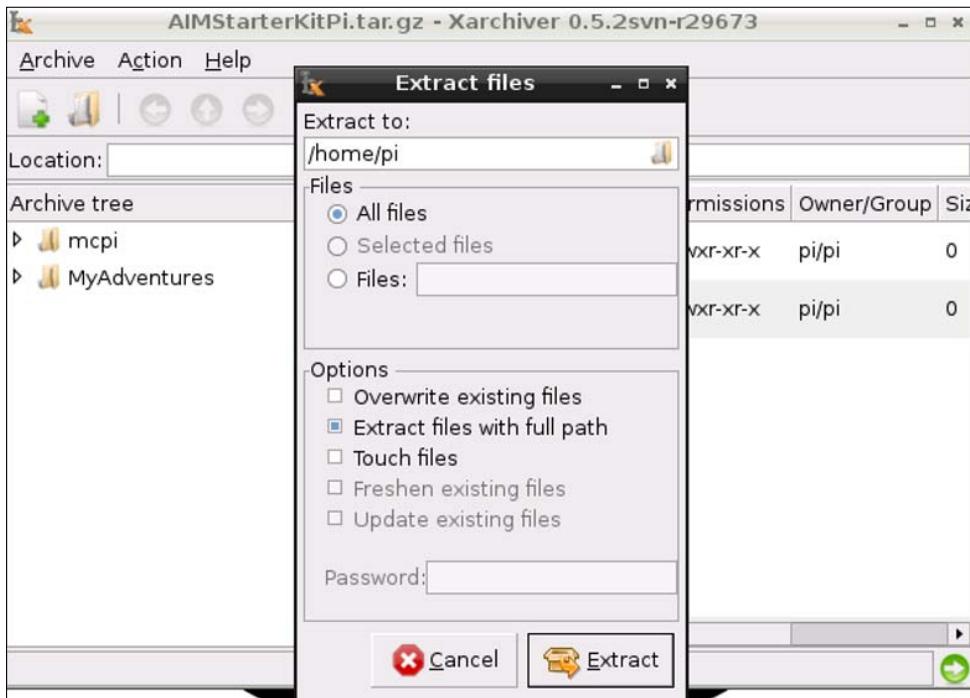


FIGURE 1-2 Choose a location and then extract your Raspberry Pi starter kit.

Starting Minecraft on Your Raspberry Pi

Now that you have downloaded and installed Minecraft: Pi Edition, run the game and have a go before moving onto creating your first program.

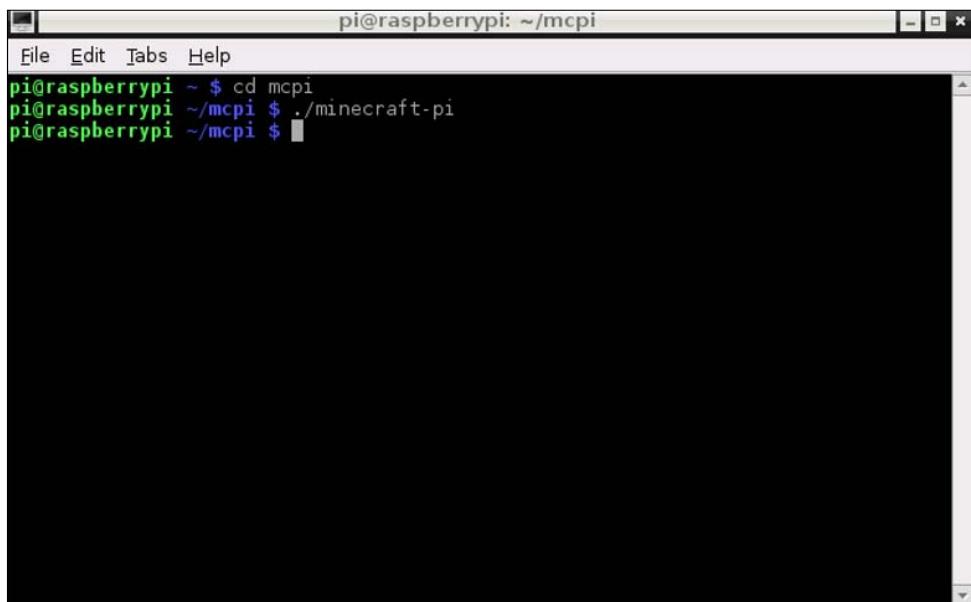
In future adventures, the instructions will tell you to start Minecraft. If you ever need a reminder of how start Minecraft on the Raspberry Pi, just refer back to this section.



To start Minecraft, follow these steps:

1. Double-click the LXTerminal icon on the desktop. The LXTerminal window will open so you can type into it.
2. To run Minecraft, type the following (as shown in Figure 1-3):

```
cd ~  
cd mcpi  
./minecraft-pi
```



```
pi@raspberrypi: ~/mcpi  
File Edit Tabs Help  
pi@raspberrypi ~ $ cd mcpi  
pi@raspberrypi ~/mcpi $ ./minecraft-pi  
pi@raspberrypi ~/mcpi $
```

FIGURE 1-3 To run Minecraft, enter some simple statements.

3. That's it! Now you can start playing Minecraft. Click Start Game→Create New to start a new Minecraft world.

The main menu has two options: Start Game to build a new or enter an existing Minecraft world and Join Game to join another player's Minecraft: Pi Edition world.

Once your Raspberry Pi is set up and you've got Minecraft running, you can skip the next section (unless you want to set up Minecraft on a PC or Mac as well as on your Raspberry Pi) and go straight to the section headed "Creating a Program" later in this chapter.

Setting up Your PC or Apple Mac to Program Minecraft

Whether you are using a Windows PC or a Mac, you need to make sure Minecraft is installed and working on your computer. If you don't have a copy of Minecraft and a user id to play it, visit www.minecraft.net to purchase the game. If you encounter any problems installing, running or playing Minecraft, help is on hand—just visit <https://help.mojang.com>.

To program the full version of Minecraft on the PC and Apple Mac, in the same way as the Raspberry Pi, you will need to use the open source Minecraft server, **Bukkit** (<http://bukkit.org>), and the RaspberryJuice plugin (<http://dev.bukkit.org/bukkit-plugins/raspberryjuice>).

Bukkit is a Minecraft server that people can change by creating plugins to make the game do different things; the RaspberryJuice plugin for Bukkit allows you to write programs to change Minecraft in the same way as you can on the Raspberry Pi. You will be using Bukkit, the RaspberryJuice **plugin** and the Python programming language to create your Minecraft programs.

Bukkit is a Minecraft sever that allows people to modify the game through plugins.



A **plugin** is a program that runs inside the Bukkit server and lets you modify Minecraft.



You need to download the Python programming language and install it on your computer. Throughout *Adventures in Minecraft* you will be using Python version 2, even though version 3 is the latest version, this is because the Python library supplied by Mojang (which is used to connect to Minecraft) only works with version 2. The programs within *Adventures in Minecraft* have all been tested to work with Python version 2.7.6, while it is not essential you use this version, it is recommended and you must use Python 2.something.



If you want to find out more about Python, visit www.python.org. You can download Python from www.python.org/download and the Python Wiki, wiki.python.org, contains lots of information, tutorials and links to Python community websites.

Setting up your Windows PC or Apple Mac to create your first Minecraft program requires three steps:

1. Download and extract the PC or Apple Mac starter kit, which contains a pre-configured Bukkit server with the RaspberryJuice plugin and a folder called **MyAdventures** where you will save your Minecraft programs.
2. Download and install the Python programming language.
3. Configure Minecraft to use the same version as Bukkit and connect it to the Bukkit server.



To see a video of how to set up your Windows PC or Apple Mac, visit the companion website at www.wiley.com/go/adventuresinminecraft.

Installing the Starter Kit and Python on Your Windows PC

If you are using a Windows PC, the next part in setting up your computer is to:

1. Download the starter kit for Windows PC and extract it to your desktop.
2. Download the Python programming language and install it on your Windows PC.

Downloading and Extracting the Starter Kit

Follow these steps to download the Windows PC starter kit and copy the contents to your desktop. Placing them on your desktop will make it easy for you to find them whenever you need them:

1. Open your PC's web browser (e.g. Internet Explorer, Chrome), go to the companion website (www.wiley.com/go/adventuresinminecraft) and download the starter kit for your Windows PC.

- When the starter kit zip file has finished downloading, open it. You can do this either by choosing the Open File option or by opening the download folder and double-clicking the **AIMStarterKitPC.zip** file.
- The zip file contains only one folder, called **AdventuresInMinecraft**. Copy this folder to the desktop by clicking the folder, holding down the button and dragging it to the desktop (see Figure 1-4).

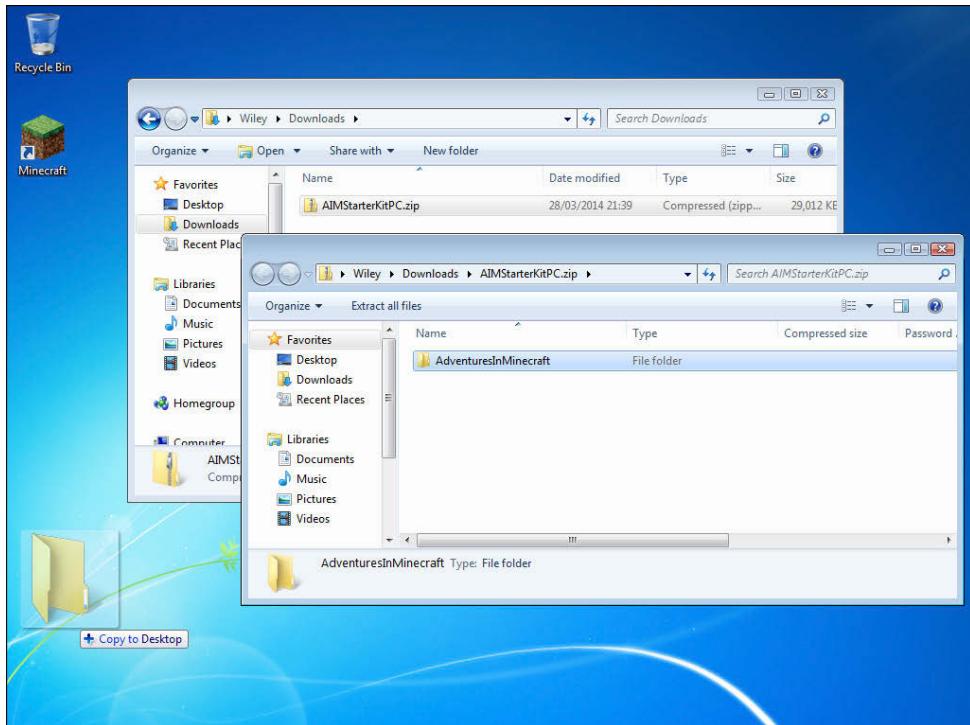


FIGURE 1-4 Copy the **AdventuresInMinecraft** folder to the PC desktop where you will always be able to find it.

Downloading and Installing Python

Because you will be doing your coding by using Python, you now have to install the Python programming language and the code editor IDLE by following these steps:

If you don't have an administrator account for your computer, you will need someone with an administrator account to enter their password before you can install Python.



1. Open your PC's web browser (such as Internet Explorer, Chrome). Go to www.python.org/download/releases/2.7.6 and click the Windows x86 MSI Installer (2.7.6) link to install it.
2. When the file `python-2.7.6.msi` has finished downloading, run it either by clicking the Open>Run menu option or by opening the download folder and double-clicking the file.
3. You may be presented with a security warning box asking "Do you want to run this file?" Click Run.
4. Click Next to start the installation.
5. You will be asked to choose a location to install Python. It is best to choose the default location. Click Next. Make a note of the location so you can find your Python files if you need them.
6. Don't change any of the options on Customize Python window, just click Next.
7. User Account Control may ask for your permission to run the set-up program. If so, click Yes.
8. Wait for setup program to complete the installation and click Finish.

Installing the Starter Kit and Python on Your Apple Mac

If you are using an Apple Mac, the next part in setting up your computer is to:

1. Download the starter kit for Apple Mac and extract it to your desktop.
2. Download the Python programming language and install it on your Apple Mac.

Downloading and Extracting the Starter Kit

Follows these steps to download the Apple Mac starter kit and copy the contents to the desktop (by placing them on your desktop it will make it easy for you to find them whenever you need them):

1. Open your Apple Mac's web browser (such as Safari, Chrome), go to the companion website (www.wiley.com/go/adventuresinminecraft) and download the starter kit for your Apple Mac.
2. When the starter kit zip file has finished downloading, open it. You can do this either by clicking the file or by opening the download folder and double-clicking the `AIMStarterKitMac.zip` file.
3. The zip file contains only one folder, called `AdventuresInMinecraft`. Copy this folder to the desktop by clicking the folder, holding down the button and dragging it to the desktop (see Figure 1-5).

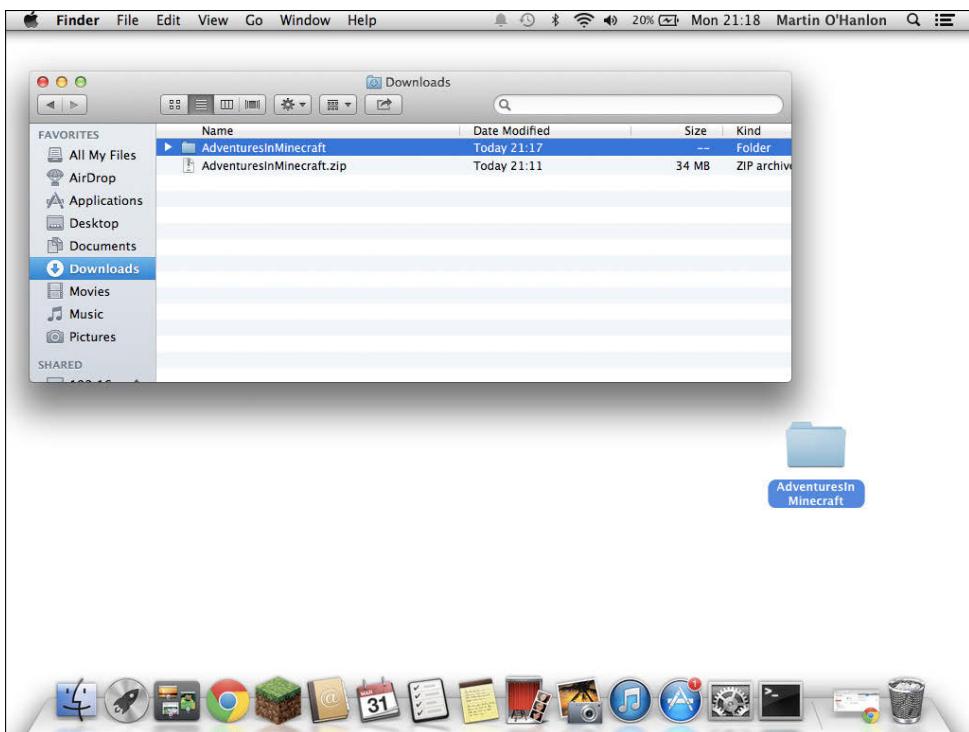


FIGURE 1-5 Copy the **AdventuresInMinecraft** folder to the Mac desktop.

Downloading and Installing Python

Because you will be doing your coding by using Python, you now have to install the Python programming language and the code editor IDLE by following these steps:

Depending on your computer set-up, you may need to enter your Apple password or get someone with an administrator account to enter their password before you can install Python.



1. Open your Apple Mac's web browser. Go to www.python.org/download/releases/2.7.6 and click the link to download the Mac OS X 64-bit/32-bit x86-64/i386 Installer (2.7.6) for Mac OS X 10.6 and later.
2. When the file **python-2.7.6-macosx10.6.dmg** has finished downloading, open the file by clicking on it or opening the download folder in Finder and double-clicking it.
3. Find the file called **Python.mpkg**. This is the python installation program. Right-click the file, then click Open With>Installer to start the Python install.



On OS X 10.8+ you may see a message that Python can't be installed because it is from an "unidentified developer". Don't be alarmed if this happens! It simply means that the Python installer is not compatible with the Gatekeeper security feature introduced in OS X 10.8. But it's perfectly okay for you to install it. It won't create any issues with your computer. Just click Open.

4. The Introduction screen will display information about the Python installer. Click Continue.
5. The Read Me screen displays important information about Python. Click Continue.
6. The License screen displays the Software License Agreement. Click Continue and then click Agree.
7. The Installation Type screen displays how much disk space will be used, click Install. You will also have the opportunity to change the install location if required.
8. Enter your Apple password and click Install Software and the installation will start, showing you progress. Wait for Python to finish installing.
9. Click Close when the installation program says the installation was successful.

Starting Minecraft on Your Windows PC or Apple Mac



In future adventures, the instructions tell you to start Minecraft. Just refer back to this section if you need a reminder of how to start Bukkit and connect Minecraft to it on your PC or Apple Mac.

You've now installed all the software you need on your Windows PC or Apple Mac. But you're not quite ready yet; first, you need to start up the Bukkit server and connect Minecraft to it by following these steps:

1. Start by opening the **AdventuresInMinecraft** folder you placed on the desktop by double-clicking it.
2. Run the StartBukkit program by double-clicking it. This will open the Bukkit command window.

If you're using an Apple Mac and depending on the set up of your computer, you may receive a message saying "StartBukkit.command" can't be opened because it is from an unidentified developer. If so, right-click the StartBukkit program and choose Open with Terminal.



3. Press any key to start Bukkit. As Bukkit is loading messages will appear on the screen to keep you updated on its progress.

The first time it runs Bukkit, your computer may display a message asking for your permission to run the program, or to allow Bukkit to access the network. If this happens, just click to agree to give permission/access.



4. When Bukkit has finished loading, the message "Done" appears (see Figure 1-6).

```
21:48:43 [INFO] Loading properties
21:48:43 [INFO] Default game type: CREATIVE
21:48:43 [INFO] Generating keypair
21:48:44 [INFO] Starting Minecraft server on *:25565
21:48:44 [INFO] This server is running CraftBukkit version git-Bukkit-1.6.4-R2.0-
-62918-jnks (MC: 1.6.4) (Implementing API version 1.6.4-R2.0)
21:48:45 [INFO] [Raspberrijuice] Loading Raspberrijuice v1.3
21:48:45 [WARNING] *** SERVER IS RUNNING IN OFFLINE/INSECURE MODE!
21:48:45 [WARNING] The server will make no attempt to authenticate usernames. Be
careful.
21:48:45 [WARNING] While this makes the game possible to play without internet ac-
cess, it also opens up the ability for hackers to connect with any username they
choose.
21:48:45 [WARNING] To change this, set "online-mode" to "true" in the server.pro-
perties file.
21:48:45 [INFO] Preparing level "world"
21:48:46 [INFO] Preparing start region for level 0 <Seed: -3789787156735749996>
21:48:46 [INFO] Preparing spawn area: 21x
21:48:47 [INFO] Preparing start region for level 1 <Seed: -3789787156735749996>
21:48:48 [INFO] Preparing spawn area: 50x
21:48:49 [INFO] Preparing start region for level 2 <Seed: -3789787156735749996>
21:48:50 [INFO] [Raspberrijuice] Enabling RaspberriJuice v1.3
21:48:50 [INFO] Server permissions file permissions.yml is empty, ignoring it
21:48:50 [INFO] Done [5.294s] For help, type "help" or "?"
```

FIGURE 1-6 The Bukkit command window displays "Done" when it is loaded and ready to use.

For all the adventures in this book, the version of Minecraft and Bukkit used is 1.6.4. That means you need to change the Minecraft launcher so it runs version 1.6.4 of the game. By using this version of the game, rather than the latest version, the code and programs included in this book are guaranteed to work. You will only need to do this once, as Minecraft will remember the version you are using.



If you wanted to use a later version of Minecraft, the **README** file within the starter kit describes how to create your own Minecraft starter kit. This should only be attempted by advanced users who have a good understanding of how to configure their computer, have experience of editing configuration files and running Java programs.

To configure your Minecraft launcher to use version 1.6.4, follow these steps:

1. Start Minecraft.
2. When the Minecraft Launcher window is displayed, click Edit Profile in the bottom left of the screen, next to your username.
3. From the Use Version drop-down list, select release 1.6.4, as shown in Figure 1-7.
4. Click Save Profile.

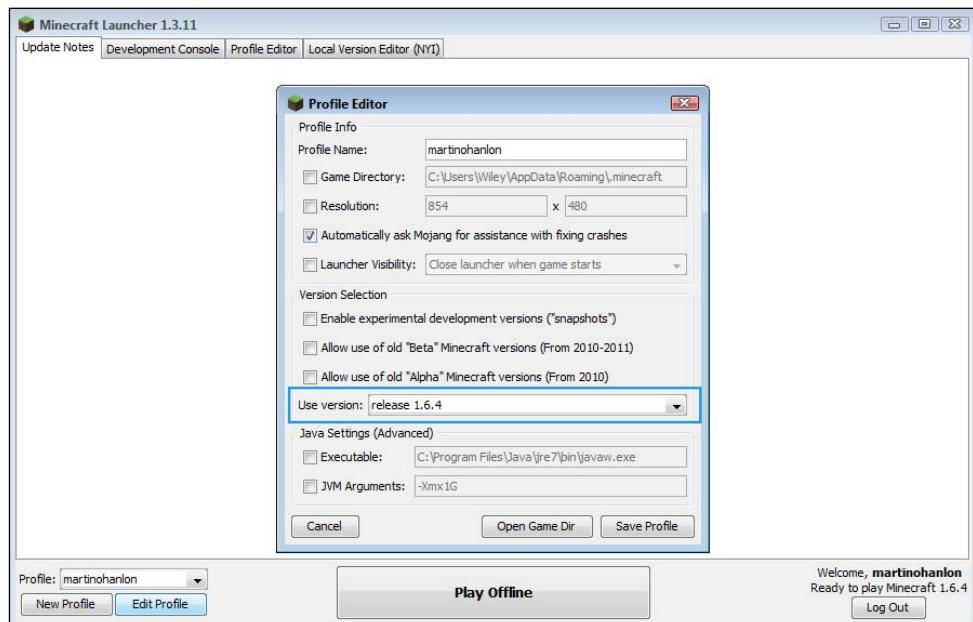


FIGURE 1-7 Choose which version of Minecraft to use.

Now connect Minecraft to the Bukkit server:

1. Click Play on the Minecraft Launcher to start the game.
2. From the Minecraft menu, click Multiplayer.

If you are using a Windows PC, a Windows Security Alert message may pop up asking you to allow Minecraft to access the network. Click Allow Access.



3. From the Play Multiplayer menu, click Direct Connect.
4. Enter `localhost` in the Server Address and click Join Server (see Figure 1-8). That's it! You should now enter the Minecraft world that has been created on your Bukkit server. You've arrived!

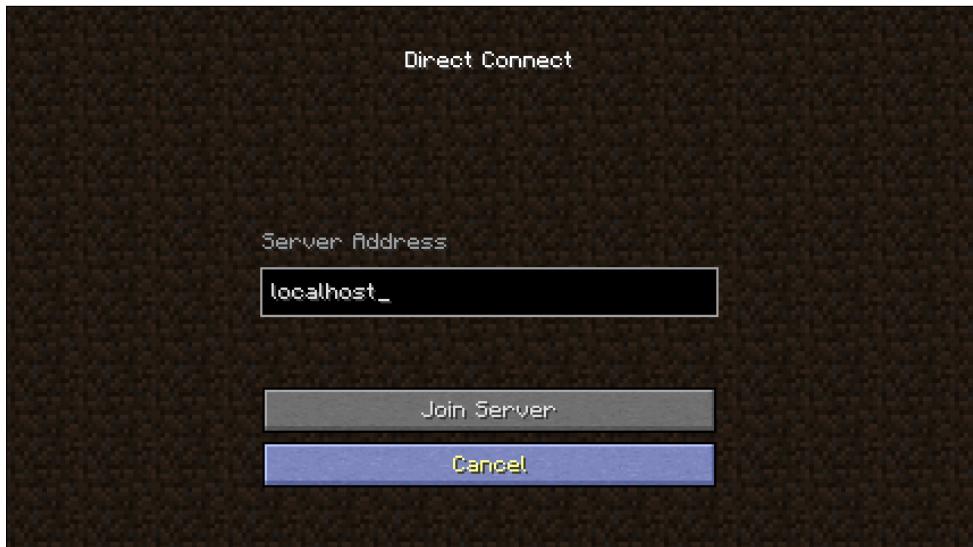


FIGURE 1-8 Connect Minecraft to the Bukkit server.

Stopping Bukkit

When you have finished your adventures for the day, you need to disconnect from the Bukkit server. Do this by pressing Escape in Minecraft to go back to the menu, then clicking Disconnect. You can now shut down Bukkit safely by typing `stop` and pressing Enter in the Bukkit command window.

On the screen, you will see a series of updates from Bukkit, telling you what is happening as it shuts down. After it tells you it is “Closing listening threads”, you may see messages saying “SEVERE”. Don’t worry! This is normal and won’t affect your computer or Minecraft programs.



You can type into the Bukkit command window to control many aspects of the Minecraft game, such as the time of day or the weather. For example, if it gets dark in your Minecraft world and you're not ready for night to fall just yet, type `set time 1` in the command window to reset the time back to morning. If it starts to rain, clear the skies by typing `weather clear`. You can find a complete list of Bukkit commands by typing `help`.

Creating a Program

Congratulations! You've set up your computer and Minecraft is up and running on it. You may have found the set-up process a little tedious, but it's done now, and you will have to do it again only if you want to use a different computer. Now it's time for the interesting stuff—creating your first program, "Hello Minecraft World".



In future adventures, the instructions will tell you to start IDLE. You can refer back to this section if you ever need a reminder of how to start IDLE on your computer.

First, you need to start the Python and open IDLE by doing the following:

- On a Raspberry Pi: Double-click the IDLE (not IDLE3) icon on the desktop.
- On a PC: Click Start button ➔ All Programs ➔ Python 2.7 ➔ IDLE (Python GUI).
- On a PC (Windows 8): Click Start button, then either click the IDLE (Python GUI) tile on the home screen, or use Search to find IDLE.
- On a Mac: Click Go on the Finder menu bar ➔ Applications ➔ Python 2.7. Double-click IDLE.

The Python Shell window will now appear.



If you are using a Raspberry Pi, the Python Shell window can take a few seconds to appear after you have double-clicked the icon.

The first time IDLE is run, your computer may display a message asking for your permission to run the program, or to allow IDLE to access the network. This is fine; just agree to it.



Once the Python Shell window has loaded, you can create new programs in IDLE. The program you are going to create here won't do anything fancy. Just as computer programmers always start by writing a "hello world" program, you're now going to create a Hello Minecraft World program to check that everything is properly installed on your computer. Here's how:

Future adventures will tell you to create a new program and save program to [MyAdventures](#). You can refer back to this section if you ever need a reminder of how to create a new program.



1. Create a new file by clicking **File**→**New File** on the IDLE menu. (Note that on a Raspberry Pi, "New File" may be called "New Window".)
2. Save the file to the [MyAdventures](#) folder by clicking **File**→**Save** on the IDLE editor menu.
3. Select the [MyAdventure](#) folder:
 - On a Raspberry Pi: Select `/home/pi` in the Directory drop-down list and double-click [MyAdventures](#) on the folder browser.
 - On a PC: Click Desktop on the navigation pane on the left, double-click [AdventuresInMinecraft](#) on the folder explorer and then double-click [MyAdventures](#).
 - On a Mac: Click Desktop on the folder browser, click [Adventures InMinecraft](#) and then click [MyAdventures](#).
4. Next you need to give your new file a name. Type in the filename [HelloMinecraftWorld.py](#) and click Save. The `.py` added to the end of the file tells your computer that the file is a Python program.



It's important that you save the program in the **MyAdventures** folder. This is where you will save all your *Adventures in Minecraft* programs, because it contains everything you need to get your programs running.

- Now it's time to start programming! Type the following code into the IDLE editor window to start the Hello Minecraft World program. Make sure you get the upper and lower case letters correct, as Python is **case-sensitive**:

```
import mcpi.minecraft as minecraft  
mc = minecraft.Minecraft.create()  
mc.postToChat ("Hello Minecraft World")
```



Python is a **case-sensitive** programming language, which means that you must enter characters in upper or lower case correctly. For example, Python will treat Minecraft (upper case) differently to minecraft (lower case). If you enter them incorrectly it will result in errors, and you will have to retrace your steps to see where you went wrong.



You will learn more about what this code means and does in the next adventure. For now, you're keeping it simple and just get the words "Hello Minecraft World" on the screen to prove that everything works.

- Save your program by choosing File ➔ Save from the IDLE editor menu.

Running a Program



In future adventures, the instructions will tell you to run a program. If you ever need a reminder of how to run a Python program, just refer back to this section.

You have now created a program! Now it's time to test it. To do this, you need to tell IDLE to run the program by following these steps:

1. Before it can run the Hello Minecraft World program, Minecraft needs to be open and in a game. If it isn't, start up Minecraft by following the steps to start Minecraft as you learned to do earlier.
2. Resize the Minecraft window, so you can see both the Minecraft and IDLE windows. (If you are using a PC or Mac and Minecraft is shown on a full screen, press F11 to exit full screen mode so you can see Minecraft in a window.) You should have the following windows open:
 - On a Pi: You should have three windows open: Python Shell; IDLE code editor with your `HelloMinecraftWorld.py` program; and Minecraft (see Figure 1-9).

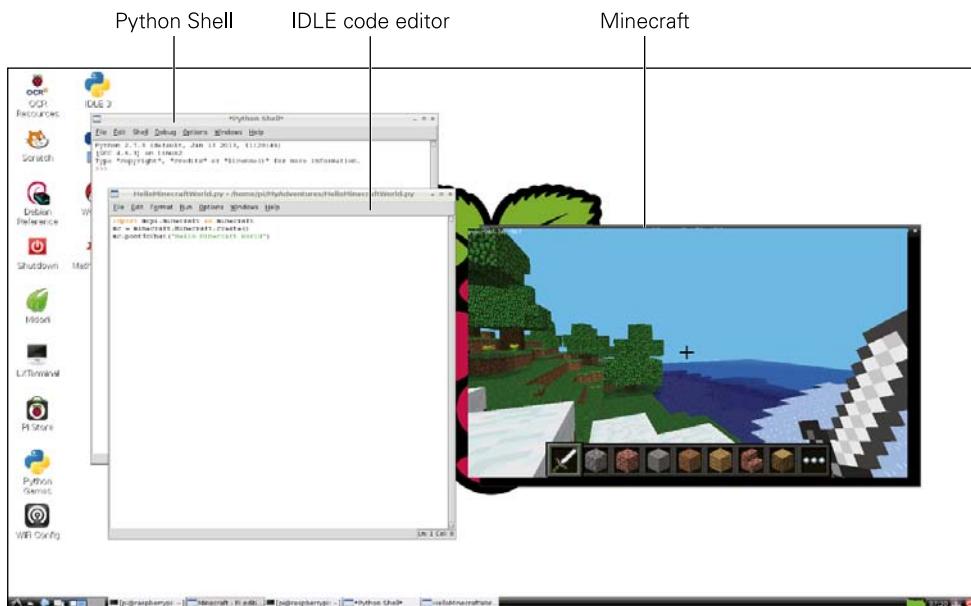


FIGURE 1-9 Raspberry Pi is ready to run your program.

- On a PC: You should have four windows open: Python Shell; IDLE code editor with your `HelloMinecraftWorld.py` program; Bukkit command window; and Minecraft (see Figure 1-10).

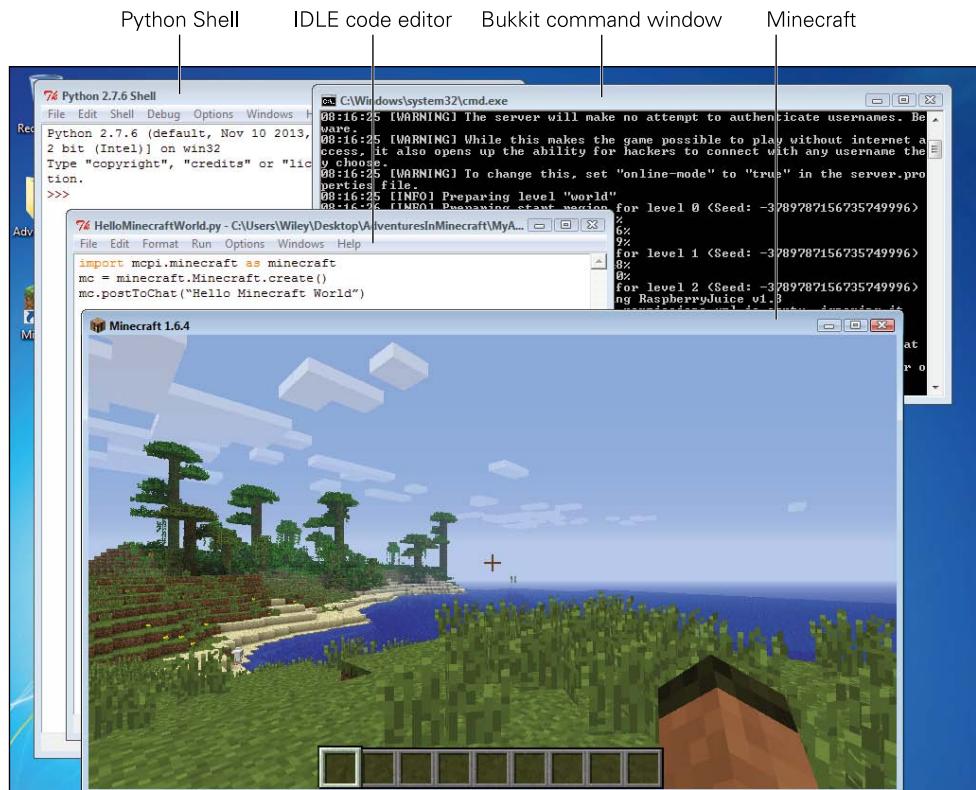


FIGURE 1-10 Windows PC is ready to run your program.

- On a Mac: You should have four windows open: Python Shell; IDLE code editor with your `HelloMinecraftWorld.py` program; Bukkit command window; and Minecraft (see Figure 1-11).
3. Press Esc to open the Minecraft menu. Use your mouse pointer to select the IDLE code editor.
 4. Run the `HelloMinecraftWorld.py` program by clicking Run → Run Module on the IDLE menu, or pressing F5.
 5. IDLE will automatically switch to the Python Shell window and run the program. If there are any errors, you will see them displayed here in red. If there are any errors, carefully check the code you typed in earlier by switching back to the IDLE code editor and comparing it to the code above to see where you went wrong.
 6. Bring the window with the Minecraft game to the front. Notice anything? All your hard work has produced results and “Hello Minecraft World” is now displayed in the chat. (It should look something like Figure 1-1.)

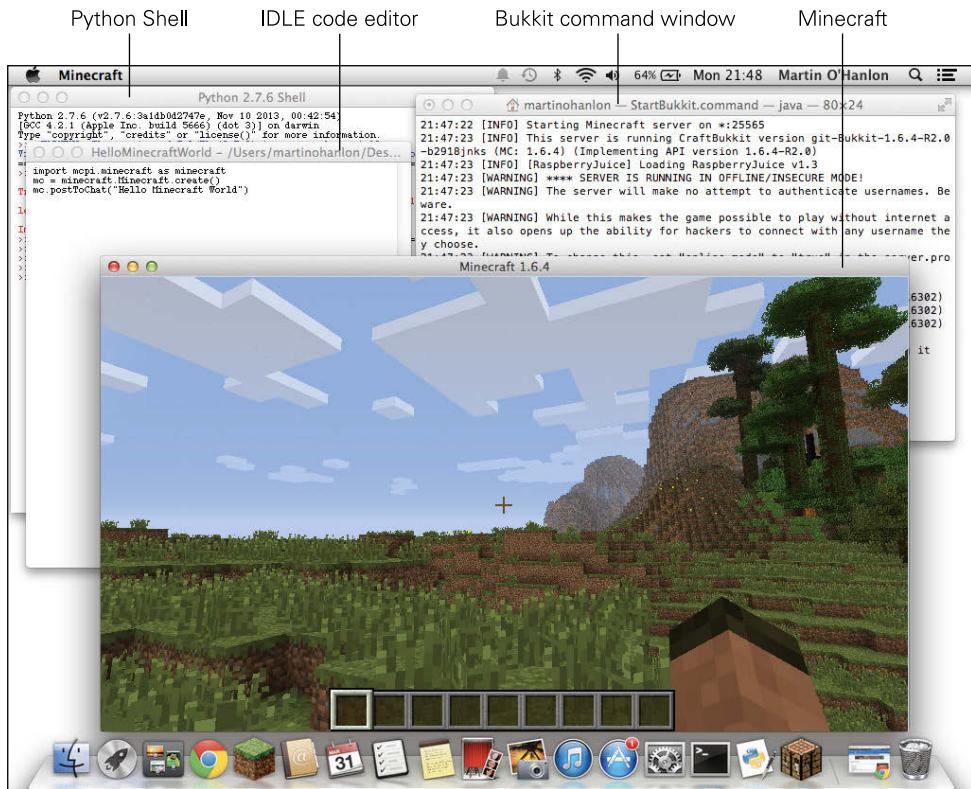


FIGURE 1-11 Apple Mac is ready to run your program.

If you have set everything up correctly, “Hello Minecraft World” will be displayed in Minecraft chat (as in Figure 1-1); but if it isn’t, or errors are displayed in the Python Shell, go back over the instructions in this adventure. It’s really important that you get your initial set-up right; if you don’t, the programs you create in future adventures won’t work.



Stopping a Program

In future adventures, the instructions will tell you to stop the program. If you ever need a reminder of how to stop a Python program, just refer back to this section.



The Hello Minecraft World program runs, displays the message on the screen and then stops. It is however important to know how to force a program to stop running, because, in future adventures, the programs you create will not stop until you tell them to stop! To stop a program, select the Python Shell window then click Shell ➔ Restart Shell from the menu, or hold down Ctrl and press C.



Achievement Unlocked: The hard part is over! You've got Minecraft up and running, you've written your first program and the world says "Hello".

In the Next Adventure...

In Adventure 2, you will learn some simple programming skills with Minecraft and Python that will allow you to track the position of your player. You'll learn about the "game within a game" concept, and you'll write some simple games that change the way they behave depending on where your player is inside the Minecraft world.



Adventure 2

Tracking Your Players as They Move

WHEN YOU PLAY Minecraft, you are playing a game that other people have designed for you. The Minecraft world is fun, but it's even more fun if you can make it do the things that you want it to do. When writing Minecraft programs in Python, you now have a complete programming environment at your fingertips, and you can invent and program anything you can possibly imagine. This adventure introduces you to some of the fun things you can create using Minecraft and the programming language Python.

A really fun thing to do when programming Minecraft is to create a game within a game. Minecraft is the “world”, and your **program** will make this world behave in new ways. You will find that most Minecraft programs have three things that make them fun and exciting: *sensing* something about the world, such as the player position; *calculating* something new, such as a score; and *behaving* in some way, such as moving your player to a new location.

In this adventure you will learn how to sense your player’s position and make different things happen in the game as your player moves. In the welcome home game you build a magic doormat that greets you when you walk on it. We’ll introduce a technique called *geo-fencing* using a real Minecraft fence, and your game will challenge you and your friends to compete to collect objects in the fastest time possible. Finally you will learn how to move your player by making your game catapult your player into the sky if you don’t get out of that field quick enough!



A **program** is a series of instructions or statements that you type in a particular programming language, which the computer then follows automatically. You must use the correct type of instructions for the programming language you are using. In this book, you are using the Python programming language.



A **statement** is a general computing term that usually means one complete instruction that you give to the computer, e.g. one line of a computer program such as `print ("Hello Steve")`. Some people also use the word *command*, but that is more relevant when you are talking about commands that you type at a command prompt to a computer.

Python has its own very precise explanation of what a statement is, but for the purposes of this book I will use *statement* to mean one individual instruction or line in the Python program. If you want to read more about the Python language, you can read the online documentation here: <https://docs.python.org/2/>

Sensing Your Player's Position

Before you can sense your player's position, you need to understand a bit about how the Minecraft world is organised. Everything in Minecraft is exactly one block in size, and a block represents a one-metre cube. As your player, Steve, moves around the Minecraft world, the Minecraft game remembers where he is by a set of **coordinates**. The world is constructed of empty blocks that are filled by materials that are usually one-metre cube in volume. There are a number of exceptions to this, for example carpets and liquids such as water and lava that can take up less volume when viewed, but cannot be combined with other materials to fill less space than a one-metre cube.

By sensing your player's position in the Minecraft world, you can make your game intelligently react as you move around the world, such as displaying messages at specific locations, automatically building structures as you move around, and changing your players position when you walk to specific locations. You will learn how to do all of these things in this book, and many of them you will learn right here in this adventure!



A **coordinate** is a set of numbers that uniquely represents a position. In Minecraft, 3D coordinates are used to represent the exact position within the three-dimensional Minecraft world, and each coordinate consists of three numbers. See also http://en.wikipedia.org/wiki/Coordinate_system for information about coordinates, and <http://minecraft.gamepedia.com/Coordinates> for information about how Minecraft uses coordinates in the Minecraft world.

These coordinates are called x, y and z. You have to be careful when you talk about coordinates in Minecraft; if you use words such as “left” and “right”, whether something is left or right depends on which way you are facing in the Minecraft world. It’s better to think of Minecraft coordinates as relating to the directions on a compass. Figure 2-1 shows how changes in x, y and z coordinates relate to movement in the Minecraft world.

It's easier to understand how coordinates change as you move your player in Minecraft by experimenting inside the game. On the Raspberry Pi, as you move your player around in the world the x, y and z coordinates are displayed in the top left corner of the screen. On the PC/Mac you can display the coordinates by pressing the F3 key. You can learn about all the advanced controls of Minecraft at <http://minecraft.gamepedia.com/Controls>.



- x gets bigger as your player heads east, and smaller as you head west.
- y gets bigger up in the sky, and smaller down into the ground.
- z gets bigger as your player heads south, and smaller as you head north.

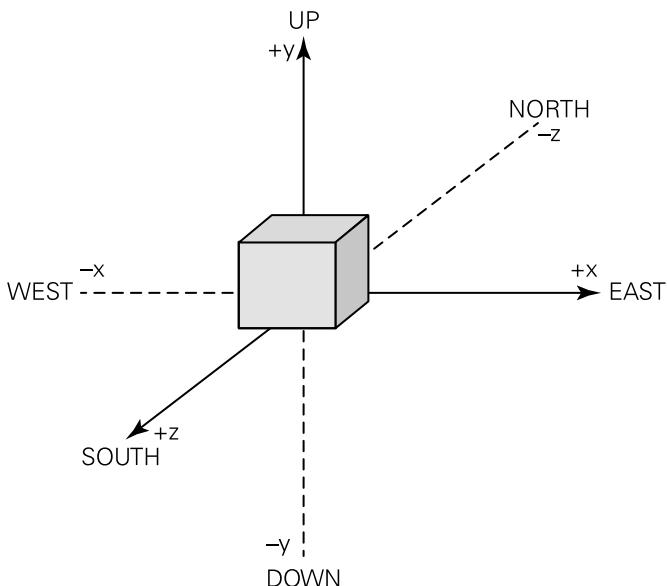


FIGURE 2-1 The x, y and z coordinates in Minecraft relate to headings on a compass like this.

Getting Started

Now that you've completed the Hello Minecraft World program in Adventure 1, it's time for you to write your own program to link up with Minecraft!

Adventure 1 explained in detail how you get everything running and you'll perfect those steps as you work your way through your adventures. All of the steps you need to know to start everything up and begin typing programs are described in Adventure 1, but we'll give you some reminders in the first few adventures while you're still learning how to do it.



For a reminder on how to get Minecraft running properly on your particular computer, visit the companion website at www.wiley.com/go/adventures inminecraft and select the Adventure 1 video.

1. Start up Minecraft, IDLE, and if you are working on a PC or a Mac, start up the server too. You should have had a bit of practice with starting everything up by now, but refer back to Adventure 1 if you need any reminders.
2. Open IDLE, the Python Integrated Development Environment (IDE), just as you did in Adventure 1. This will be your window into the Python programming language, and it's where you will type and run all your Python programs.

Showing Your Player's Position

The best way to understand how coordinates work is to write a little experimental program, so that's what you're going to do now. This program will show the 3D coordinates of your player, and you'll be able to see the coordinates changing as your player moves.

1. From the Python Shell, create a new program by choosing File→New File. A new untitled window will open, which is where you will type your new program.
2. Before you start to type a program, it is good practice to save the file first, so that if you have to rush away from your computer you don't have to think of a name for the program under pressure. You just have to choose to save it. Professional programmers do this all the time to make sure they don't lose their work. From the Python Shell menu, choose File→Save As, and type the name `whereAmI.py`. Make sure that you save your program inside the `MyAdventures` folder.
3. Your program will be communicating directly with the Minecraft game through the Minecraft **API**. To gain access to this API you will import the `minecraft` module, which gives your Python program access to all the facilities of the Minecraft game. Use the following to import the module:

```
import mcpi.minecraft as minecraft
```

- To communicate with a running Minecraft game, you need a connection to that game. Type the following to connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

Be aware that Python is a case-sensitive language—so be careful how you capitalise words! It's important that you type upper case and lower case letters correctly. The second Minecraft in the statement you just typed in must start with a capital letter for it to work properly.



- Next, ask the Minecraft game for the position of your player by using `getTilePos()`:

```
pos = mc.player.getTilePos()
```

- Finally, tell the Minecraft game to display the coordinates of the player's position. `print()` will display these coordinates on the Python Shell when your program runs:

```
print(pos.x)
print(pos.y)
print(pos.z)
```

- Save this file by choosing File→Save, from the Editor menu.
- Run the program by choosing Run→Run Module, from the Editor menu.

You should now see the coordinates of the player's location displayed on the Python Shell (see Figure 2-2). Later in this adventure, you'll use these coordinates to sense where your player is standing, and build a magic doormat that welcomes your player home when he stands on it!

A screenshot of the Python 2.7.6 Shell window. The title bar says "7k Python 2.7.6 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, Help. The main window shows the Python interpreter prompt: >>>. Below the prompt, the output of the code execution is shown, including the coordinates (1, 4, 7) and the word "RESTART". At the bottom right of the window, there is a status bar with "Ln: 8 Col: 4".

FIGURE 2-2 Use `getTilePos` to show the player position on the Python Shell.



API stands for application programming interface. An API gives you a way to safely access parts of an application program from within your programs. It is the Minecraft API that gives you access to the Minecraft game from within your Python programs. The Minecraft game must be running before you can connect your Python programs to it via this API. The `mcpi` in the line `import mcpi.minecraft as minecraft` stands for *Minecraft Pi*, because the very first version of the API only ran on the Raspberry Pi.

An **interface** is a set of rules that explain how, as a programmer, you can access some other part of a computer system. The API is a set of rules explaining how you can communicate with a running program—in this case, the Minecraft game. All your Minecraft programs will access the running Minecraft game through the Minecraft API.



All of the Python programs you create will need to access the Minecraft API. For this to work properly, your programs must be able to access the `mcpi.minecraft` module, which is stored in the `mcpi` folder of your `MyAdventures` folder. The easiest way to make sure that this always works is always to create your Python programs inside your `MyAdventures` folder.

DIGGING INTO THE CODE

Congratulations—you have just used a variable in your program! `pos.x` is a variable, and so are `pos.y` and `pos.z`.

A **variable** is just a name for a location in the computer's memory. Whenever you put a variable name to the left of an equals sign, it stores a value in that variable, for example:

```
a = 10
```

Whenever you use a variable inside a **print statement**, the program will display the value of that variable on the Python Shell. So, if `a = 10`, the following `print(a)`

will print out 10, the value of the variable `a`.

The `pos` variable that you use from `pos = mc.player.getTilePos()` is a special type of variable. Just think of it as a box with three sub-compartments labelled `x`, `y` and `z`:

```
print(pos.x)
```

Try moving your player to a new location and running the program again. Check the coordinates displayed on the Python Shell: these should now have changed to reflect your player's new position. You can use this little program at any time if you want to confirm the coordinates of a position in the Minecraft world.

A **variable** is a name for a location in the computer's memory. You can store new values in variables at any time in your program, and you can read back the values and display them or use them in calculations. Think of a variable as being a bit like the memory button on a calculator.



Tidying Up Your Position Display

Before you move on and do more with your Minecraft programs, you can improve the output of your player position to make it easier to understand. Having three separate lines of output every time you display the position takes up a lot of space; it would be nice to have all three parts of the player position on a single line, like this:

```
x=10 y=2 z=20
```

To do that, follow these steps:

1. Modify your `whereAmI.py` program by removing the three `print()` statements and replacing them with this single line:

```
print ("x="+str(pos.x) + " y="+str(pos.y) + " z="+str(pos.z))
```

2. Save your program again by selecting `File`→`Save`, from the Editor menu, and then run it by clicking `Run`→`Run Module`. See how the output has improved—now it is all on one line!

DIGGING INTO THE CODE

In your new version of the `whereAmI.py` program, a little bit of extra magic took place, which I need to explain.

In Python, `print()` will display anything you put in the brackets, and the output will be displayed on the Python Shell. You have already used different ways of displaying information with `print()`. Now you can examine them a little further. Look at this example:

```
print("hello")
```

continued

continued

Just like in your Hello Minecraft World adventure, `print()` displays the word *hello*. The quotes around the word tell Python that you want to display the word *hello* exactly as it is given. If you put spaces between the quotes, those spaces will be displayed too. The text between the quotes is called a **string**.

Here is another type of `print()`:

```
print(pos.x)
```

`print()` displays the value inside the `pos.x` variable. This is a number, but the number might change every time you use the `print()` statement depending on what value is stored in that variable.

Finally, you need to understand what `str()` does in your `print()` statement. Python `print()` gives you many different ways to mix numbers and strings together. When you mix numbers and strings inside Python, as in the following code, you have to tell Python to convert the numbers to strings so that they can be tacked on to the rest of the characters in the string that is being displayed. `str()` is built in to Python and it converts whatever variable or value is between its brackets into a string. The plus symbol (+) tells Python to join together "x= ." and whatever is stored in the `pos.x` variable to make one big string, which is then printed onto the Python Shell window.

```
print("x= "+str(pos.x))
```

Why don't you try this line without the `str()` to see what happens:

```
print("x= "+ pos.x)
```



A **string** is a type of variable that can store a sequence of letters, numbers, and symbols. It is called a string because you can imagine a long piece of string with beads that you slide on it like a necklace or a wrist band. Each bead could have a different number, letter, or symbol printed on it. The string then keeps them all in the same order for you.

You can use strings for many different purposes, such as storing your name, or an address, or a message to display on the Minecraft chat.

Using `postToChat` to Change Where Your Position Displays

It's unusual, but in the `whereAmI.py` program, you play the game in the Minecraft window but your position appears on the Python Shell window. You can easily fix that by using a technique you have used already, in Adventure 1: `postToChat`! Here's how:

1. Change the `print()` statement to `mc.postToChat()`, like this:

```
print ("x="+str(pos.x) +" y="+str(pos.y) +" z="+str(pos.z))  
mc.postToChat ("x="+str(pos.x) +" y="+str(pos.y) +  
" z="+str(pos.z))
```

2. Click File→Save to save your program, and run it again by choosing Run→Run Module from the editor menu.

Well done! Your player position should now be displayed on the Minecraft chat, which is much more convenient for you as you play the game.

Introducing a Game Loop

Having to run your `whereAmI.py` program every time you want to find out the player position is not very helpful. Luckily, you can get round this by modifying your program to add a game loop. Almost every other program you write in this book will need a game loop to make sure that the program continues to run and interact with your Minecraft gaming. Most games you play will continue to run forever until you close them down, and this is a useful technique to learn. In computing, this is called an **infinite loop**, because it loops forever.

An **infinite loop** is a loop that never ends—it goes on and on to infinity. The only way to break out of an infinite loop is to stop the Python program. You can stop the program in IDLE by choosing Shell→Restart Shell, from the Python Shell menu or by pressing CTRL then C on the keyboard at the Python Shell.



You can add the game loop by modifying your existing program; the new program will be very similar, so you will save a bit of time this way. Be sure to save the new file under a different file name so you don't lose your first program.

1. Start by saving your program as a new file by choosing File→Save As from the editor menu and calling it `whereAmI2.py`. Make sure you save it in your `MyAdventures` folder; otherwise it will not work.

- Now you need to import a new module that allows you to insert time delays. You do this because if you don't slow down the loop, you will be bombarded with a flood of messages on the chat and won't be able to see what you are doing. Add the following **bold** line to your existing program:

```
import mcpi.minecraft as minecraft  
import time
```

- Next, modify the main part of your program by adding the lines that are marked in **bold** in the following, noting that you have to indent the lines beneath **while True** to tell Python which instructions it needs to repeat. See the Digging into the Code section to understand why this indent is needed:

```
while True:  
    time.sleep(1)  
    pos = mc.player.getTilePos()  
    mc.postToChat("x="+str(pos.x) + " y="+str(pos.y) +  
    " z="+str(pos.z))
```

- Choose File→Save from the editor menu to save your changes.
- Now you can run your program by choosing Run→Run Module from the editor menu. Have a walk around the Minecraft world and notice that, once every second, you will see the coordinates of your player posted to the chat.

DIGGING INTO THE CODE

Indentation is extremely important in all programming languages, as it shows how the program is structured and which groups of statements belong with other statements. Indentation is even more important in the Python programming language, because it is used by Python to understand the meaning of your program (unlike other languages such as C that use {} braces to group statements together). Indentation is important when you use **while** loops and other statements, as it shows which of these belong to the loop and will therefore be repeated over and over again. In your game loop, all the program statements underneath **while True:** are indented, because they belong to the loop and will be repeated each time round the loop.

In this example, you can see that the "hello" is printed once but the word "tick" is then printed once every second. **time.sleep()** and **print()** belong to the loop because they are indented:

```
print("hello")  
while True: # this is the start of the loop  
    time.sleep(1)  
    print("tick")
```

In Python, indentation is even more important than in other programming languages, because the amount of indent (the indent level) tells the Python language exactly which statements belong to loops. If you get the indentation wrong, the whole meaning of the program changes.

Indentation is the space at the left hand edge of each line of a program. Indents are used to show the structure of the program and to group together program statements under loops and other statements. In Python, the indents are important as they also change the meaning of the program.



It is important that you get the indentation right. Python can work with both spaces and tabs when you indent code but if you mix tabs and spaces inside the same program, Python can easily get confused about what you mean, so it is best always to use a consistent method for indenting your programs. Most people like to press the Tab key for each level of indent, as it is quicker than typing in lots of spaces, but if you start using Tab you should always use it and not mix it with spaces.



Now that your program is running in an infinite game loop, it will never stop! Before you run your next program, stop this program from running by choosing Shell→Restart Shell, from the Python Shell menu, or pressing CTRL+C on the keyboard.



Building the Welcome Home Game

It's time for you to put your game loop into practice and develop a simple application. The Welcome Home game will use sensing (tracking your player's position) to follow your player as he moves around the Minecraft world. You will also learn how to do more complex sensing with the `if` statement in Python. You will build a magic door mat, which will make the message "welcome home" pop up on the Minecraft chat when you stand on it.

VIDEO

To see a tutorial about how to write and play the Welcome Home game, visit the companion website at www.wiley.com/go/adventuresinminecraft and choose the Adventure 2 video.

Using if Statements to Make a Magic Doormat

To work out whether your player is standing on the doormat, you will use an `if` statement to compare your player's position against the position of the doormat. When the two match, your player will have arrived home.

First, look at how the `if` statement works; trying it out yourself on the Python Shell will help you to understand it:

1. Click on the Python Shell. Make sure you click just to the right of the `>>>` prompt to make sure that what you type goes in at the right place.
2. Type the following line into the Python Shell, and Python will run it as soon as you press the Enter key. It won't display anything yet, as all this is doing is storing the number 20 inside the variable called `a`:

```
a = 20
```

3. Check that the variable `a` has the correct value in it by typing the following into the Python Shell:

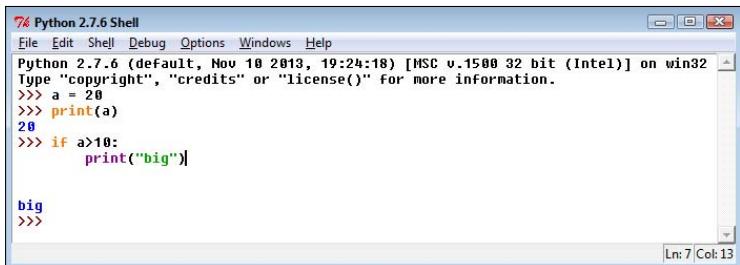
```
print(a)
```

The number 20 should display on the screen.

4. Try out an `if` statement to see if the value stored in variable `a` is bigger than 10. Note that you have to put the colon at the end of the `if` statement. When you press Enter at the end of the first line, Python will automatically indent the next line for you:

```
if a>10:  
    print("big")
```

5. Now press Enter again to tell the Python Shell that you have finished typing the `if` statement. You should see the word "big" appear on the Python Shell. To see what this looks like, see Figure 2-3.



The screenshot shows the Python 2.7.6 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main area displays the following code and its output:

```
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a = 20
>>> print(a)
20
>>> if a>10:
    print("big")

big
>>>
```

In the status bar at the bottom right, it says Ln: 7 Col: 13.

FIGURE 2-3 Using an `if` statement interactively in the Python Shell

The `if` statement has two outcomes, for example in the above code `if a>10` is either `True` (`a` is greater than `10`) or it is `False` (`a` is not greater than `10`).

`If` statements only ever have two outcomes, and you will meet the values `True` and `False` again a little later in this adventure.



Checking if Your Player Is at a Particular Location

For your program to be able to work out whether your player is standing on your magic doormat, it will have to check at least two parts of the coordinate. If you check that the `x` and `z` coordinates of the doormat are the same as your player's position, this will act as a reasonable method to detect that your player is standing on the mat. You can check this by using the keyword `and` inside the `if` statement to check one condition *and* another condition. The `y` coordinate is less important here, and because you don't check the `y` coordinate in the program, as your player hovers over the doormat you will still see the welcome home message.

Try this out interactively in the Python Shell first to make sure it works:

1. At the Python Shell `>>>` prompt, type the following to set a variable:

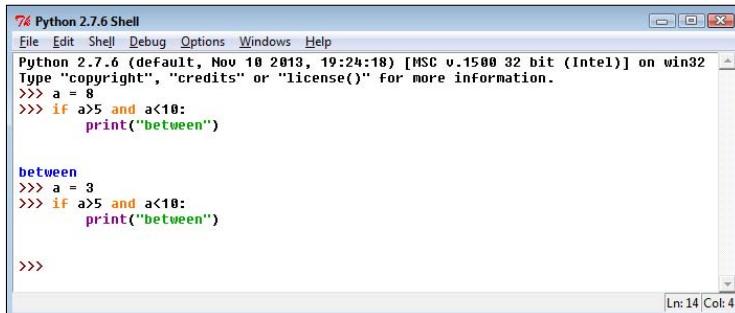
```
a = 8
```

2. Now you're going to type in an `if` statement that also uses an `and`. This statement will check if the `a` variable is between two numbers. The first time you type this you will see the word "between" appear on the Python Shell, because `8` is greater than `5` and less than `10`. Remember to press Enter a second time to finish the indented region of the `if`:

```
if a>5 and a<10:
    print("between")
```

- Now change `a` to be smaller than 5, and see what happens (see Figure 2-4):

```
a = 3
if a>5 and a<10:
    print("between")
```



```
74 Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a = 8
>>> if a>5 and a<10:
...     print("between")

between
>>> a = 3
>>> if a>5 and a<10:
...     print("between")

>>>
Ln: 14 Col: 4
```

FIGURE 2-4 You use an `if` statement with `and` to check two or more conditions.

CHALLENGE



Try the preceding example with different numbers, especially numbers close to 5 and 10. Which numbers cause the word “between” to be displayed?



When you are testing programs, it is very useful to test them with lots of different numbers. One way that you can reduce a possibly infinite amount of testing to something that is manageable is to look at the conditions of the `if` statements and only test around those numbers. In the previous example, I would test 4,5,6 and 9,10,11, and if those numbers work it would be safe to assume that all other possible numbers work in the program too.

Building a Magic Doormat

Before you can write your game, you first need to build something in the Minecraft world, to give your program something to interact with. All you really need is a doormat. But first, open Minecraft and load the world you’ve already been using.

- To place a doormat on the floor, choose an item from the inventory and right-click to place it on the floor in front of you. Wool is probably a good choice for a doormat.

- To find out the coordinates of your doormat in the Minecraft world, run your `whereAmI.py` program again, and then stand on the doormat. Write down the x, y and z coordinates somewhere, because you will need them when you come to write your new program so that you can locate the doormat in the Minecraft world.

You're probably already impatient to start building huge structures! In Adventure 3, you will learn how to do that automatically by writing Python programs but, for now, just build something simple, focusing on getting the program to work properly. You might like to build a simple house around your doormat, making sure that the doormat is in the open doorway of your house.



Writing the Welcome Home Game

Now that you understand how `if` statements work, you can write the Welcome Home Game by following these steps:

- Choose `File`→`New File` from the Python Shell menu to create a new program.
- Choose `File`→`Save As` from the editor menu to save your program, and call it `welcomeHome.py`. Remember that you need to save your program in your `My Adventures` folder for it to work properly.
- Import the modules you need for this program by typing:

```
import mcpi.minecraft as minecraft  
import time
```

- Connect to the Minecraft game, remembering to check the capitalisation of the word `Minecraft`:

```
mc = minecraft.Minecraft.create()
```

- Put in the main game loop that senses your player's position, with a delay so it doesn't run too quickly:

```
while True:  
    time.sleep(1)  
    pos = mc.player.getTilePos()
```

- Add the `if` statement that checks whether your player is standing on the mat. This uses the `if` statement with an `and` to check two conditions. Both the x and z parts of the coordinate for your doormat must match the player's position for the program to say the player is standing on the doormat. Still have the x and z

coordinates for the doormat that you wrote down earlier? Enter them here, so that the program knows exactly where your doormat is:

```
if pos.x == 10 and pos.z == 12:  
    mc.postToChat("welcome home")
```

Time to see if your program works! By choosing Run→Run Module from the editor window, you can move around the Minecraft world. When your player stands on the doormat, the program should say “welcome home”, like Figure 2-5. Cool!



FIGURE 2-5 Walking on the doormat displays a “welcome home” message.



Notice how in the `if` statement you used the `==` (double equals) symbol? A common mistake here is to use an `=` symbol (single equals). Try that and see what happens; you should see an error when you run your program. Python uses a single equals sign (`=`) to mean “store the value on the right hand side of the statement into the variable on the left hand side of the statement” like `a = 3`. Python uses `==` (double equals) to mean “compare the value on the right hand side against the value on the left hand side”, like `if a == 3:`



Make sure you get the indentation correct when you code your Welcome Home Game. The code that is part of the `while True` loop is indented by one level. The `mc.postToChat()` that belongs to the `if` statement is indented by two levels. If you get the indentation wrong, your program will do the wrong thing.

Believe it or not, you have just learned all the basics you will be using in your other adventures. Every Minecraft program has to import modules, connect to the game, loop in a game loop, sense that something has happened and behave differently as a result. From these simple beginnings you will find that many great things will grow!



CHALLENGE

Do you think that checking the y coordinate (the up and down direction) will improve the detection of whether your player is standing on the doormat? Have a go. Modify your program to test all three coordinates of the doormat and see if the application works any better.



DIGGING INTO THE CODE

Chances are you will have typed something wrong at some point, in which case you will probably have seen a red error message on the Python Shell. It's a good idea to have a look at the type of errors you can make, so you don't take fright when you see them from time to time.

You might see a **syntax** error if you type the wrong symbols or get things in the wrong order. If you miss out a symbol or type in a variable name when it is not expected, you will get a syntax error.

For example, if you type the following line into the Python Shell, you will see the syntax error shown in Figure 2-6, because the equals sign is not expected to appear on its own; it is usually used together with variables and other values:

=

A screenshot of the Python 2.7.6 Shell window. The title bar says "Python 2.7.6 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, Help. The main window shows the Python interpreter prompt ">>>". The user has typed an equals sign "=" on a new line. A red error message "SyntaxError: invalid syntax" is displayed above the cursor. The status bar at the bottom right shows "Ln: 5 Col: 4".

```
>>> =
SyntaxError: invalid syntax
>>> |
```

FIGURE 2-6 Python shows that your code contains a syntax error.

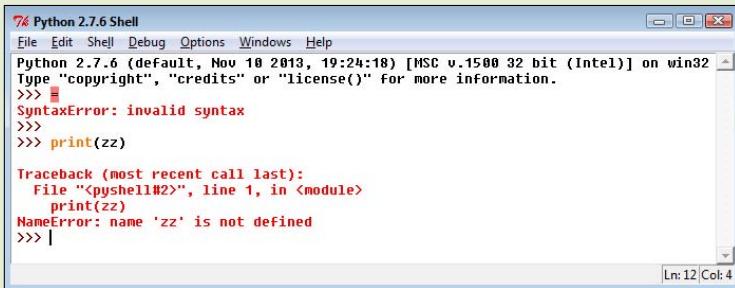
continued

continued

Now type this into the Python Shell:

```
print(zz)
```

You will see that Python tells you this is a name error (shown in Figure 2–7) because you have not yet stored a value in the variable name `zz`, so Python doesn't yet know about a variable called `zz`.

A screenshot of the Python 2.7.6 Shell window. The title bar says "Python 2.7.6 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main window shows the following text:

```
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
SyntaxError: invalid syntax
>>> 
>>> print(zz)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(zz)
NameError: name 'zz' is not defined
>>> |
```

The status bar at the bottom right shows "Ln: 12 Col: 4".

FIGURE 2–7 Python tells you about a name error.

Try typing in a `while` loop, but miss out that all-important colon (`:`) at the end:

```
while True
```

Again, you get a syntax error, because Python was expecting the colon but you did not use it.

Don't be alarmed when you see an error message coming back from your program. Look carefully at the line it is talking about and see if you can work out what you have typed incorrectly. Sometimes you might get a syntax error because of an error in an earlier line in the program, so check all around the error message to see if you can figure out what the problem is. If all else fails, ask a friend to look over your program for you. It is often easier to spot errors in other peoples programs than your own! When errors are reported on the Python Shell they also include the line number of the error. You can look at the bottom right of the IDLE window to see what line number your cursor is at, or you can use `Edit`→`GoTo Line` from the IDLE menu.



Syntax refers to the rules of a language (in this case, the Python language), and is mainly related to the order in which you type things.

Using Geo-Fencing to Charge Rent

Now you are going to write a new game, called `rent.py`. This game consists of a field with a fence around it. Your program will detect when your player is standing in this field and charge rent all the while the player is in the field. Your player's challenge is to remove any objects in this field in the fastest time possible, so that rent is as low as possible.

Your `welcomeHome.py` program used an `if` statement to check whether the coordinates of your player were the same as the coordinates of your doormat. For this to work, your player has to be standing exactly on the doormat. You can improve on this a bit by using a technique called **geo-fencing**, which you will use in this new game. The detection of your player's position is improved because it checks to see if your player is standing in an area of the Minecraft world, not just at a specific coordinate. This allows you to detect when your player is standing anywhere in a larger region of the world, and you don't have to be quite so precise when standing on blocks.

Geo-fencing is a general technique that builds a virtual fence around coordinates on any map. When something enters this virtual fenced region, something happens as a result. “Something” could mean a player in the Minecraft world, a person in the real world, or a device like a lawnmower or even an animal in the real world.

In many cases, geo-fencing is used to alert the user when something—such as cattle, for example, or a vehicle—moves outside of a pre-determined region. You can find out more about real world uses of geo-fencing at <http://en.wikipedia.org/wiki/Geo-fence> and also find out how geo-fencing is being used in real life to track elephants in the wild here: <http://www.cbsnews.com/news/kenya-uses-text-messages-to-track-elephant/>



For geo-fencing to work, you need two things: an object or region to track, and the allowed coordinates of the “fence” around that object. You’re now going to find out how to do this by building a field with a real Minecraft fence around it and noting down the coordinates of the corners of the field.

You don’t have to build the fence for this program to work but it makes the game a bit more interesting to play if you do, because you have to jump over the fence or run a long way round to get to the gap in the fence, to get in and out of the field. Adding obstacles to games makes them more challenging and exciting. This Minecraft fence will line up with the coordinates of your virtual geo-fence, and it gives your program a simple way to answer the question “is the player in the field or not?”



When you have built your field and fence, it should look something like Figure 2-8.

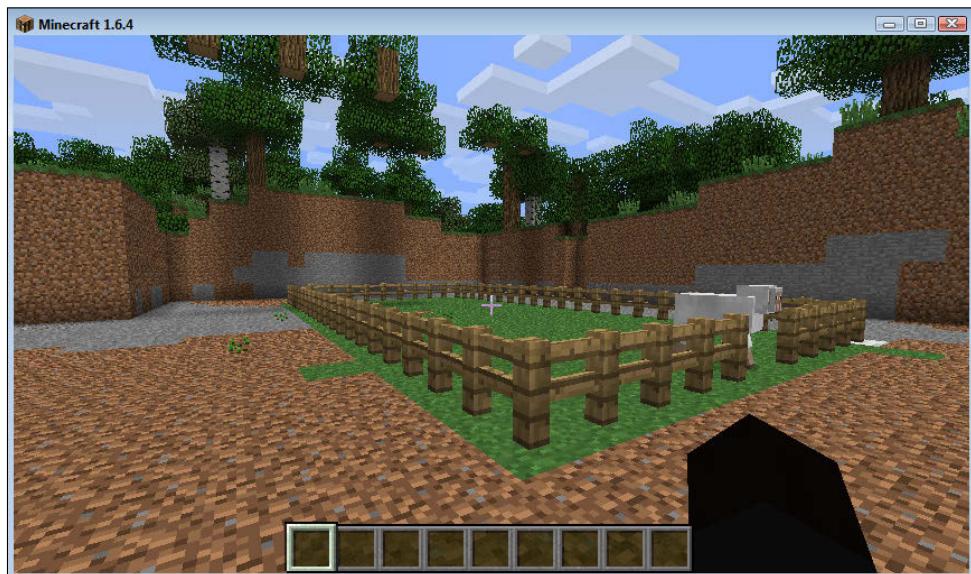


FIGURE 2-8 A field with a fence around it. The field is 10 blocks by 10 blocks in size.

Working out the Corner Coordinates of the Field

In order to geo-fence the field with your program, you need to have an accurate record of the coordinates of the corners of the field so that you can enter these numbers into your Python program.

The easiest way to do this is to run your `whereAmI.py` program from earlier, then have your player run to each corner of the field to get the x, y and z coordinates of each corner. On a piece of paper, make a sketch of what you're building and write the coordinates on the sketch. You will need them when you write your program. Remember that the z coordinate gets bigger as your player moves south and smaller as your player moves north, as you saw in Figure 2-1.

In Figure 2-9, you can see the coordinates of the field that I built when writing this book. I have written down the coordinates for all four corners, as they are needed to work out the smallest and biggest numbers in both the x and z directions. Note how the smallest coordinates are at the top left corner of the field.

You need four numbers for your geo-fence program. First, you need to know the smallest and biggest x coordinates, so work these out from your diagram. In the previous example, the smallest x is 10 and the biggest x is 20. I have called the smallest x coordinate X1 and the biggest x coordinate X2.

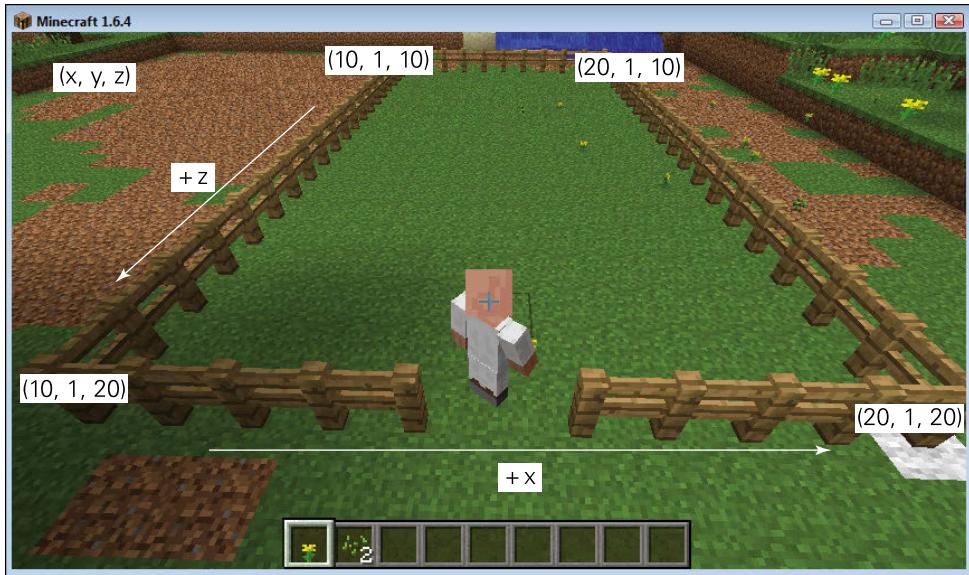


FIGURE 2-9 Now that you have the coordinates of the corners of the field, it is ready for geo-fencing.

Depending on where your player is standing in the Minecraft world, some of the coordinates might be negative (e.g. $x=-5$, $y=0$, $z=-2$), which can make these calculations a little tricky. You might find it easier to move to a place in the Minecraft world where all three parts of the coordinate are positive (e.g. $x=5$, $y=2$, $z=4$) first. You can check the coordinates where your player is standing by looking at the top left corner of the Minecraft screen on the Raspberry Pi, or pressing F3 on the PC/Mac.



Next you need to work out the smallest and biggest z coordinates. In the example, the smallest z is 10 and the biggest z is 20. I have called the smallest z coordinate $Z1$ and the biggest z coordinate $Z2$:

```
X1 = 10  
Z1 = 10  
X2 = 20  
Z2 = 20
```

Number your biggest and smallest coordinates in the same way, as you will need them in the next step—writing the program.

Writing the Geo-Fence Program

The final step is to write the program that makes the geo-fencing work and charges rent whenever the player is standing in the field. This program is similar in structure to the `whereAmI.py` program.

You are also going to use **constants** in this new game to make it easier to move your field around later. That way you won't have to hunt through your programs to find the coordinate values you need to change if you move your field.



A **constant** is a name for a part of the computer's memory (just like a variable) where you can store values that normally don't change while the program is running.

The Python language does not really have a special way of handing constants, unlike other programming languages. Python programmers usually use a programming convention (which simply means a standard way of working) where constants are always typed in using upper case letters. This makes it more obvious when you're programming something that it is a constant, and you should not change it once you have set an initial value. Other programming languages have their own way of using constants, and in these languages an error is raised if you try to change their value. Python does not have this feature.

To write the program, follow these steps:

1. Start by opening a new window in IDLE by choosing **File**→**New File** from the menu.
2. Choose **File**→**Save As** from the menu, and name the file `rent.py`.
3. Import the modules needed for this program:

```
import mcpi.minecraft as minecraft  
import time
```

4. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

5. Now you need to define some constants for the four coordinates of the geo-fence. Make sure you use the coordinates that you worked out earlier. I've used the coordinates that I used in my program, but yours might be different:

```
X1 = 10  
Z1 = 10  
X2 = 20  
Z2 = 20
```

6. Next, create a variable that will keep a running tally of how much rent the player has been charged. When the game starts you haven't charged any rent, so create a new variable called `rent` and set it to zero:

```
rent = 0
```

7. Create the main game loop with a `while` statement. Your program will tick round once every second, so use a `sleep` statement to make it delay 1 second each time round the loop. By delaying a short time each time around the loop slows the program down a little, and also it gives you a convenient method of timing how long your player is in the field. Later you will add 1 to the rent variable each time round the game loop, and as you have added the one-second delay at the top of this loop, this means that the rent will increase by 1 for every second. Make sure you get the indents right at this point:

```
while True:  
    time.sleep(1)
```

8. In order to check whether your player is in the geo-fenced region, you need to know their position. Just like in your earlier programs, do this with `player.getTilePos()`:

```
pos = mc.player.getTilePos()
```

9. You've now arrived at the most important part of the program. This is where you direct the program to use the four coordinates you worked out earlier to decide whether the player is standing in the field and, if he is, to charge him rent and inform him of this by posting a message to the chat. To do this, type:

```
if pos.x>X1 and pos.x<X2 and pos.z>Z1 and pos.z<Z2:  
    rent = rent+1  
    mc.postToChat("You owe rent:"+str(rent))
```

At this point in the program, you have to be very careful to get the indentation correct. Python uses indentation to work out which program statements belong together. You can see that all of the statements under `while True:` are indented by one level, so Python knows that all of these statements belong to the `while` loop. But look carefully at the `if` statement: the next two lines in the program are indented another level. This means that these two statements are part of the `if` statement and will run only if your player is standing inside the field.



Save the program again, and run it by choosing Run ➔ Run Module from the menu.

Have some fun making the player run in and out of the field. When he is inside the field, a message should appear on the chat every second, saying how much rent he has

been charged. When he runs out of the field he won't be charged rent. Your program also remembers how much rent you have been charged, so when you run back into the field again, it doesn't forget how much you already owe.

CHALLENGE



To make this game more fun, your player needs something to do while he's in the field. Create a few random blocks in different locations inside the field, then start your program running again. The challenge is to collect all the blocks while paying as little rent as possible. Just hitting the block to remove it could be considered collecting the block. The program won't know this has happened but it will make your game more interesting. You can come up with all sorts of objects to scatter throughout the field and challenge your friends to collect them, with the winner being the person who collects all the objects while paying the least rent!

Moving Your Player

There is a way to make your game more challenging and exciting, involving another feature of the Minecraft API that you'll find useful in your game creations—moving the player to a different position in the Minecraft world! To do this, you will be modifying your existing `rent.py` program so that if the player is in the field for more than three seconds, he gets catapulted into the sky and out of the field and has to get back into the field again.

You will first need to find a position just outside the field where you want the player to land when he has been in the field for too long. The simplest way to do this might be to look at the biggest of the X and Z coordinates for your field, and add 2 to each of them to get some new coordinates. For the dramatic effect you are going to create, choose a Y coordinate that is up in the sky. The actual number you choose will depend how far up in the Minecraft world you have built your field, but if your field is built at `y = 0` then you could choose a value for `y` that is round about 10. This will catapult your player into the sky and gravity will make them fall to the ground again.

You are now going to modify your `rent.py` program so that if the player is in the field for more than three seconds he gets thrown out of the field, using the following steps:

1. Set three new constants at the top of your program that remember a home position. This home position will be just outside of the field, and when your player gets catapulted out of the field they will be moved to these coordinates. The new lines to add are marked in bold:

```
X1 = 10  
Z1 = 10
```

```
X2 = 20  
Z2 = 20  
  
HOME_X = X2 + 2  
HOME_Y = 10  
HOME_Z = Z2 + 2
```

- Now you are going to time how long your player is in the field for, and to do this you need another variable which you can call **inField**. It will store the number of seconds that the player has been in the field for and it will be used to decide when your player has been in the field for too long! Again, you only need to add the parts shown in bold:

```
rent = 0  
inField = 0
```

- Next, you need to add a line of code that adds 1 to the **inField** variable if the player is in the field. To do this, add the code shown in bold:

```
if pos.x>X1 and pos.x<X2 and pos.z>Z1 and pos.z<Z2:  
    rent = rent+1  
    mc.postToChat("You owe rent:"+str(rent))  
inField = inField+1
```

- Now you're going to add an **else** statement, so that the program resets the timer to zero if the player is not in the field. Make sure you get the indentation correct on the **else** statement so that Python knows exactly what you mean. You will learn about the **else** statement in a moment as well as what the **#** symbol is used for but, for now, just type in the code shown in bold:

```
mc.postToChat("You owe rent:"+str(rent))  
inField = inField+1  
else: # not inside the field  
inField = 0
```

- Finally, you need to add some additional lines of code at the end of the program to catapult the player into the sky and out of the field if he is in the field for more than three seconds. Gravity will make him fall to the ground and he will have to run back into the field again. This code goes right at the end of your program. The **if** must be indented one level because it is part of the **while True** loop, and the statements that belong to the **if** must be indented two levels from the far left of the program. Type the following:

```
if inField>3:  
    mc.postToChat("Too slow!")  
    mc.player.setPos(HOME_X, HOME_Y, HOME_Z)
```

6. Save your program and run it using Run→Run Module from the editor menu.

You should be having a lot of fun with the game now because you have to plan your strategy carefully, running in and out of the field to avoid getting catapulted into the sky! Now, as you did before, choose a variety of objects and scatter them randomly around the field. Challenge yourself and your friends to “collect” (that is, destroy) as many blocks as you can, while clocking up the smallest possible rent bill. (See Figure 2-10.)

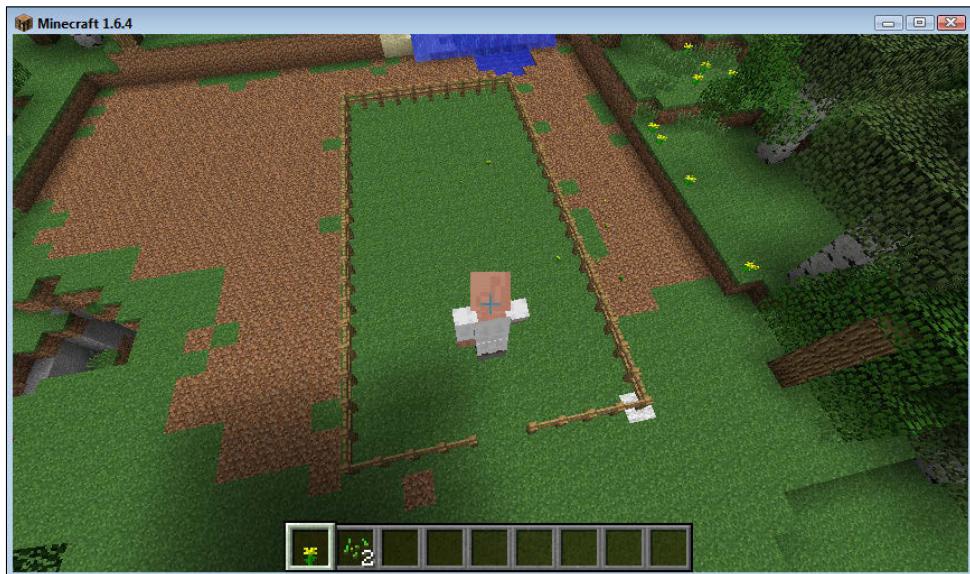


FIGURE 2-10 Your player is catapulted up into the sky when he has been in the field for too long.

DIGGING INTO THE CODE

You have now used two new parts of the Python language, and these need to be explained further.

First, you used an `else` statement. `else` is an optional part of an `if` statement. It should always be at the same indent level as the `if` that you pair it with, and the statements that you want to run as part of the `else` should also be indented. Here is an example:

```
if a>3:  
    print("big")  
else:  
    print("small")
```

If the `a` variable holds a value greater than 3, Python will print “big” on the Python Shell, `else` (meaning, if it is not greater than 3) it will print “small” on the Python Shell. Or put another way, if `a>3` is True, it prints big, else if `a>3` is False, it prints small.

An `if` statement doesn’t have to have an `else` (it is optional), but sometimes in your programs you want to handle the part that is true in a different way than the part that is false.

The other thing you used here was a comment. You will find comments a useful way of leaving little reminders in your program for yourself or anyone else who reads it.

A comment starts with a hash symbol, like this:

```
# this is a comment
```

You can put a comment anywhere on a line. Wherever you put a hash symbol, everything to the end of that line will be ignored by the Python language.

Further Adventures in Tracking Your Player

In this adventure, you have learned and put into practice the basics of a game loop, using sensing and geo-fencing, to create a fun game. When you make games, it’s a great idea to get your friends and family to play them and tell you what they like or don’t like about them. It’s a good way of getting feedback to help you improve the game.

- The Welcome Home Game has a problem: if your player stands on the doormat, it repeatedly says “welcome home” and fills up the chat. Can you think of a way to modify this program so that it only says “welcome home” once when you come into your house?
- A good way to come up with ideas for exciting games is to play other games and see how they keep the player gripped. Play lots of other computer games and make a list of as many different things as possible that a game could do, in terms of the player’s position changing.
- Search on the Internet for geo-fencing and see if you can find some of the ways it is used to good effect in industry. If any of these ideas grab your interest, have a go at writing a Minecraft program that uses some aspect of them by writing your own simple game based on what you have learned in this adventure.

Quick Reference Table

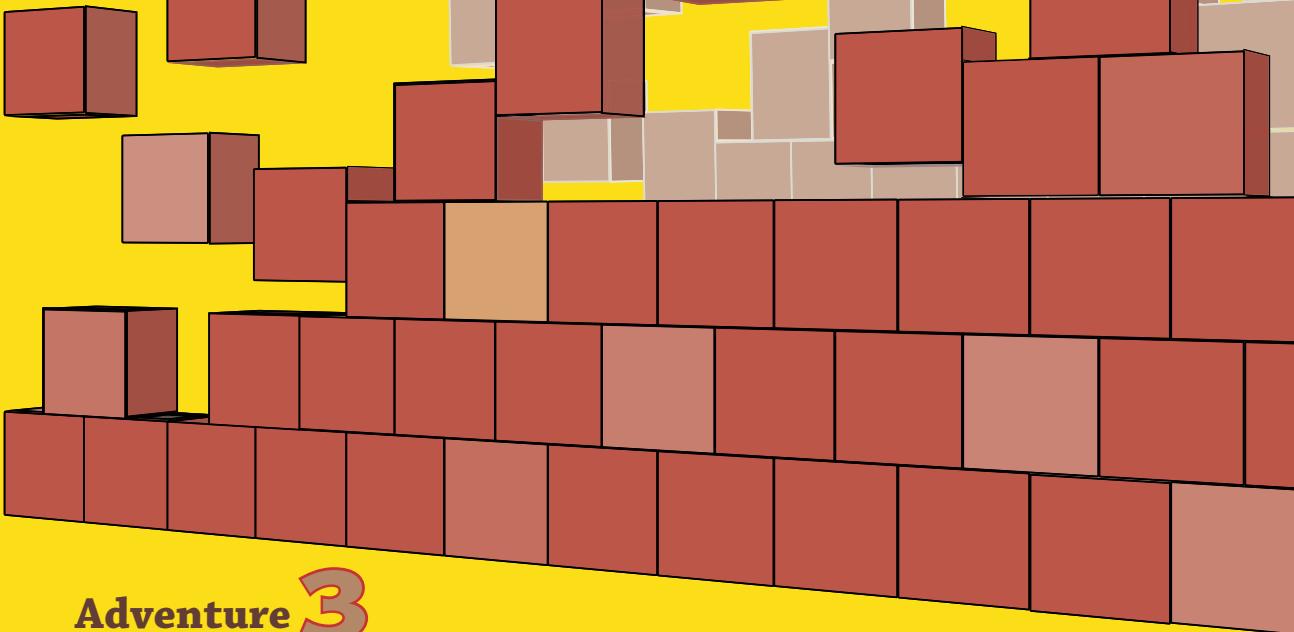
Importing the Minecraft API	Creating a connection to Minecraft
<pre>import mcpi.minecraft as minecraft</pre>	<pre>mc = minecraft.Minecraft.create()</pre>
Getting the player's tile position	Posting a message to the Minecraft chat
<pre>pos = mc.player.getTilePos() x = pos.x y = pos.y z = pos.z</pre>	<pre>mc.postToChat ("Hello Minecraft")</pre>
Setting the player's tile position	
<pre>x = 5 y = 3 z = 7 mc.player.setTilePos(x, y, z)</pre>	



Achievement Unlocked: Creator of an exciting game that changes as your player moves around in Minecraft.

In the Next Adventure...

In Adventure 3, you'll learn how to automate the building of large structures such as houses, using Minecraft blocks and Python loops. With a Python program you can build huge structures much faster than you could build them by hand. You will be able to construct a whole Minecraft town in Python in less time than it takes your friends to build it manually.



Adventure 3

Building Anything Automatically

BUILDING THINGS IN Minecraft is great fun. You can build almost anything you can dream up, limited only by the size of the Minecraft world and your imagination. You can build houses, castles, underground waterways, multi storey hotels with swimming pools, and even whole towns and cities! But it can take a lot of time and hard work to build complex objects with many different types of blocks, especially if there is a lot of repetition. What if you could automate some of your building tasks? Wouldn't that look like magic to your friends?

A programming language like Python is ideal for automating complex tasks. In this adventure, you will learn some magic that allows you to automatically build large numbers of blocks inside the Minecraft world, and then loop through those instructions to build huge repeating structures, such as a whole street of houses (see Figure 3-1). Your friends will be amazed at your huge creations as they walk through rows and rows of houses and wonder how you built such complex structures! You will also learn how to read numbers from the keyboard when your program first runs, and this will enable you to write programs that your users can vary without changing your program code!

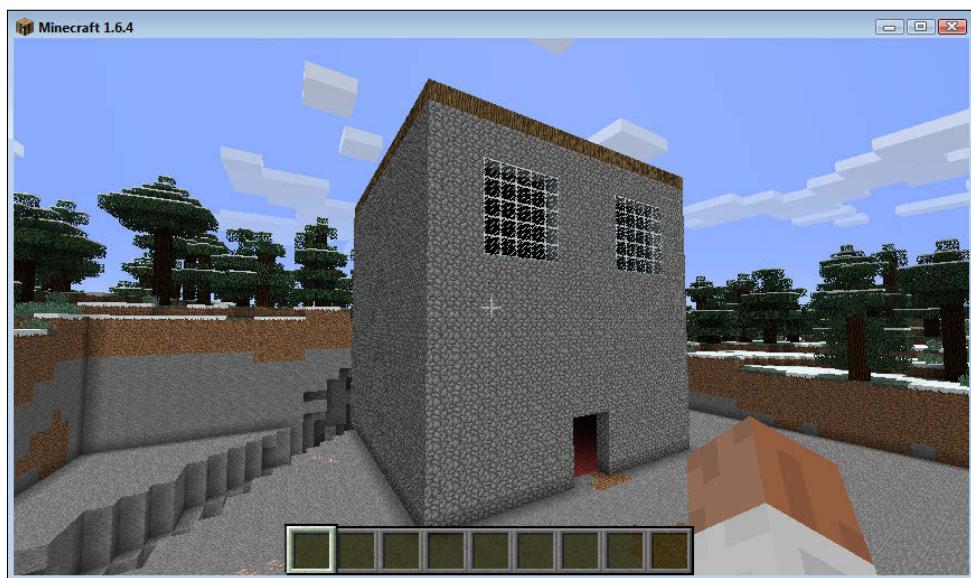


FIGURE 3-1 A house built from Minecraft blocks

Creating Blocks

Each type of block in the Minecraft world has its own block number. If you are a regular Minecraft gamer, you probably already know a lot of block types by their numbers. However, the Minecraft programming interface allows you to refer to block types by name rather than number, which makes things much easier. The name is just a constant (you met constants in Adventure 2) that holds the number of the block type. There is a list of the most common block types and their names and numbers in the reference section in Appendix B.

The Minecraft world is divided into invisible cubes (each with its own unique coordinate), and the computer remembers the number of the type of block that's in that invisible cube at any one time, even if it's just a block of type **AIR**. As a Minecraft programmer, you can ask the Minecraft programming interface for the block type at any particular coordinate, and you can change the block type of the block at any coordinate. This will change the block that is displayed there. You can, for example, build bridges by changing the block type from **AIR** to **STONE**, and you will do this in Adventure 4.

You are now going to start by writing a simple program that demonstrates this, by automatically placing a single block immediately in front of your player. Once you can place a single block in the Minecraft world, you can build anything you can possibly imagine, automatically!

1. Start up Minecraft, IDLE, and if you are working on a PC or a Mac, start up the server too. You should have had a bit of practice with starting everything up by now, but refer back to Adventure 1 if you need any reminders.

2. Open IDLE, the Python Integrated Development Environment (IDE). This is your window into the Python programming language and where you type and run all your Python programs.
3. Click File→New File from the menu. Save this program by clicking File→Save As and name the file **block.py**. Remember to store your programs inside the **MyAdventures** folder, otherwise they will not work.
4. Now, import the modules you need for this adventure. You'll be using an extra module called **block**, which holds all the constant numbers for all the block types that Minecraft supports. Do this by typing:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block
```

5. Connect to the Minecraft game by typing:

```
mc = minecraft.Minecraft.create()
```

6. Get your player's position into the **pos** variable. You will use this position to calculate the coordinates of a space in front of your player, where you are going to create your new block:

```
pos = mc.player.getTilePos()
```

7. Now create a block in front of the player, using coordinates that are relative to your player's position. You can read all about **relative coordinates** after this section. By using **pos+3** in your program it ensures that the stone block doesn't appear right on top of your player! Do this by typing:

```
mc.setBlock(pos.x+3, pos.y, pos.z, block.STONE.id)
```

8. Click File→Save to save your program and then run it by choosing Run→Run Module from the editor menu.

You should now see a block of stone very close to your player! You have just programmed the Minecraft world to create a block in front of you. From these simple beginnings, you can now create some really interesting structures automatically by programming Minecraft.

CHALLENGE

Stone isn't a very interesting block. Look at the blocks in Appendix B and experiment by changing the **STONE** in this program to other blocks. Some of the blocks don't work on all platforms, so Appendix B lists which platforms they work on.

GLOWING_OBSIDIAN for example works on the Raspberry Pi, but not on the PC/Mac versions. Some blocks are affected by gravity, so experiment with **WATER** and **SAND** for some interesting effects!





Relative coordinates refer to a set of coordinates such that their position is relative to some other point (or in Minecraft, your player). For example, `pos.x`, `pos.y+10`, `pos.z+3` is a location inside the Minecraft world 10 blocks above and 3 blocks south of your player: in other words, the coordinates are relative to the position of your player. As your player moves around the Minecraft world, the relative coordinates change as well.

Absolute coordinates refer to a set of coordinates that uses numbers at a fixed location to represent the location of a point (or in the case of Minecraft, a block). The coordinates `x=10`, `y=10`, `z=15` are an example of absolute coordinates. Every time you use these coordinates you refer to a block at location 10, 10, 15 inside the Minecraft world, which is always the same location.

CHALLENGE



If you have already done Adventure 2, load and run your `whereAmI.py` program and move to a location in the Minecraft world that has some free space. Jot down the x, y and z coordinates where your player is standing. Modify your `block.py` program to `setBlock()` at that **absolute coordinate**, and run the program to see what it does. How do you think relative coordinates and absolute coordinates might help in your programs?

Building More than One Block

From this basic beginning, you can build anything! You can now extend your program to build more than one block. When you are building with Minecraft blocks, the only thing you need to remember is that you will need to use some simple maths to work out the coordinates of each block that you want to create.

You are now going to extend your `build.py` program to create five more blocks in front of your player, using one `setBlock()` for each block that you want to create. The program you're about to write will create a structure that looks a bit like the dots on a dice. To do this, work through the following steps.

1. Start by choosing File ➔ Save As from the editor menu and saving your program as `dice.py`.
2. Now add these lines to the end of the program:

```
mc.setBlock(pos.x+3, pos.y+2, pos.z,     block.STONE.id)
mc.setBlock(pos.x+3, pos.y+4, pos.z,     block.STONE.id)
mc.setBlock(pos.x+3, pos.y,    pos.z+4, block.STONE.id)
mc.setBlock(pos.x+3, pos.y+2, pos.z+4, block.STONE.id)
mc.setBlock(pos.x+3, pos.y+4, pos.z+4, block.STONE.id)
```

- Save the program and run it. You should see a simple structure that looks like six dots on a dice, in front of your player! (See Figure 3-2.)



FIGURE 3-2 The Minecraft dice in front of the player shows six stone dots.

CHALLENGE

Modify the `dice.py` program so that some of the space around stone dots is filled in with a white block, so that it looks more like the square face of a dice. Experiment with setting different blocks to `STONE` or `AIR` to create different numbers on the dice. Remember that there is a detailed list of blocks in Appendix B.



Using for Loops

Building with individual blocks allows you to build anything you like, but it's a bit like drawing a complex picture on a computer screen by hand, one dot at a time. What you need is some way to repeat blocks, so that you can build bigger structures without increasing the size of your program.

Building Multiple Blocks with a for Loop

Fortunately, like any programming language, Python has a feature called a loop. You encountered loops already in Adventure 2 with the `while True:` game loop. A loop is just a way of repeating things multiple times in a programming language like Python.

The loop you will use here is called a `for` loop, sometimes called a **counted loop** because it counts a fixed number of times.



The `for` loop is called a **counted loop** or a count controlled loop, because it counts a fixed number of times. You decide, in the `range()` statement, how many times it counts. The `for` loop in Python is quite special because it can count things other than just numbers, and you will meet its more advanced uses in Adventure 6.

Just as you did with the `while` loop you used to build your game loop in Adventure 2, you must indent the program statements that belong to the `for` loop, so the Python language knows that only these statements should be repeated every time the loop runs.

To understand how this works, use the Python Shell to try out a `for` loop by following these steps:

1. Click the Python Shell window, just to the right of the last `>>>` prompt.
2. Type the following and press the Enter key on the keyboard:

```
for a in range(10):
```

Python won't do anything yet, because it is expecting the program statements that belong to the loop (these are often called the loop body).

3. The Python Shell automatically indents the next line for you by one level, so that Python knows that your `print()` statement belongs to the `for` loop. Now type in this `print()` statement:

```
    print(a)
```

4. Press the Enter key twice. Your first press marks the end of the `print()` statement; your second tells the Python Shell that you have finished the loop.

You should now see the numbers 0 to 9 printed on the Python Shell screen.

DIGGING INTO THE CODE

The `for` loop that you just used has some interesting features:

```
for a in range(10):  
    print(a)
```

The `a` in this case is called the loop control variable. This means it is the variable that is used to hold the value of the loop every time it runs the code in the loop body. The first time it runs, it has the number 0 in it, the second time it has the number 1 in it, and so on.

The `range(10)` tells Python that you want it to generate a sequence of ten numbers from 0 to 9, through which the `for` loop will count.

The colon (`:`) at the end is just like the colon that you used in earlier adventures with your `while` loop and your `if` statements. The colon marks where the start of the body code begins. The body code is the code that (in this case) is to be repeated 10 times.

Any Python statements inside the loop (in other words, the statements that are indented one level from the `for` statement) will be able to access the variable `a` and this variable will hold a different value each time round the loop.

You can use the loop control variable `a` anywhere in the program statements inside the loop body for any purpose. You don't have to call the variable `a` all the time. You can think up any name you like for it, and it is good practice to think up variable names that are more descriptive, so that others who read your program can more easily understand it. A better name here for the variable instead of `a` would be `count` or `number_of_times`.

Building a Huge Tower with a `for` Loop

How would you like to build an enormous tower out of blocks inside the Minecraft world? Now that you know all about `for` loops, you can. Just follow these steps:

1. Start writing a new program by choosing `File`→`New File` from the menu. Choose `File`→`Save As` from the menu, and name it `tower.py`.
2. As usual, import the modules you need:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. If you build your tower where your player is standing it will be easier to find, so your first job is to identify the position of your player by typing:

```
pos = mc.player.getTilePos()
```

5. Your tower is going to be 50 blocks high, so start building it with a `for` loop that counts 50 times. Don't forget the colon (`:`) at the end of the line:

```
for a in range(50):
```

6. The next line is indented, because it belongs to the body of the `for` loop. The `y` coordinate controls height within the Minecraft world, so add the loop control variable `a` onto the player's `y` position. This tells the program to build at an increasing height in the Minecraft world each time round the loop:

```
mc.setBlock(pos.x+3, pos.y+a, pos.z, block.STONE.id)
```

7. Save the program and run it. Has it worked? Your `for` loop should have created a massive tower in front of your player, similar to the one in Figure 3-3. Start counting—is it 50 blocks high?

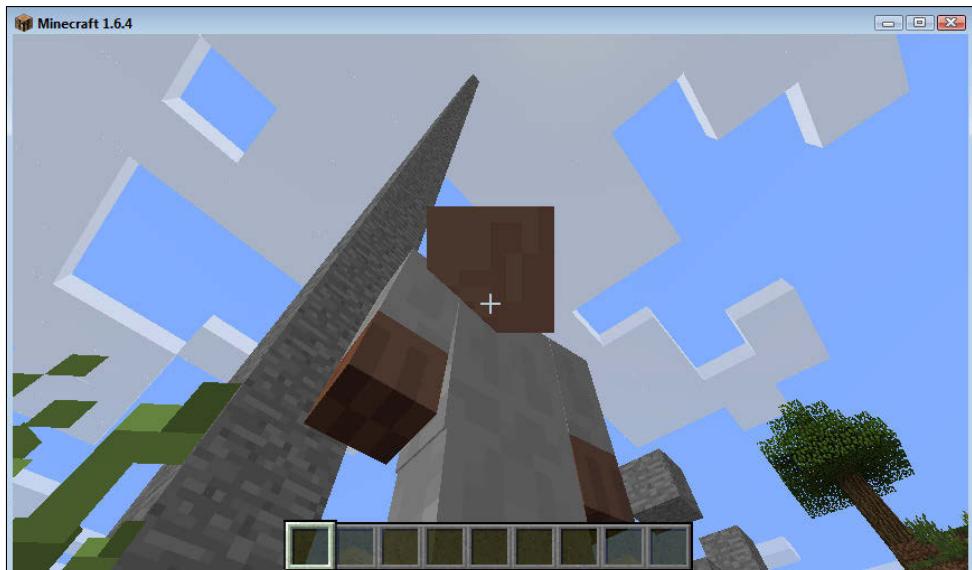


FIGURE 3-3 This huge tower was created inside the Minecraft world using a `for` loop.



In the previous code, the `for` loop cycles around 50 times, and each time round the loop, it runs the indented lines below it, such that the `a` variable has a value one bigger than the previous time round the loop. It adds the value in the `a` variable (the loop control variable) onto the `pos.y` variable (which is the players height above bedrock). This then creates a tower 50 blocks high!

CHALLENGE

Change the `range()` inside the `for` loop to create a tower that is even taller. How tall a tower can you create before you can no longer see the top of it?



Clearing Some Space

Sometimes, finding enough space in the Minecraft world to build your structures can be a bit frustrating. There are usually a lot of trees and mountains around your player, meaning there is often not enough space to build big structures. You can solve this problem by writing a program that clears some space for you to build whatever you want.

Using `setBlocks` to Build Even Faster

All the blocks in the Minecraft world have a block identity (id) number, and this includes the blank spaces that you see in front of you, called `block.AIR.id`. So, if you make sure every block in a large area is set to `block.AIR.id`, it will clear a nice space ready for building other objects. The id number (which is the block type number discussed earlier) is listed in Appendix B for the most common blocks you will use. For example, `block.AIR.id` is 0 and `block.STONE.id` is 1. Remember, the empty space you see in the Minecraft world is really just a block with a block id of `block.AIR.id`—it is a 1 metre cube space filled with air!

Computers are very fast, but the more program statements you use inside a `for` loop, the slower the program will run. It takes a little while for the Minecraft game to interpret your request to set a block inside the world, change the block type, and update the display on the screen. You could always write a big loop that sets hundreds of blocks in front of you to `block.AIR.id` to clear some space, and that would work, but it would be really slow. Fortunately there is a better way to set a lot of blocks in one go, and it is called `setBlocks()`. (Note the extra `s` at the end of the name.) `setBlock()` sets one block, `setBlocks()` sets a whole 3D area of blocks in one go!



The Minecraft programming interface has a `setBlocks()` statement that you can use to give all the blocks inside a three-dimensional (3D) rectangular space the same block id. Because this is a single request to the Minecraft game, Minecraft can optimise how it does this work, meaning it will run much quicker than if you used a `setBlock()` inside a `for` loop like you did with the tower.

Because `setBlocks()` works on a 3D area, it needs two sets of coordinates—the coordinates of one corner of the 3D rectangle, and the coordinates of the opposite corner. Because each 3D coordinate has three numbers, you need six numbers to completely describe a 3D space to Minecraft.

You are now going to write a useful little utility program that you can use any time you want to clear a bit of space in front of you to do some building:

1. Start a new program by choosing File→New File from the menu.
2. Save the program using File→Save As and choose the name `clearSpace.py`.
3. Import the modules that you need:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block
```

4. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

5. You want to clear space in front of the player, and you will be using relative coordinates to do this. First get the player's position:

```
pos = mc.player.getTilePos()
```

6. Now clear a space that is 50 by 50 by 50 blocks. The bottom left corner of your space will be the position of your player, and the top right corner will be 50 blocks away in the x, y and z dimensions:

```
mc.setBlocks(pos.x, pos.y, pos.z, pos.x+50, pos.y+50, pos.  
z+50, block.AIR.id)
```

7. Save your program.

The fantastic thing about the program you've just created is that it allows you to walk to any location inside Minecraft and clear a 50 by 50 by 50 area any time you like; just run the program by choosing Run→Run Module from the menu, and the trees and mountains and everything nearby will magically vanish before your eyes!

Move around the Minecraft world and re-run your program to clear lots of space in front of your player.

Reading Input from the Keyboard

Another useful thing you can do to your `clearSpace.py` program is to make it easy to change the size of the space that is cleared. That way, if you are only going to build a small structure, you can clear a small space but if you know you are going to build lots of big structures, you can clear a huge space first.

To do this, you will use a new Python statement called `raw_input()` to read a number from the keyboard first. You can then go to any part of the Minecraft world, run

your program and simply type in a number representing the size of the space you want to clear, and then your program will clear all that space for you. This means you don't have to keep modifying your program every time you want to clear a different size of area in the Minecraft world.

There are two main versions of Python—Python 2 and Python 3. The Minecraft programming interface is written specifically to work with Python 2, and you use Python 2 for all the adventures in this book. It is possible to write Minecraft Python programs in Python 3 but you need an updated mcpi module. If you see any Python 3 programs on the Internet, they will use `input()` instead of `raw_input()` to read from the keyboard. At the time of writing, the precise version numbers of the latest versions are 2.7.6 and 3.3.3. There are significant differences between Python 2 and Python 3, which is why it's important to stick to using Python 2 for this book.



Now that you understand what `raw_input()` does, it's time to add it to your program. To do this, you only have to make two small modifications to your `clearSpace.py` program. Follow these steps:

1. First, use `raw_input()` to ask the user to type in a number that represents the size of the space to clear, and store that number inside a variable called `size`. Just like in Adventure 2 where you used `str()` to convert something to a string, here you use `int()` to convert the string from `raw_input()` into a number that you can perform calculations with. Try this line without the `int()` to see what happens! Type in only the new lines that are marked in bold:

```
pos = mc.player.getTilePos()  
size = int(raw_input("size of area to clear? "))
```

2. Next, modify the `setBlocks()` line so that instead of using the number 50, it uses your new variable `size`:

```
mc.setBlocks(pos.x, pos.y, pos.z, pos.x+size, pos.y+size,  
pos.z+size, block.AIR.id)
```

3. Save your program and run it by choosing Run→Run Module from the menu.

Now move to somewhere in the Minecraft world where there are lots of trees or mountains. Type a number into the Python Shell Window when prompted—perhaps choose 100 or something like that—and press the Enter key on the keyboard. All the trees and mountains near your player should magically disappear and, as in Figure 3-4, you will have a nice space to build other structures!

Keep this little utility program, as it will be useful later when you need to clear a bit of space in the Minecraft world.



FIGURE 3-4 After running `clearSpace.py`, note how some of the trees are sawn off at the trunks!

CHALLENGE



The `clearSpace.py` program clears a cuboid space near the player, using the player's position as the bottom left corner of that space. However it depends which way your player is facing as to whether you will see this space at first without turning around. It might be more convenient for your purposes if it cleared some space with the player at the centre of that space. If so, you can modify the relative coordinates in your `setBlocks()` statement to calculate the two corner coordinates of the 3D rectangle to clear, so that your player is placed at the centre. You might also want to get `clearSpace.py` to "dig down" a few blocks by reducing the y coordinate, to give you some space to lay foundations, grass or other ground blocks. Try this now to see if you can work out how to do it. It will be a much more useful program!

Building a House

When playing Minecraft in survival mode, one of the first things you need to is build a shelter for your player to protect him from the dangers lurking in the Minecraft night. What if you could build a house with the touch of a button? Fortunately, when programming with Minecraft, you can turn complex tasks into just that—the touch of a button.

In the `clearSpace.py` program, you learned how to set a large number of blocks to the same block type, by using just one programming statement. Now you are going to learn how to use the same techniques to build a house really quickly.

One way to build a house might be to build each wall using a separate `setBlocks()`. For that, you would need to use four separate `setBlocks()` and quite a lot of maths to work out all the coordinates of every corner of every wall. Fortunately, there is a quicker way to build hollow rectangular structures: all you have to do is build a huge cuboid space, and then use `setBlocks()` again with slightly different coordinates to carve out the inside with the `AIR` block type.

Before you go any further, you need take some time to make sure you understand the maths associated with the design of the house you are going to build, as you will need to get the coordinates right to write the program. Figure 3-5 shows a sketch of the house design, labelled with all the important coordinates that you will use to build it. When you are building something complex, it is important to sketch it out on paper and work out all the important coordinates first. This is a very special house, because it is the program that calculates the size and position of the doorway and windows. You will see a little later why this is important.

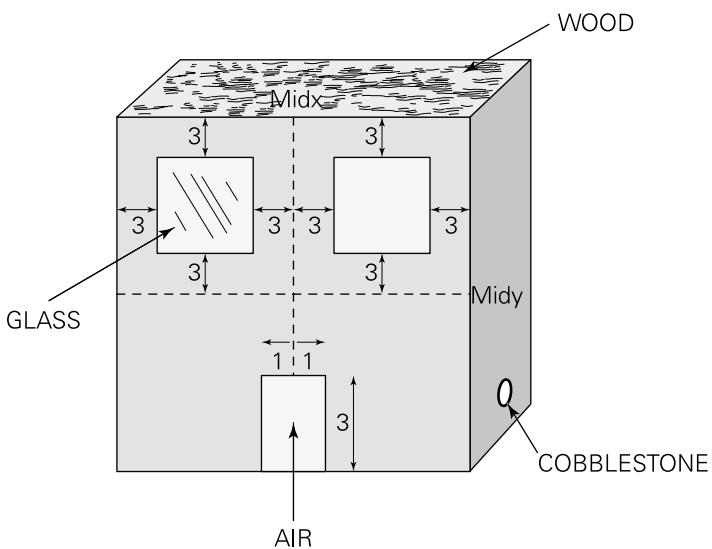


FIGURE 3-5 A design of your house on paper, with all the important coordinates worked out



To watch a tutorial on how to build your house, visit the companion website at www.wiley.com/go/adventuresinminecraft and choose the Adventure 3 Video.

Now that you have a design for your house, try and build it by following these steps:

1. Start a new program by choosing File ➔ New File from the menu.
2. Save this program by using File ➔ Save As from the menu, and call your program `buildHouse.py`.
3. Import the required modules:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block
```

4. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

5. Use a constant for the size of your house. You use this `SIZE` constant quite a lot in your house builder code. Using a constant here, instead of using the number `20` all over your code, will make it much easier to alter the size of your house later:

```
SIZE = 20
```



Remember that it is a programmer's convention that you put constants in upper case. There is no rule in the Python language that says you have to do this, but it helps you to remember that this is a constant and you intend it not to be changed while the program is running.

6. Get the player's position so you can build the house just nearby:

```
pos = mc.player.getTilePos()
```

7. Store the x, y and z coordinates of the player in new variables. This will make the next few program statements easier to type and read, and will help when you come to do some other clever construction tasks with your house design later on. The x variable will be set to two blocks away from the player position, so that the house is not built right on top of the player:

```
x = pos.x+2  
y = pos.y  
z = pos.z
```

- Calculate two variables called `midx` and `midy`, which are the midpoints of the front of your house in the x and y directions. This will make it easier for you to work out the coordinates of the windows and doorway later on, so that if you change the size of your house, the windows and doorway will change their position too and fit properly.

```
midx = x + SIZE/2  
midy = y + SIZE/2
```

- Build the outer shell of the house as a huge rectangular 3D area. You can choose any block type here, but cobblestone is a good one to start with, making it an old house:

```
mc.setBlocks(x, y, z, x+SIZE, y+SIZE, z+SIZE,  
block.COBBLESTONE.id)
```

- Now carve out the inside of the house by filling it with air. Note how you have used the simple maths from your design to work out the coordinates of the air inside the house, as relative coordinates to the outer corners of the cobblestone of the house:

```
mc.setBlocks(x+1, y, z+1, x+SIZE-2, y+SIZE-1, z+SIZE-2,  
block.AIR.id)
```

- Carve out a space for the doorway, again using the `AIR` block type. You won't use a normal Minecraft door here, because this is a huge house. Instead you create a large doorway that is three blocks high and two blocks wide. Your doorway needs to be in the middle of the front face of the house, so `midx` gives you that middle x coordinate:

```
mc.setBlocks(midx-1, y, z, midx+1, y+3, z, block.AIR.id)
```

- Carve out two windows using the block type `GLASS`. As this is a large house, you will build the windows three blocks from the outer edge of the house and three blocks from the middle point of the front of the house. If you change your `SIZE` constant and run the program again, all of the calculations in the program automatically adjust the positioning of everything and the doorway and windows will be in the right place so it still looks like a house.

```
mc.setBlocks(x+3, y+SIZE-3, z, midx-3, midy+3, z,  
block.GLASS.id)  
mc.setBlocks(midx+3, y+SIZE-3, z, x+SIZE-3, midy+3, z,  
block.GLASS.id)
```

- Add a wooden roof:

```
mc.setBlocks(x, y+SIZE-1, z, x+SIZE, y+SIZE-1, z+SIZE,  
block.WOOD.id)
```

- Now, add a woollen carpet:

```
mc.setBlocks(x+1, y-1, z+1, x+SIZE-2, y-1, z+SIZE-2,  
block.WOOL.id, 14)
```

You'll see there is an extra number at the end of `setBlocks()`. This number sets the colour of the carpet; in this case, it is number 14, which is red. This is explained in the following Digging into the Code sidebar.

Save your program, then move to somewhere in your Minecraft world where there is a bit of space and run your program by choosing Run ➔ Run Module from the menu. You should see your house miraculously materialise in front of your eyes! Walk inside and explore it. Look up at the roof and through the windows, and marvel at how quickly you built this house from scratch! (See Figure 3-6.)

Don't forget that `buildHouse.py` always builds the house relative to your player's position. Move around the Minecraft world and run `buildHouse.py` again to build another house. You can build houses all over your Minecraft world—how cool is that?



FIGURE 3-6 This house was built automatically by a Python program.

DIGGING INTO THE CODE

When you built your carpet, you used an extra number with `setBlocks()`:

```
mc.setBlocks(x1, y1, z1, x2, y2, z2, block.WOOL.id, 14)
```

Let's dig into that and work out what it means.

`WOOL` is a really interesting block type, because it has what is called "extra data". This means that not only can you ask for the block to be wool but you can also use this extra data to change the appearance of the `WOOL` block. There are other

blocks in the Minecraft world that have extra data, too, and each type of block uses it in a different way. For `WOOL`, the “extra data” number selects the colour of the block. This useful feature makes `WOOL` a versatile block to build with, because you can choose any colour from a palette of supported colours, allowing you to make your designs much more colourful and realistic.

At the end of this adventure you will find a table listing the numbers and their associated colours for the `WOOL` block. Appendix B also lists various other block types that have an extra data value that you can change.

If you have seen other Minecraft programs on the Internet, you might be wondering why in this book blocks are always referred to like this: `block.WOOL.id`. Why do you always put the `.id` at the end of the block? And why is it that other programs on the Internet don’t do this? This is because `setBlock()` and `setBlocks()` sometimes don’t work when the “extra data” field is used, unless you put the `.id` at the end of the block. So, to keep everything simple and consistent in this book, we decided always to use `.id` with blocks so that every time you use it, it looks the same. You don’t always have to put the `.id` at the end, but it makes sense to put it there so you don’t have to remember when you have to use it and when you don’t.



Try changing the `SIZE` constant in your program to a bigger number, such as 50, and running your program again. What happens to your house when you do this? Try changing `SIZE` to a small number like 10. What happens to your house now? Why do you think this happens when the number stored in the `SIZE` constant is very small?



Building More than One House

Building one house is fun, but why stop there? Thinking back to your `tower.py` program from earlier in this adventure, it’s easy to write a `for` loop that repeats program statements a fixed number of times. So, it must therefore be possible to build a whole street of houses, or even a whole town, just by looping through your house-building program many times.

Before you do this, though, you’re going to improve your house-building program a little, to make sure it doesn’t get too big and complex to manage.

Using Python Functions

One of the things you might want to do later is build a whole town with houses of many different designs. The program for a whole town could become quite big and complex, but fortunately there is a feature inside the Python programming language that helps you package up that complexity into little chunks of reusable program code. It is called a **function**.



With a Python **function** you can group related Python program statements and give them a name. Whenever you want to run those program statements as a group, you just use the name of the function with brackets after it.



You might not realise it, but you have been using functions all the way through this book, ever since you posted a message to the Minecraft chat with `mc.postToChat("hello")` in the very first adventure. `postToChat()` is a function, which is just a group of related program statements that get run whenever you use the word `postToChat()` inside your program.

Before you change your working `buildHouse.py` program to use a function for the house, try out some functions at the Python Shell to make sure you understand the idea:

1. Click on the Python Shell window to bring it to the front.
2. Type the following into the Shell window, which will define a new function called `myname`:

```
def myname () :
```

The `def` means “define a new function and here is its name”. Just like earlier with the `while`, `if` and `for` statements, you must put a colon (`:`) at the end of the line so that Python knows to expect you to provide other program statements as part of the body of this function.

3. The Python Shell automatically indents the next line for you, so that Python knows they are part of the function. Type in a few lines of `print` statements that print your name and information about yourself. Don’t be surprised that nothing happens as you type in each line; that is correct. Be patient, all will become clear in a moment!

```
print("my name is David")
print("I am a computer programmer")
print("I love Minecraft programming")
```

- Finally, press the Enter key on the keyboard twice, and the Python Shell recognises that you have finished typing in indented statements.
- Now ask the Python Shell to run this function by typing in its name with brackets after it:

```
myname()
```

- Try typing `myname()` a few more times and see what happens.

At first it might seem a little strange that you typed instructions into the Python Shell but nothing happened. Normally when you type at the Python Shell, things happen as soon as you press the Enter key, so why didn't it work this time? This time, you did something different, however: you "defined" a new function called `myname` and asked Python to remember the three `print` statements as belonging to that function. Python has stored those `print` statements in the computer memory, rather than running them straight away. Now, whenever you type `myname()` it runs those stored statements, and you get your three lines of text printed on the screen.

Functions are very powerful, and allow you to associate any number of Python program statements to a simple name, a little bit like a mini-program. Whenever you want those program statements to run, you just type the name of the function.

Let's put this to good use by defining a function that draws your house. Then, whenever you want a house made from cobblestone, all you have to do is type `House()` and it is built automatically for you!

- So that you don't break your already working `buildHouse.py`, choose File→Save As from the editor menu, and call the new file `buildHouse2.py`.
- At the top of the program after the `import` statements, define a new function called `house`:

```
def house():
```

- Now move the `midx`, `midy`, and all of your `setBlocks()` statements so that they are indented under the `def house():` line. Here is what your program should now look like. Be careful to get the indents correct:

```
import mcpi.minecraft as minecraft
import mcpi.block as block

mc = minecraft.Minecraft.create()

SIZE = 20

def house():
    midx = x + SIZE/2
    midy = y + SIZE/2
```

```

mc.setBlocks(x, y, z, x+SIZE, y+SIZE, z+SIZE,
block.COBBLESTONE.id)
mc.setBlocks(x+1, y+1, z+1, x+SIZE-2, y+SIZE-2,
z+SIZE-2, block.AIR.id)
mc.setBlocks(x+3, y+SIZE-3, z, midx-3, midy+3, z,
block.GLASS.id)
mc.setBlocks(midx+3, y+SIZE-3, z, x+SIZE-3, midy-3, z,
block.GLASS.id)
mc.setBlocks(x, y+SIZE, z, x+SIZE, y+SIZE, z+SIZE,
block.SLATE.id)
mc.setBlocks(x+1, y+1, z+1, x+SIZE-1, y+1, z+SIZE-1,
block.WOOL.id, 7)

pos = mc.player.getTilePos()
x = pos.x
y = pos.y
z = pos.z

house()

```

- Notice how, in the last statement of your program, you have just put the name `house()`. This line will run the code that is now stored in the computer's memory, which was set up by the `def house():` statement.
- Save your program, move to a new location and run your program. You should see a house get built in front of you.



You might be thinking at this point, "so what?". You changed your program by moving some program statements around and it just does exactly the same thing! Sometimes, when you are writing computer programs, it is necessary to improve the layout or structure of the program first, before you can do more amazing things with it afterwards. That is just what you have done here. You have restructured your program slightly, so that it is now easier to reuse your house-building code many times.

CHALLENGE



The `house()` function will always build the house relative to the coordinates stored in `x`, `y` and `z`. Add some extra statements at the end of your program that change the `x`, `y` and `z` variables and then insert another `house()`, and see what happens. How many houses do you think you could build like this?

DIGGING INTO THE CODE

In your new program, by putting all of your `setBlocks()` statements into the `function house()`, there is an extra little bit of magic that is taking place.

The variables `x`, `y`, `z`, and the constant `SIZE` are all called **global** variables. They are global because they are first given a value inside the main program (the non-indented part). Because they are global, it means they can be used anywhere in the program, including inside the `house()` function. So, if you did some experiments and drew a few different houses, by changing the value of `x`, `y` and `z` in the main program, because these variables are global and can be used anywhere in the whole program, this works.

In Adventure 6 you will learn that global variables can make a large program very hard to fix when it goes wrong, and there is a better way to share information with functions. However, for now, this use of global variables is good enough and your program is small enough that it won't cause a problem.

A **global** variable is a variable that can be used anywhere in the program. Any variables (and also constants) defined without a left-hand indent are global, and can be accessed from anywhere in the program. So, if you use `a = 1` and it has no left-hand indent, that variable can be used anywhere in that program.



However, any variables that are defined with some left-hand indent are not global (often called local variables). So, if you use `a = 1` inside the indented region under a `def` statement (that is *inside* the function), then that variable cannot be accessed from code anywhere except from within the indented region of that function. There is more to learn about global variables, but that's all you need to know about them at this point.

Building a Street of Houses with a for Loop

You are now ready to put together all the things you have learned in this chapter, and build a huge street of houses. If you were building all of these houses manually by choosing items from the inventory, it might take you hours to build them, and you might make mistakes in building them or make some of them slightly smaller or larger by accident. By automating the building of houses and other structures with Minecraft programming, you can speed things up and make sure that they are perfectly built to a very precise size!

To build lots of houses, you need to add a `for` loop to your program as follows:

1. So that you don't break your existing `buildHouse2.py` program, use File ➔ Save As from the menu and save a new file called `buildStreet.py`.
2. At the end of the program, add a `for` loop above the final `house()` along with a line that changes the `x` position that the house is built at. Each house is built `SIZE` blocks away from the previous house. The new lines are marked as bold:

```
for h in range(5):  
    house()  
    x = x + SIZE
```

Save your program, then move to a place in the Minecraft world where there is a bit of space, and run the program. As shown in Figure 3-7, you should get a huge street of five houses as far as the eye can see! Walk your player into each of the houses and make sure that they have been built properly.



FIGURE 3-7: A street of five identical houses built automatically with a Python program

CHALLENGE



How could you modify your loop so that instead of building your houses in a row, it builds them upwards to create a huge tower block? Try it. Turn your tower block builder into a function called `tower()` and then use a loop to build a few tower blocks inside your Minecraft world.

Depending on where your player is standing when you build your houses, and how much other terrain is around, like trees and mountains, you might get some interesting effects on some of your houses. One of them might chop off half of a tree, or be partly built into a mountain. You might even get some interesting subsidence if your houses end up being built on top of the sea! You might like to modify your `house()` function to build a layer of bedrock under the house so that it is always on solid ground.



Adding Random Carpets

At this point, you should have a huge number of houses inside your Minecraft world and things will be looking pretty awesome. From a small number of lines of Python code, you've built some large engineering structures already!

However, you're probably thinking that having a street of identical houses is starting to look a little boring. How can you make the houses slightly different to add interest to your street?

One way to do this is to write a few different `house()` functions like `cottage()`, `townHouse()` and even `maisonette()`, and modify your program to use these different functions at different places to add some variety to your street designs.

Another way to make your structures more interesting is to slightly change part of them, such as the carpets, in a way that is different every time you run the program. This way even you, the programmer, won't know quite what you have created until you explore all the houses!

Generating Random Numbers

Computers are very precise machines. In many aspects of everyday life, we all rely on computers to be predictable and to do the same thing every time a program is run. When you pay £10 into your bank account, you want to make sure that exactly £10 makes it into the part of the computer's memory that holds your balance, every time, without fail. So the concept of randomness might seem quite unusual to such a precise system.

However, one area where randomness is really important is in game design. If games did everything exactly the same every time, they would be too easy to play—not fun or challenging at all. Almost every computer game you play has some kind of randomness in it to make things slightly different each time and hold your interest.

Fortunately for your Python programming, the Python language has a built-in module that will generate **random numbers** for you so you don't have to write the code for this yourself, just use this built-in random number generator instead.

To make sure you know what these random numbers look like, try this out at the Python Shell:

1. Click on the Python Shell window to bring it to the front.
2. Import the random module so that you can use the built-in random function, by typing:

```
import random
```

3. Ask the program to generate a random number between 1 and 100 and print it to the screen:

```
print(random.randint(1,100))
```

You should see a number between 1 and 100 appear on the screen. Type the `print` statement again. What number do you get this time?

4. Now use a `for` loop to print lots of random numbers. Make sure you indent the second line so that Python knows that the `print` statement is part of the `for` loop:

```
for n in range(50):
    print(random.randint(1,100))
```

The two numbers inside the brackets of the `randint()` function tell it the range of numbers you want it to generate; 1 is the smallest number you should expect it to generate, and 100 the largest.



A **random number** is usually generated from a random number sequence—a list of numbers designed not to have any obvious pattern or repeating sequence.

Computers are very precise machines and often do not generate truly random numbers; instead, they generate pseudo-random numbers. These numbers might seem to be part of a random sequence, but there is a pattern to them. You can read more about random numbers at http://en.wikipedia.org/wiki/Random_number_generation and find out more about real random numbers at www.random.org.

Laying the Carpets

Earlier in this adventure you used an extra number in the `setBlocks()` function to make the colour of the woollen carpet red, using the extra data of `WOOL`. The allowed range of extra data numbers for `WOOL` is between 0 and 15—in other words, there are 16 colours you can choose from. Now use the following steps to change your house-building program to generate a random number and use that as the colour of `WOOL` for the carpet in the house:

1. Save your `buildStreet.py` with File ➔ Save As from the menu, and call it `buildStreet2.py`.
2. Add this import statement at the top of the program to gain access to the random number generator:

```
import random
```

3. Change your `house()` function so that it generates a random carpet colour each time it is used. As `WOOL` can have an extra data value ranging between 0 and 15, that is the range of random numbers you need to generate. Store the random number in a variable called `c` so that the carpet-building line doesn't become too long and hard to read, and make sure that both of these lines are indented correctly, as they are both part of the `house()` function:

```
c = random.randint(0, 15)
mc.setBlocks(x+1, y+1, z+1, x+SIZE-1, y+1, z+SIZE-1,
    block.WOOL.id, c)
```

4. Save your program.

Now you need to find somewhere in your Minecraft world that has plenty of space to build more houses! Move around the world and find somewhere to build, then run your new program. It should generate another street of houses but this time, when your player explores inside the houses, the carpet in each house should be a different, random colour. See Figure 3-8 to see what these new houses look like, but your player will need to actually go into each house and look at the carpets so that you can check they are indeed all different!



FIGURE 3-8 Each house has a random carpet colour.

CHALLENGE



Define a `house2()` function that draws a different type of house. Use randomness in the main `for` loop to create one of two houses based on a random number. Every house you build will be random. Extend this program to build three different types of house. The more different types of house you build, the more interesting your street will become. You could even experiment with negative coordinates like `y=-5` to build a basement or even a swimming pool inside your house!



To a large extent, the sort of huge structures you can build in Minecraft is limited only by your imagination. There is a limit to the height of the Minecraft world, of course, and that will restrict the height of the tallest structure you can build. And some buildings in real life have slanting or curved structures. For example, look at the design of The Shard, currently the tallest building in the European Union at <http://the-shard.com>. This amazing building would be possible to build in Minecraft, but because it has slanted edges it will be more difficult.

Be patient and start off with simple structures that are mainly rectangular in shape. In Adventure 7, Martin will introduce you to some extra special helper functions that are capable of building angled lines and curves. After that, there will be no limit to the type of buildings you can build in your Minecraft world!

There are two other restrictions on how big you can build things in Minecraft—how much memory the computer has, and how far you can see in front of you. A larger structure takes up more computer memory; the size of the Minecraft world has edges to it, so that the amount of memory required to store it is practical. Secondly, your player can only see a certain distance, especially so on the Raspberry Pi, which has a very limited viewing distance.

Quick Reference Table

Importing and using the block name constants	Setting/changing a block at a position
<pre>import mcpi.block as block b = block.DIRT.id</pre>	<code>mc.setBlock(5, 3, 2, block.DIRT.id)</code>
Setting/changing lots of blocks in one go	Useful block types for houses
<code>mc.setBlocks(0,0,0,5,5,block.DIRT.id)</code>	<code>block.AIR.id</code> <code>block.STONE.id</code> <code>block.COBBLESTONE.id</code> <code>block.GLASS.id</code> <code>block.WOOD.id</code> <code>block.WOOL.id</code> <code>block.SLATE.id</code>

All of the new Python and Minecraft statements you learned in this chapter are listed in the reference section in Appendix B.

You can find a complete list of block id numbers on the Minecraft wiki: <http://minecraft.gamepedia.com/Blocks>. Minecraft extra data values are taken from: http://minecraft.gamepedia.com/Data_values.

`Wool` is a very useful block type to build with, as you discovered when laying the random carpets in your street of houses. Here is a reference to the different colours that can be used with the `block.WOOL.id` block type. Look back at your `buildStreet2.py` program to see how to provide this extra data to the Minecraft API.

0 white	1 orange	2 magenta	3 light blue
4 yellow	5 lime	6 pink	7 grey
8 light grey	9 cyan	10 purple	11 blue
12 brown	13 green	14 red	15 black

Further Adventures in Building Anything

In this adventure, you've learned how to create single blocks and whole areas of blocks inside the Minecraft world using a single line of a Python program. You've built some pretty impressive structures already. You've learned that with functions you can split up your program into smaller logical units and, with `for` loops, repeat things over and over again. With this knowledge you should be able to build almost any structure you could possibly imagine!

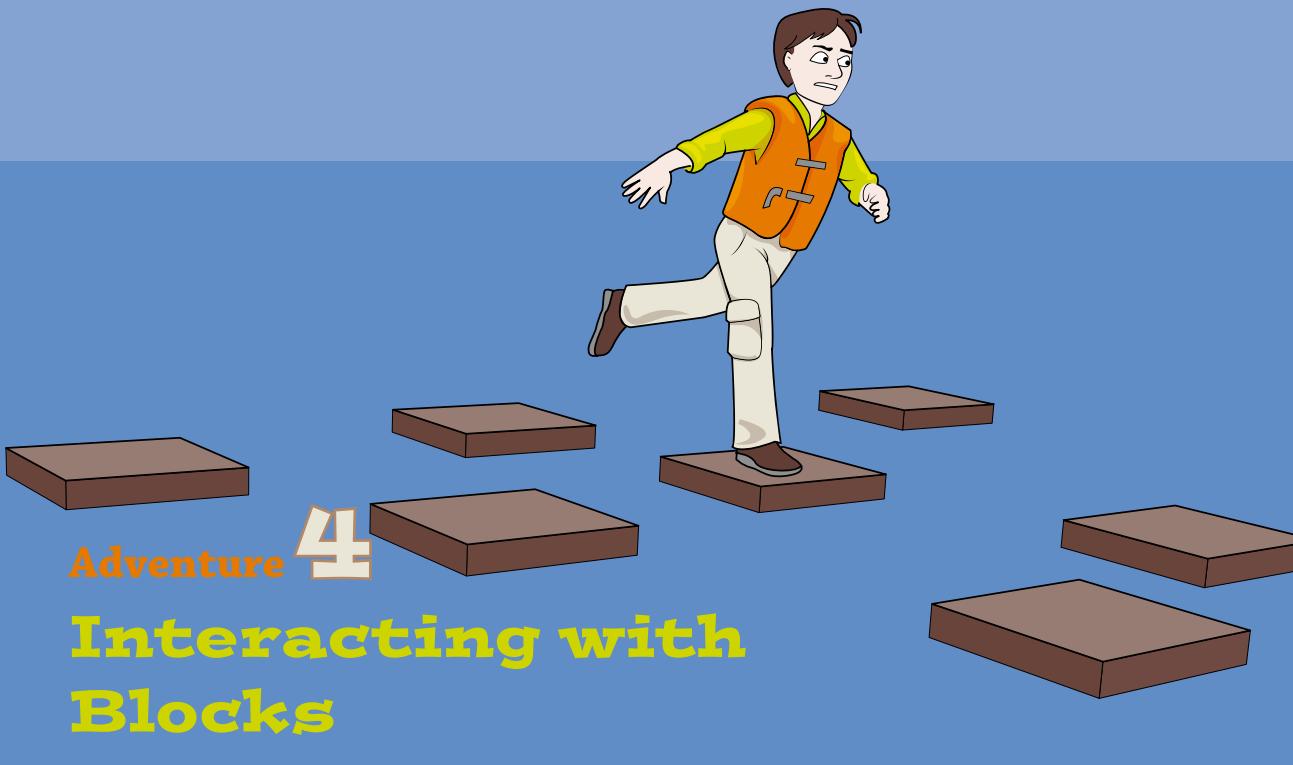
- Using the techniques learned in this adventure, write one function for each of the six faces of a dice. Write a loop that spins round a random number of times, showing a different dice face each time round. Use `random.randint()` to stop on a random pattern, and challenge yourself and your friends to try and guess the number the Minecraft dice will stop on.
- Make a list of other things you could randomise about your street. Perhaps do some research by walking down your own street and seeing how the houses differ from each other. Define more house functions, one for each type, and try to build more complex streets with houses in lots of different styles.
- Get together with some friends and join all your different house-building programs into one big program. Use it to build a whole community of houses of different styles inside your world. Marvel, as you walk around your new town, at how much variety there is in house design!



Achievement Unlocked: Designer of remarkable buildings and builder of amazing huge structures inside Minecraft!

In the Next Adventure...

In Adventure 4, you learn how to sense and interact with blocks, including how to detect what type of block you are standing on and how to detect (and what to do) when blocks have been hit by the player. You'll use all this new knowledge to build an exciting treasure hunt game!



Adventure 4

Interacting with Blocks

ONE WAY YOU can make a Minecraft game more interesting is to make it change what it does based on what is going on around your player. As you move around the game world, the choices that you are faced with depend on what you have already done before, making the game slightly different every time you play it. The Minecraft API allows you to interact with blocks by finding out what block type you are standing on and detecting when you hit a block with your sword.

In this adventure, you will first learn the basics of interacting with blocks by writing a magic bridge program. This bridge is special, because as you walk on water or walk into the sky, a bridge magically appears in front of you to keep you safe. Soon your Minecraft world will fill up with bridges. Version 2 of your magic bridge builder will use a Python list to remember where you built the bridge and will make it do a disappearing act right before your eyes when you land on safe ground again!

Finally, you learn how to sense that a block has been hit, and then build an exciting treasure hunt game using your magic bridge to find and collect treasure that appears randomly in the sky, complete with a homing beacon and a score.

In this adventure you will work just like a real software engineer, building up a large program one function at a time, and finally stitching it all together at the end to make an exciting larger program. Fasten your seatbelt and take careful note of the instructions. It's going to be an exciting journey into the sky!

Finding Out What You Are Standing On

You learned in Adventure 2 that it is possible to track your player's position by reading the coordinates with `getTilePos()`. These coordinates represent the x, y and z coordinates in the Minecraft world where your player, Steve, is located at the moment. You used these coordinates to sense whether Steve was standing on a magic doormat or, using geo-fencing, whether he was standing in a field.

However, unless your programs maintain a detailed map of exactly where every block is in the Minecraft world, just sensing by position is not going to be flexible enough for you, as your programs become more sophisticated. You could always keep a detailed map of your own—but why go to all that trouble when Minecraft must already have that information in the computer's memory to display the 3D world on the screen?

Fortunately, the Minecraft API also includes a `getBlock()` function. This function gives you full access to the in-memory world map of Minecraft and, by using coordinates, you can use it to tell you about every block—not just the block where Steve is located but every block at every position in the Minecraft world.

You also saw in Adventure 3 that it is possible, through block types, to change any block in the Minecraft world. Fortunately, `setBlock()` and `getBlock()` work together as a pair, so if you use `setBlock()` with a block id, and then use `getBlock()` immediately after that, you get the same block id back.

Soon you are going to build another exciting game inside Minecraft, but your program will be quite big. The best way to build a big program is to build little programs and then stick them all together once you know they work. Let's start by bringing this idea to life with a simple program that tells you if your player is standing on something safe or not.

Finding out if Your Feet Are on the Ground

Start up Minecraft, IDLE, and if you are working on a PC or a Mac, start up the server too. You should have had a bit of practice with starting everything up by now, but refer back to Adventure 1 if you need any reminders. You are now going to build a program that gives you important information about your player's exact location. You need to find out if his feet are on the ground before you can build your magic bridge into the sky.

1. Create a new window by clicking File→New File. Save your new program as `safeFeet.py`. Remember to store your programs inside the `MyAdventures` folder, otherwise they will not work.

2. Import the necessary modules. You will need the normal `Minecraft` module and, because you are interacting with blocks, also the `block` module:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import time
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. Because you plan to use parts of this program in a bigger program later, you are going to write the bulk of the code inside a function called `safeFeet()`. This will make it easier for you to reuse this code later. The first thing this function does is to get the position of your player, Steve:

```
def safeFeet():
    pos = mc.player.getTilePos()
```

5. `getBlock()` will get the block id of the block at the coordinates you provide. Because `pos.x`, `pos.y` and `pos.z` are the coordinates of your player, you must use `pos.y-1` to get the block directly below Steve's feet:

```
b = mc.getBlock(pos.x, pos.y-1, pos.z)
# note: pos.y-1 is important!
```

6. You now use a simple method to work out whether your player is safe or not. If he is standing on air or water, then he is not safe. Otherwise, he is safe. It is simpler to check the block types for the “not safe” condition, as there are a few hundred blocks you would need to check for the “safe” condition. This is quite a long piece of code, so make sure you type it in all on one line:

```
if b == block.AIR.id or b == block.WATER_STATIONARY.id ↵
or b == block.WATER_FLOWING.id:
```

7. If the block is one of the unsafe blocks, post a message to the Minecraft chat saying that your player is not safe. Otherwise, he is safe.

Make sure you get the indents correct here, otherwise the program will not work. Remember that all of the code inside the function is indented one level, and any code inside an `if` or an `else` is indented by another level. The `if` and the `else` statements should line up with each other as they are related and logically at the same level.



```
mc.postToChat("not safe")
else:
    mc.postToChat("safe")
```

- This is now the end of the `safeFeet()` function. Leave a blank line to remind yourself that it is the end of the function, and start the `while` of the game loop without any indent. As in your earlier programs, you put a short delay and, finally, use the new `safeFeet()` function which will do all the work.

```
while True:  
    time.sleep(0.5)  
    safeFeet()
```

Now click File ➔ Save to save your program, and then run it by clicking Run ➔ Run Module from the editor menu.

What happens as your player moves around the Minecraft world? You should see the words “safe” or “not safe” appear on the Minecraft chat as appropriate, as shown in Figure 4-1. Try flying in the air and swimming in the sea to see what happens.

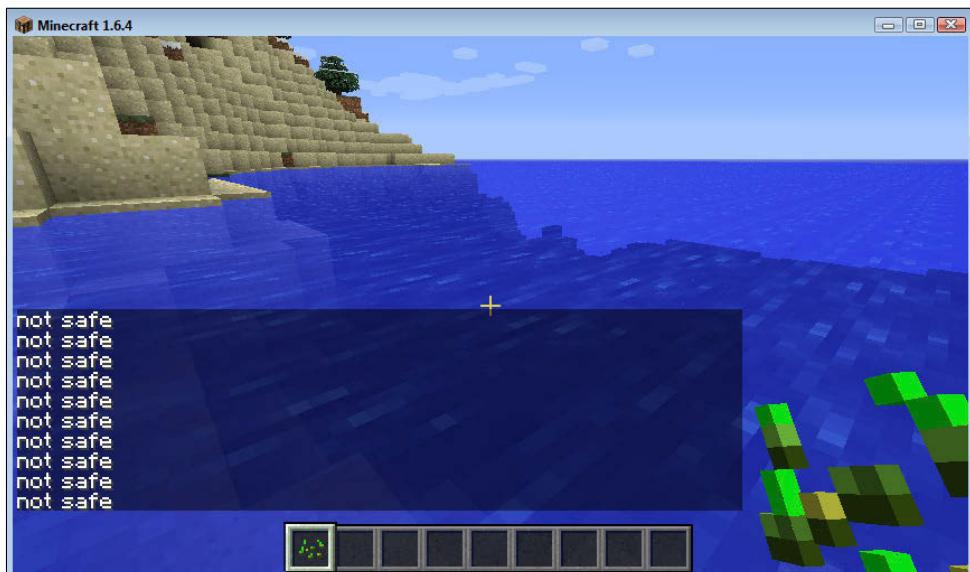


FIGURE 4-1 The safeFeet program shows you are not safe while you are in the sky.



In your `safeFeet.py` program, just like other programs, you used a short time delay in your main game loop. This is not always necessary, and later you will work on programs that are slowed down so much by the time delay that they don't work properly. When posting messages to the Minecraft chat, however, it is actually useful to slow things down a bit; otherwise the chat fills up very quickly. Try shortening the delay or removing it altogether to see what happens.

Building Magic Bridges

In the previous program, you wrote code that sensed the block directly under your player and posted a message on the chat. This is an interesting tiny step towards a bigger program but let's see if you can now turn this experiment into something real, by stitching it together with some of the techniques you learned in Adventure 3 with the `setBlock()` function.

By making a small change to `safeFeet.py`, you can turn it into a magic bridge builder that places a glass bridge under your feet wherever you walk, making sure that your player never falls into the sea or falls out of the sky! You will reuse this new function in a later program in this adventure, so make sure you name it correctly.

1. Click File→Save As and rename your `safeFeet.py` program as `magicBridge.py`.
2. Change the name of the `safeFeet()` function so that it is now called `buildBridge()`, and modify the `if/else` statement as marked in bold by removing the `mc.postToChat()` and replacing it with a `mc.setBlock()`. Every time your player's feet are unsafe, this builds a glass bridge under him. Be careful of the long line in the `if` statement:

```
def buildBridge():
    pos = mc.player.getTilePos()
    b = mc.getBlock(pos.x, pos.y-1, pos.z)
    if b == block.AIR.id or b == block.WATER_FLOWING.id ↵
    or b==block.WATER_STATIONARY.id:
        mc.setBlock(pos.x, pos.y-1, pos.z, block.GLASS.id)
```

3. As you can see in step 2, the `else` and the `mc.postToChat()` have been removed from the code, as they are no longer needed.
4. If you build the bridge too slowly, your player will fall off, so you now need to remove the time delay from the game loop. Make sure you use the new `buildBridge()` function:

```
while True:
    buildBridge()
```

5. Save your program by clicking File→Save from the editor menu.

Run your program, and walk around the world, jumping up into the sky and walking on water. As your player walks around, whenever his feet are not on safe ground, a magic glass bridge appears to keep him from falling, as in Figure 4-2. It is now possible for him to walk on water. It's a miracle!



This program works very nicely on the Raspberry Pi. On the PC and on the Mac, the response time isn't quite as quick, and as a result of this the player may keep falling because the blocks aren't placed quickly enough. Take your time, and don't start running around too fast expecting the bridge to keep you safe all the time! You can experiment with faster and slower delays to improve this game, or you can try creep mode (hold down the Shift key on the keyboard while moving) to walk forward slowly.

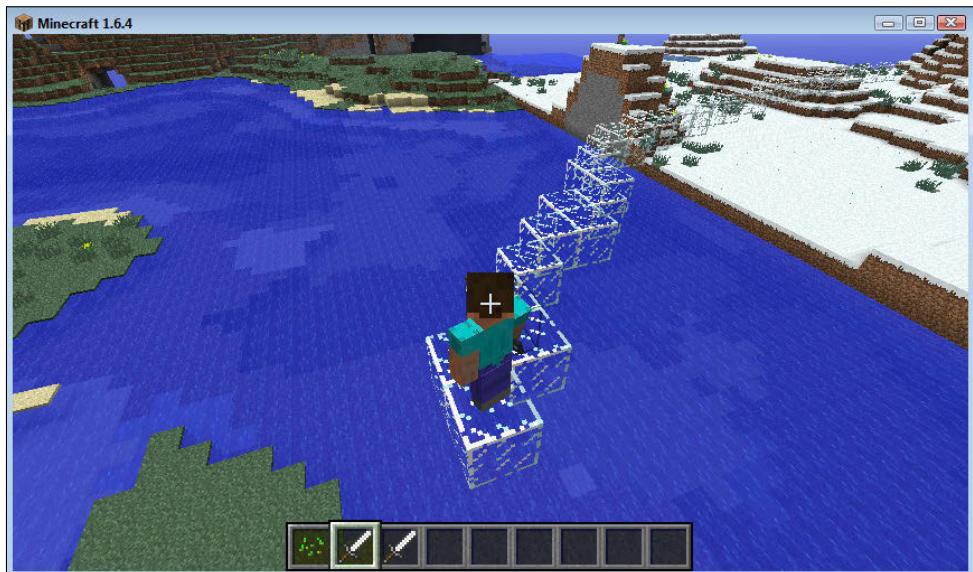


FIGURE 4-2 The `magicBridge.py` program builds a glass bridge, allowing your player to walk on water.



In the `magicBridge.py` program, you removed the time delay from the main loop. What happens if you put this time delay back in? Experiment with different delay values somewhere between the range of 0.1 to 2 seconds, and see how this affects the usability of your magic bridge. What value provides the best usability?

DIGGING INTO THE CODE

Let's just pause for a minute, as there is a little bit of magic to do with functions that hasn't quite been explained properly yet.

You have been using functions since the start of this book, `mc.postMessage()` is a function, as is `mc.getTilePos()`, they are both part of the Minecraft API. You have also defined your own functions with `def`, such as in Adventure 2 you designed a `house()` function of your own. But what really is happening when you use these functions in your code?

I've been cheating a little up until now and saying "now use this function," but the proper way to talk about what is happening here is to say that when you put `house()` or `buildBridge()` in your code, you can say that you **call** your function. But what really happens?

All the time your Python program is running, the computer remembers which statement it is working on at any time, a bit like an invisible finger pointing to the line in the code that is running. Normally, this invisible finger moves down the page from top to bottom. When you use a loop, this invisible finger jumps back to the top of the loop and runs down the code again a number of times.

When you call a function, some extra magic is happening behind the scenes. Python remembers where this invisible finger was (imagine that it sticks a little coloured tab at that point in the program) and then jumps into your function, such as your `buildBridge()` function. When it gets to the end of the `buildBridge()` function, it jumps back to where it left that little coloured sticky note and continues from there again.

Using functions in your programs is useful for many reasons, two of the most important ones being:

- You can split a large program up into lots of smaller programs.
- You can re-use the code inside a function from lots of different places in the same program.

Both of these reasons will make your programs easier to read and easier to modify.

When you **call** a function, Python will remember where it has got to in your program and temporarily jump into the function at the point you defined it with `def`. When the end of the function is reached, Python jumps back to just after where it was when it jumped into that function.



Using Python Lists as Magic Memory

In all of the programs you have written up until now, you have used some form of variable to store information that changes as the program runs. Each of these variables has a name and a value, and if you want your programs to remember more than one thing, you use more variables.

However, these variables are a fixed size and each can store only one thing (e.g. a number or a string of text). Many programs you will write as a Minecraft programmer will need you to store an undefined number of items. You won't always know how many items you will want to store in a program.

Fortunately, like any modern programming language, Python has a feature called a **list**, which can store varying amounts of data as the program runs.



A **list** is a type of variable in a programming language that can store any number of items of data. You can add new items to the list, count the length of the list, access items at particular positions, remove items from anywhere in the list and do many other things. Lists are a very useful way of storing collections of similar items that belong together, such as a list of numbers that need sorting, a list of members in a user group or even a list of blocks in a Minecraft game.

Experimenting with Lists

The best way to understand lists is to experiment with them in the Python Shell.



Make sure you are careful with the brackets in this section. There are two types of brackets in use. The round brackets `()` and the square brackets `[]`. Don't worry. The difference between these two types of brackets will become clear as you work through this section.

1. Bring the Python Shell window to the front by clicking on it. Click the mouse to the right of the last `>>>` prompt, which is where you will start typing your interactive commands. Don't forget that you must stop your existing program running by choosing `Shell` \rightarrow `Restart Shell` from the Python Shell menu, or pressing `CTRL` then `C`. If your previous program is still running, this next section will not work!

2. Create a new empty list, and show what is in the list:

```
a = [] # an empty list  
print(a)
```

3. Add a new item to the list, and print it again:

```
a.append("hello")  
print(a)
```

4. Add a second item to the list, and print it again:

```
a.append("minecraft")  
print(a)
```

5. Check how long the list is:

```
print(len(a))
```

6. Look at items at specific positions within the list. These positions are called **indexes**:

```
print(a[0]) # the [0] is the index of the first item  
print(a[1]) # the [1] is the index of the second item
```

7. Remove one word from the end of the list, and show that word and the remaining list:

```
word = a.pop()  
print(word)  
print(a)
```

8. Check how long the list is, and check how long the word string is:

```
print(len(a))  
print(len(word))
```

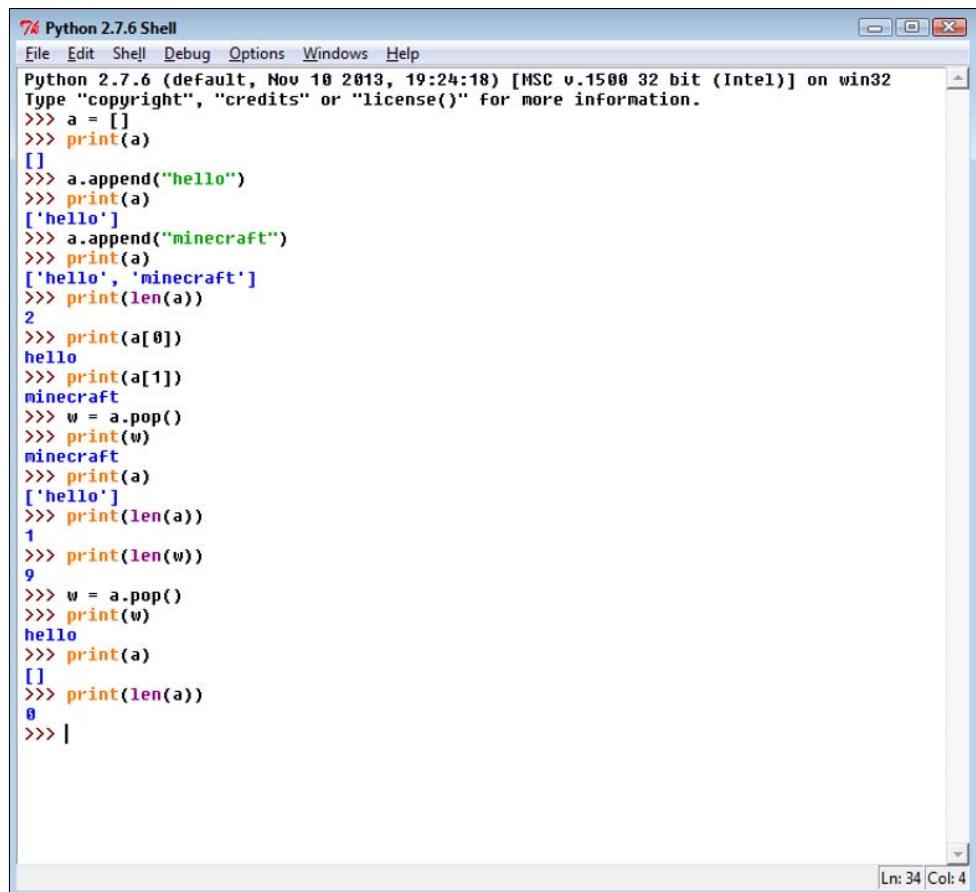
9. Remove the final item from the list, show the item and the list, and show the length of the list:

```
word = a.pop()  
print(word)  
print(a)  
print(len(a))
```

Figure 4-3 shows the output of these experiments in the Python Shell.

An **index** is a number that identifies which item in a list of items to access. You specify the index in square brackets like `a[0]` for the first item and `a[1]` for the second item. Indexes in Python are always numbered from 0, so 0 is always the first item in a list. In the steps just completed, you *indexed* into the list.





```
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a = []
>>> print(a)
[]
>>> a.append("hello")
>>> print(a)
['hello']
>>> a.append("minecraft")
>>> print(a)
['hello', 'minecraft']
>>> print(len(a))
2
>>> print(a[0])
hello
>>> print(a[1])
minecraft
>>> w = a.pop()
>>> print(w)
minecraft
>>> print(a)
['hello']
>>> print(len(a))
1
>>> print(len(w))
9
>>> w = a.pop()
>>> print(w)
hello
>>> print(a)
[]
>>> print(len(a))
0
>>> |
```

FIGURE 4-3 Experimenting with Python lists in the Python Shell



The length of a list will change as you add new items to it and remove items from it. What happens if you try to access an item in the list that does not exist? Try this at the Python Shell and see what happens:

```
b = []
print(b[26])
```

How do you think you could prevent this from happening in your programs? (Note there is more than one right answer to this question. You could do some research on the Internet to find out the different ways you can prevent this problem in your programs.)

A Python list is a versatile type of variable, because it can be used to store any type of item. It is not the only variable type in Python that can store multiple items. At this stage, the most important point for you to note about a list is that you don't need to know in advance how big the list is before you run the program. The list will grow and shrink automatically as you `append()` items to it and `pop()` items off the end.



When you `pop` an item from a list, you remove the last item from that list.



Building Vanishing Bridges with a Python List

You are now going to use your new knowledge about lists to write a bridge builder program, where the bridge will vanish once your player's feet are safely on the ground again. This program is similar to the `magicBridge.py` program, so you could save that as a new name and edit it, but the full program is shown here to make it easier to explain each of the steps. Use copy and paste from your `magicBridge.py` program if you want to save a little bit of typing.

To see a tutorial on how to build and play the vanishing bridge game, visit the companion website at www.wiley.com/go/adventuresinminecraft and choose the Adventure 4 video.



1. Create a new file by clicking `File`→`New File`, and save it as `vanishingBridge.py` by choosing `File`→`Save As` from the editor menu.
2. Import the necessary modules:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block  
import time
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. Create a bridge list that is empty. There are no blocks in your bridge at the moment, so it always starts empty:

```
bridge = []
```

5. Define a `buildBridge()` function that will build the bridge for you. You will use this `buildBridge()` function in the final program in this adventure, so make sure you name this function correctly. Most of the code at the start of this function is the same as in the `magicBridge.py` program, so you could copy that if you want to save some typing time. Be careful to get the indents correct:

```
def buildBridge():
    pos = mc.player.getTilePos()
    b = mc.getBlock(pos.x, pos.y-1, pos.z)
    if b == block.AIR.id or b == block.WATER_FLOWING.id ↵
    or b == block.WATER_STATIONARY.id:
        mc.setBlock(pos.x, pos.y-1, pos.z, block.GLASS.id)
```

6. When you build part of the bridge, you have to remember where you built the block, so that the block can be removed later on when your player's feet are safely on the ground. You will use another list to keep the three parts of the coordinate together, and add this to the bridge list. See the following Digging into the Code sidebar for a fuller explanation of what is going on here:

```
coordinate = [pos.x, pos.y-1, pos.z]
bridge.append(coordinate)
```

7. To make your bridge vanish when your player is no longer standing on it, you need an `else` statement to check if he is standing on glass or not. If he is not, your program will start deleting blocks from the bridge. The program has to check that there is still some bridge left, otherwise it will raise an error if you try to pop from an empty list. The `elif` is short for “`else if`”, and the `!=` means “not equal to”. Be careful with the indents here: the `elif` is indented once, as it is part of the `buildBridge()` function; the next `if` is indented twice as it is part of the `elif`:

```
elif b != block.GLASS.id:
    if len(bridge) > 0:
```

8. These next lines are indented three levels, because they are part of the `if` that is part of the `elif` that is part of the `buildBridge()` function! Phew!

Remember that earlier you appended a list of three coordinates to the bridge list? Here, you have to index into that list with `coordinate[0]` for x, `coordinate[1]` for y and `coordinate[2]` for z. Adding the `time.sleep()` also makes the bridge vanish slowly, so that you can see it happening:

```
coordinate = bridge.pop()
mc.setBlock(coordinate[0], coordinate[1],
            coordinate[2], block.AIR.id)
time.sleep(0.25)
```

9. Finally, write the main game loop. As in your earlier experiments, you might like to try different delay times in the game loop to improve the usability of the bridge builder. Remember that this game loop is part of the main program (the `while True:` is not indented at all), so check the indentation very carefully:

```
while True:  
    time.sleep(0.25)  
    buildBridge()
```

Save your program with File ➔ Save and then run it with Run ➔ Run Module. Walk your player around the Minecraft world and then walk him off a ledge or into a lake, then turn and walk him back onto safe ground again. What happens? Figure 4-4 shows the bridge starting to vanish once Steve is on safe ground.

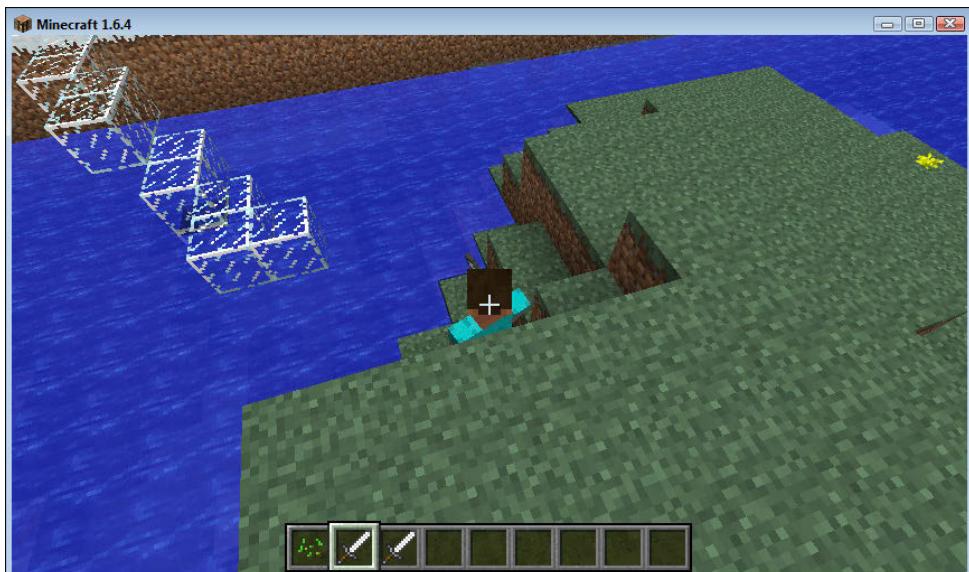


FIGURE 4-4 The bridge magically vanishes once your player is safely on the ground.

DIGGING INTO THE CODE

Earlier, when you experimented with lists at the Python Shell, you added text strings to your list. In the `vanishingBridge.py` program, there is an extra little bit of Python magic that is worth explaining.

Python lists can store any type of data, such as text strings, numbers and even lists. You have used this magic already in your program, perhaps without realising it.

continued

continued

The following line creates a list with three items in it (the x, the y and the z coordinates of a block you have just placed):

```
coordinate = [pos.x, pos.y-1, pos.z]
```

By putting values between the [] brackets, you are creating the list with values already in it, rather than creating an empty list. You could have done the same like this:

```
coordinate = []
coordinate.append(pos.x)
coordinate.append(pos.y-1)
coordinate.append(pos.z)
```

Later, when your program retrieves the coordinates of a block to delete, it pops off a list (of coordinates) from your bridge list:

```
coordinate = bridge.pop()
```

The list that is stored in the coordinate variable has three items inside it: `coordinate[0]` is the x position of the block; `coordinate[1]` is the y position of the block; and `coordinate[2]` is the z position of the block.

Figure 4-5 shows what this “list of lists” actually looks like.

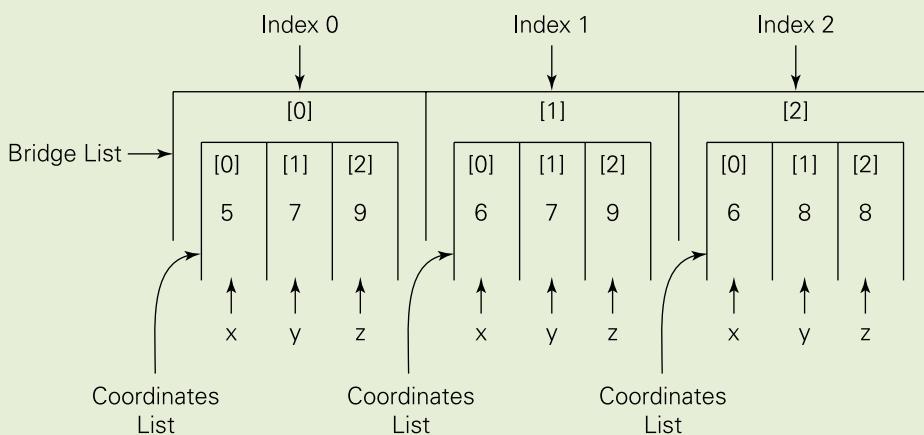


FIGURE 4-5 A bridge is a list of coordinates, and each coordinate is a list.



Python actually has a number of different variable types for storing collections of items, in addition to the lists that you have just used. Professional Python programmers would probably use a feature called a **tuple** in this program for storing the coordinates. I didn't want to introduce too many concepts here in one go. If you like, you can do some research on the Internet about tuples, and find out how they are different from lists and what additional benefits they bring to a Python program.

Sensing that a Block Has Been Hit

The last sensing ability that you need in your tool-bag for this adventure, is the ability to sense when your player hits a block. Block-hit detection will allow you to create some really exciting games and programs of your own, because it allows your player to interact directly with each and every block inside the Minecraft world.

Start a new program for this adventure, as it will begin life as a little self-contained experiment but will later make its way into your final game of this adventure.

1. Create a new file by choosing File→New File from the editor window. Click File→Save As and name the new file `blockHit.py`.
2. Import the necessary modules:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import time
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. Work out the position of the player and move very slightly to one side. Use this as the position of the diamond that you are going to create—this is your magic treasure:

```
diamond_pos = mc.player.getTilePos()
diamond_pos.x = diamond_pos.x + 1
mc.setBlock(diamond_pos.x, diamond_pos.y, diamond_pos.z,
            block.DIAMOND_BLOCK.id)
```

5. Define a function called `checkHit()`. You will reuse this function in your final program, so make sure you name it correctly:

```
def checkHit():
```

6. Ask the Minecraft API for a list of events that have happened. This is just a normal Python list, like the one you used in your `vanishingBridge.py` program earlier:

```
events = mc.events.pollBlockHits()
```

7. Process each event in turn, using a `for` loop. See the following Digging into the Code sidebar for a more detailed explanation of this new form of the `for` loop:

```
for e in events:
    pos = e.pos
```

- Ask your program to check to see if the position of the block the player just hit with his sword is the same as the position of the diamond. If it is, ask it to post a message to the Minecraft chat:

```
if pos.x == diamond_pos.x and pos.y == diamond_pos.y ←  
and pos.z == diamond_pos.z:  
    mc.postToChat("HIT")
```

- Finally, write your game loop. For now, you'll use a time delay of one second, to limit how quickly messages can appear on the Minecraft chat, but you might like to experiment with different time delays to get the best usability from your program. Check your indentation very carefully here:

```
while True:  
    time.sleep(1)  
    checkHit()
```

Save your program with File ➔ Save As and run it using Run ➔ Run Module from the editor window.

Move your player around a bit until you can see the diamond. Now, hit it on each of its faces with a sword. What happens? As Figure 4-6 shows, when you hit the diamond, the message "HIT" appears on the Minecraft chat.

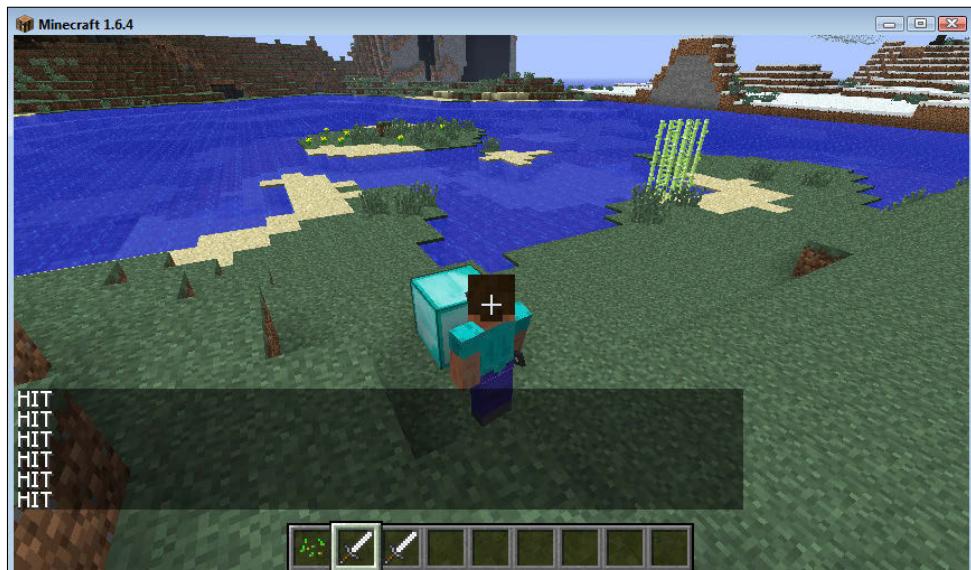


FIGURE 4-6 A block has been hit.

If you accidentally use the wrong mouse button and delete your treasure, you will never be able to hit it! You will either have to re-run the program to create new treasure, or manually build a block of treasure in the same place before you can hit it. This is because the block type **AIR** does not process block-hit events.



CHALLENGE

Modify your `blockHit.py` program so that it also reads the `e.face` variable from the event that it is returned in the `for` loop. This `e.face` variable is a different number depending on which of the block's six faces has been hit. Start by displaying `e.face` on the Minecraft chat and then hitting each face to work out which numbers are used for which faces. Finally, modify your program to display a different message for each of the six faces of your diamond block.



DIGGING INTO THE CODE

In the block-hit detection program, you used a `for` loop, but you used a style of it that you have not used before. Let's look a little bit into this extra magic, as it is quite a powerful feature of Python.

The normal `for` loop you have used before looks like this:

```
for i in range(6):  
    print(i)
```

`range(6)` is actually a function that generates a list of numbers `[0, 1, 2, 3, 4, 5]`. You can prove this by typing the following at the Python Shell:

```
print(range(6))
```

You will see the following:

```
[0, 1, 2, 3, 4, 5]
```

That looks a bit familiar, doesn't it? Actually, all the `range()` function does is to generate a new list of numbers.

The `for/in` statement in Python will loop through all items in a list, storing the first item in the loop control variable (`i` in the previous example), then running the loop body, then storing the next item in the loop control variable, and doing this until the list is exhausted.

continued

continued

This means that if you have a list with anything in it, you can loop through all the items in the list in the same way. Try this at the Python Shell:

```
for name in ["David", "Gail", "Janet", "Peter"]:  
    print("hello " + name)
```

You can also loop through the characters of a text string like this:

```
name = "David"  
for ch in name:  
    print(ch)
```

Writing a Treasure Hunt Game

For most of this adventure, you have been learning skills and building snippets of program code to test out and experiment with various sensing features in Minecraft. It's now time for you to knit all of that code together into a complete game. The game you are going to write is called "Sky Hunt", a treasure hunt in which you have to find diamond blocks hanging randomly in the sky using a homing beacon, and hit them to get points.

There is a twist to this game though: every time you move forward you leave a trail of gold, and this costs you one point off your score per gold block. If you run around aimlessly looking for the treasure, your score will rapidly decrease and even become negative! You will have to use your Minecraft navigation skills to look for the diamond blocks quickly, and try to get to them in as few moves as possible.

When you find each piece of treasure you score points, and the trail of gold magically melts away (possibly leaving holes in the ground for you to trip over, so watch out!).

This program is mostly made up of reusable parts from all the other little experimental programs you have already written in this adventure. You can cut and paste bits of your other programs and modify them to save typing time if you wish. But I have included the full program here to make sure you know what is needed.

Professional software engineers often start with a simple framework program built with just print statements and test this first to make sure the structure is correct, and then add and test new features to it gradually. In this section, you are also going to be a real software engineer, and write and test this program in steps. First, let's get the framework of the game loop in, and some dummy functions that you can flesh out as you go along.

Writing the Functions and the Main Game Loop

1. Start a new file with File ➔ New File from the menu. Save it with File ➔ Save As and call it `skyHunt.py`.
2. Import the necessary modules:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import time
import random
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. Set a score variable that will keep track of your score as the game plays. You will also use a `RANGE` constant to set how difficult the game is, by setting how far away from the player the random treasure is placed. Set this to a small number to start with, while you are testing, and make the number bigger later on when your program is completed:

```
score = 0
RANGE = 5
```

5. As you are going to develop and test this program in steps, first write some dummy functions for each of the features of the program. Python functions need at least one statement in them, so you can use the Python `print` statement here to print a message. This just acts as a placeholder for code you will write later:

```
treasure_x = None # the x-coordinate of the treasure

def placeTreasure():
    print("placeTreasure")

def checkHit():
    print("checkHit")

def homingBeacon():
    print("homingBeacon")

bridge = []

def buildBridge():
    print("buildBridge")
```

- Now write the main game loop. You will run this loop quite fast when the game is running (10 times per second) so that the gold block trail is accurate enough to walk along in the sky, but slow it right down to once per second while you are testing it. Inside this game loop you will use your dummy functions, which you will write soon.

```
while True:  
    time.sleep(1)  
  
    if treasure_x == None and len(bridge) == 0:  
        placeTreasure()  
  
        checkHit()  
        homingBeacon()  
        buildBridge()
```

- Save your program with File→Save and then click Run→Run Module from the menu to run it. You should not have any errors in the program, and for now it should just print some messages once per second on the Python Shell window. You now have the framework of your program in place, ready for you to start adding new code to it, bit by bit.

Placing Treasure in the Sky

The first function you need to write is the one that places treasure at a random position in the sky. You will use three global variables for the coordinates of the treasure, and their initial value will be `None`. The `None` value is a special Python value, indicating that the variable is in memory but has nothing stored in it. You will use this in your game loop to check whether a new piece of treasure needs to be built.

- Create the global variables to track the position of the treasure, add the lines in bold:

```
treasure_x = None  
treasure_y = None  
treasure_z = None
```

- Fill in the `placeTreasure()` function (and take out the `print` statement you put in earlier) with this code:

```
def placeTreasure():  
    global treasure_x, treasure_y, treasure_z  
    pos = mc.player.getTilePos()
```

3. Use the `random` function to place the treasure at a position no more than `RANGE` blocks away from the player, but set the y coordinate so that it is somewhere above the player (which will probably place it in the sky):

```
treasure_x = random.randint(pos.x, pos.x+RANGE)
treasure_y = random.randint(pos.y+2, pos.y+RANGE)
treasure_z = random.randint(pos.z, pos.z+RANGE)
mc.setBlock(treasure_x, treasure_y, treasure_z,
block.DIAMOND_BLOCK.id)
```

Run your program and test that a piece of treasure is created up in the sky, near where your player is standing.

Collecting Treasure when It Is Hit

Now you will use the code from the `blockHit.py` program with a few small modifications to detect when the treasure is hit by your player's sword.

1. Remove the `print` statement from the `checkHit()` function and replace it with the code shown here. The `score` and `treasure_x` variables have to be listed as global variables here, because the `checkHit()` function will change their values. Python requires you to list inside a function, any global variables that it changes the value of. If you don't do this, your program will not work:

```
def checkHit():
    global score
    global treasure_x
```

2. Read through any block-hit events and check if the position matches the position of your treasure:

```
events = mc.events.pollBlockHits()
for e in events:
    pos = e.pos
    if pos.x == treasure_x and pos.y == treasure_y ←
        pos.z == treasure_z:
        mc.postToChat("HIT!")
```

3. Now you are going to tell your program to add points to the `score` for hitting the treasure, then delete the treasure so it disappears. Finally, you must remember to set `treasure_x` to `None` (so that `placeTreasure()` can create a new random piece of treasure later). Be careful with the indents here, as this code is part of the body of the `if` statement:

```
score = score + 10
mc.setBlock(treasure_x, treasure_y, treasure_z,
block.AIR.id)
treasure_x = None
```

Save and run your program, and check that when your player hits the treasure it disappears. You should also find that when you hit the treasure and it disappears, a new piece of treasure is created at a random position close to your player.

Adding a Homing Beacon

The homing beacon will display the score and the approximate distance to the treasure every second on the Minecraft chat. Here is how to add this.

1. Create a `timer` variable. As the main game loop will eventually run 10 times per second, you will have to count 10 loops for every second. This `timer` will help you to do that. If you change the speed of the game loop, you will have to adjust this `TIMEOUT` value as well. Make sure you put this code just above the `homingBeacon()` function (note, there is no indent at all here):

```
TIMEOUT = 10  
timer = TIMEOUT
```

2. Remove the `print()` from inside the `homingBeacon()` function and list the `timer` as a global variable, as this function will want to change its value:

```
def homingBeacon():  
    global timer
```

3. Treasure will be present in the sky if the `treasure_x` variable has a value in it. You have to check here if treasure has been created, otherwise you will get homing beacon messages on the Minecraft chat when there is no treasure to find:

```
if treasure_x != None:
```

4. This function will be called 10 times per second from the game loop, so you only want to update the homing beacon every 10 times:

```
    timer = timer - 1  
    if timer == 0:  
        timer = TIMEOUT
```

5. When the `timer` times out (every 10 calls to this function, or once every second), calculate a rough number that tells you how far away from the treasure you are. The `abs()` function will find the absolute value (a positive value) of the difference between two positions. By adding all the positive differences together, you get a number that is bigger when you are further away from the treasure, and smaller when you are nearer to it. Check your indents here, as this code all belongs to the body of the most recent `if` statement:

```
pos = mc.player.getTilePos()
diffx = abs(pos.x - treasure_x)
diffy = abs(pos.y - treasure_y)
diffz = abs(pos.z - treasure_z)
diff = diffx + diffy + diffz
mc.postToChat("score:" + str(score) + " treasure:" +
str(diff))
```

Save and run your program and make sure that the homing beacon and score are displayed on the Minecraft chat. Because you are still testing and developing your program, the game loop is set to run 10 times slower than normal, so you should see messages on the Minecraft chat every 10 seconds at the moment. Make sure this is the case by counting from 1 to 10 in your head. You will still see some of the dummy functions printing out on the Python Shell window every second for now, because your program is not quite finished.

Adding Your Bridge Builder

You will now add the bridge builder from your earlier `vanishingBridge.py` program. You only need to modify it a little, so that it checks whether your player is standing on gold and, if not, creates a gold trail.

1. Make sure your `buildBridge()` function looks like the following. The important lines that have changed from your `vanishingBridge.py` program are marked in bold:

```
bridge = []

def buildBridge():
    global score
    pos = mc.player.getTilePos()
    b = mc.getBlock(pos.x, pos.y-1, pos.z)

    if treasure_x == None:
        if len(bridge) > 0:
            coordinate = bridge.pop()
            mc.setBlock(coordinate[0], coordinate[1],
            coordinate[2], block.AIR.id)
            mc.postToChat("bridge:" + str(len(bridge)))
            time.sleep(0.25)
```

```
elif b != block.GOLD_BLOCK.id:  
    mc.setBlock(pos.x, pos.y-1, pos.z, block.GOLD_BLOCK.id)  
    coordinate = [pos.x, pos.y-1, pos.z]  
    bridge.append(coordinate)  
    score = score - 1
```

2. Congratulations! You have finished writing your program. As the game is ready to be played properly now, modify the `time.sleep(1)` in the game loop to sleep every 0.1 seconds. This will run the game loop 10 times per second. The timer in `homingBeacon` will count 10 of these and therefore only display a message on the Minecraft chat every second.

Save and run your program. Check that the gold trail disappears once you collect the treasure, and that your score goes down for every gold block that you spend.

Now all you have to do is enjoy the game! See how hard it is to get a good score by collecting the treasure?

Figure 4-7 shows the score and homing beacon display on the Minecraft chat.



FIGURE 4-7 The homing beacon showing vital statistics as you play skyHunt

Quick Reference Table

Getting the block type at a position	Finding out which blocks have been hit
<code>b = mc.getBlock(10, 5, 2)</code>	<code>hits = mc.events.pollBlockHits()</code> <code>for hit in hits:</code> <code>pos = hit.pos</code> <code>print(pos.x)</code>
Creating lists	Adding to the end of a list
<code>a = [] # an empty list</code> <code>a = [1,2,3] # an initialised list</code>	<code>a.append("hello")</code>
Printing the contents of a list	Working out the size of a list
<code>print(a)</code>	<code>print(len(a))</code>
Accessing items in a list by their index	Accessing the last item in a list
<code>print(a[0]) # 0=first, 1=second</code>	<code>print(a[-1])</code>
Removing the last item from a list	Looping through all items in a list
<code>word = a.pop() # remove last item</code> <code>print(word) # item just removed</code>	<code>for item in a:</code> <code>print(item)</code>

Further Adventures in Interacting with Blocks

In this adventure, you have learned how to use `getBlock()` to sense the block that your player is standing on, and how to use `events.pollBlockHits()` to respond when your player hits any face of any block. You've built a fantastic and complete game within Minecraft, complete with scoring!

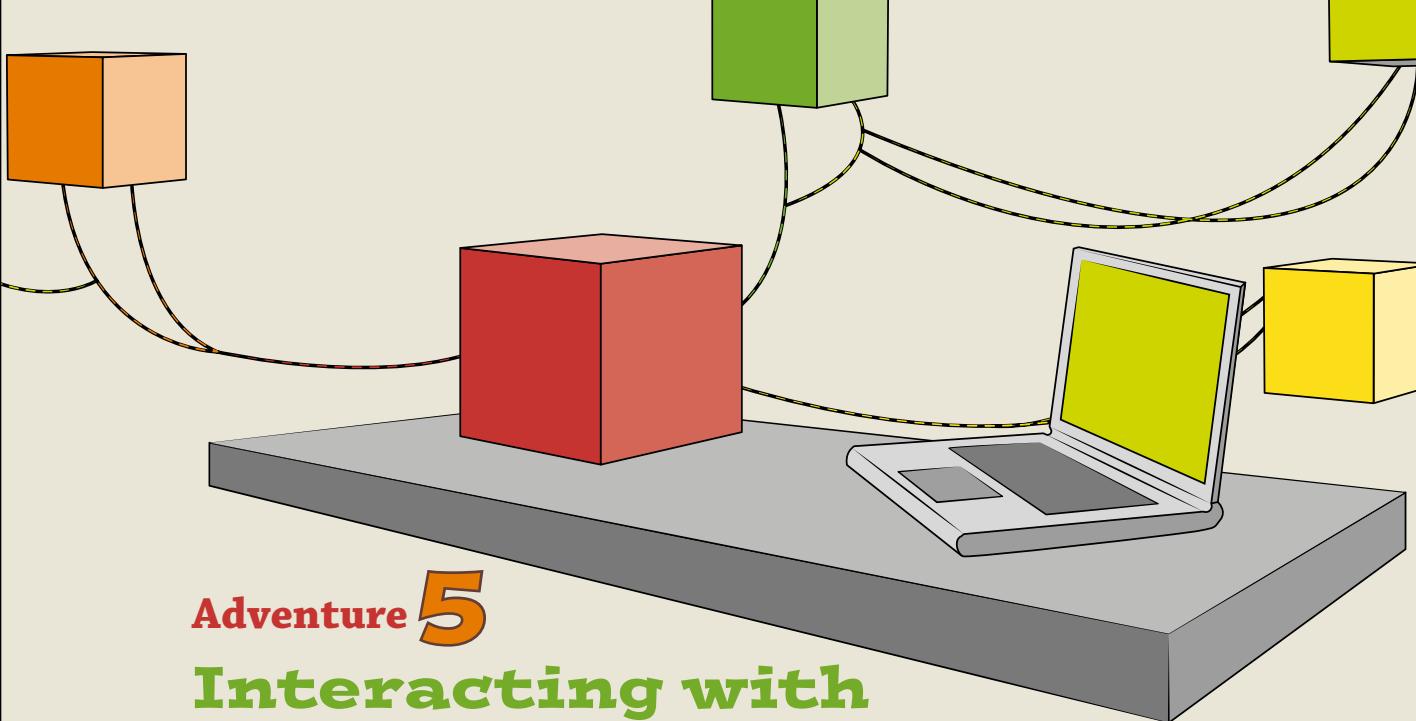
- Set the `RANGE` constant to a larger number so that treasure is created further away from your player. This will make your game more difficult to play, so try to add a new feature that displays cold, warm or hot on the Minecraft chat depending on the distance your player is from the treasure.
- Design a better scoring scheme to make it possible to get a reasonable positive score as you play the game normally. Do lots of testing and work out what the best score increment is for finding the treasure, and what the best score penalty is for spending a gold block.
- Research Pythagoras' theorem on the Internet, and see if you can code a better distance estimator to make the `homingBeacon()` function more accurate. (Psst! Martin covers a little bit about this in Adventure 7, so you could sneak a peek at that adventure and see if you can work it out!)



Achievement Unlocked: Expert in defying the laws of gravity and walking on water—two miracles achieved in one adventure!

In the Next Adventure...

In Adventure 5, you will learn how to break out of the confines of the Minecraft virtual game world and even beyond the boundaries of your computer, by linking Minecraft to real-world objects. You will use electronic components to build your own interactive game controller, and another complete game within a game!



Adventure 5

Interacting with Electronic Circuits

WHEN YOU PLAY in the Minecraft world, even with the programming interface, it is a virtual world. The only way you can interact with the game is by using your keyboard and mouse to direct the controls that were designed by the engineers who designed the program.

But there's another way to interact with Minecraft—by breaking out of the barriers imposed by the sandbox game and linking it to the physical world. Here you will quickly discover that the lines between what is virtual and what is real become blurred and your gaming experience becomes even more creative and exciting.

In this adventure you will learn how to link Minecraft to small electronic circuits through the API. You will begin by wiring up a light that flashes when you walk into your house. Then you are going to add a special type of display called a 7-segment display; you can use this to display countdowns and other information about your game as you interact with Minecraft. You will add a button that you can press to trigger all sorts of interesting actions in Minecraft. Finally, you are going to put all of it together to make a big red, fully functional detonator button counting down to an explosion, just like the one in Figure 5-1. It means you'll never have any difficulty clearing space for your building adventures in Minecraft again, and your friends will marvel at your new magic tricks and ask you to create detonator button circuits of their own for them!

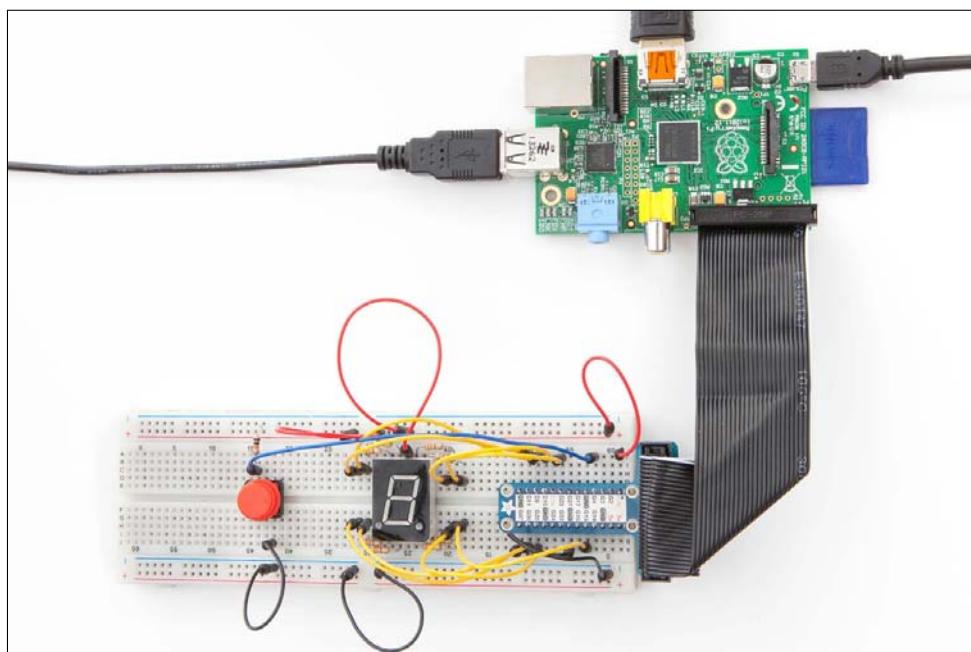


FIGURE 5-1 The detonator button built with a Raspberry Pi

What You Will Need for this Adventure

Here is a complete list of everything you'll need to build the electronic circuits described in this adventure—you will need some electronic components, and also some additional parts to allow you to connect your computer to these components. All the parts you need are shown in Figures 5-2 and 5-3:

- A breadboard
- At least four push buttons of the push-to-make type
- At least 1 LED of any colour (but more will be fun to use!)
- A 7-segment LED display
- Ten or more 330-ohm resistors for use with your LEDs and display
- Four or more 10K-ohm resistors for use with your buttons
- At least 20 jumper wires of the pin-to-pin type, or some thin solid-core wire and some wire strippers
- A small battery holder with two AA or AAA batteries

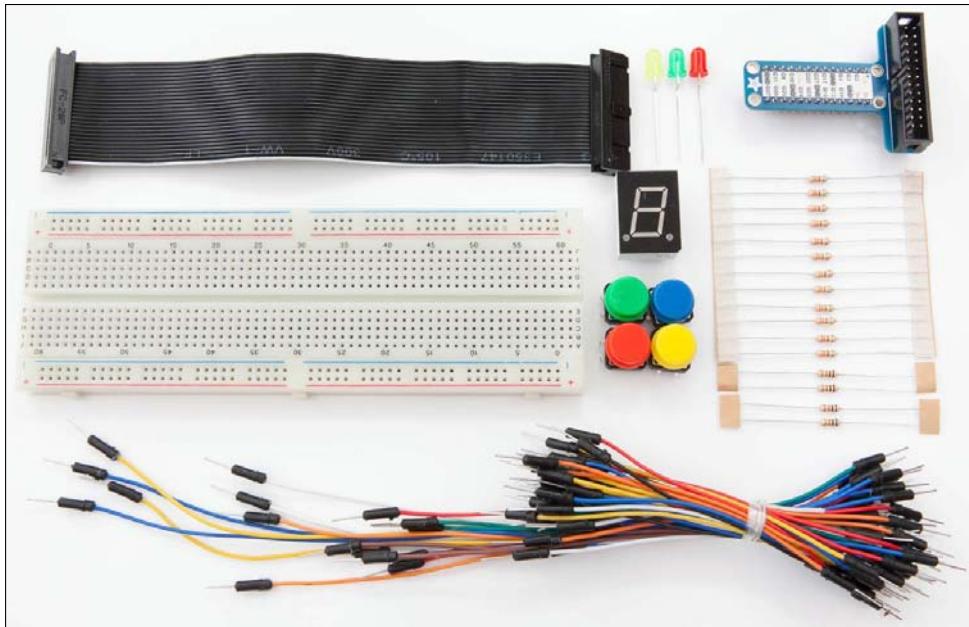


FIGURE 5-2 The parts you need to do this adventure with a Raspberry Pi

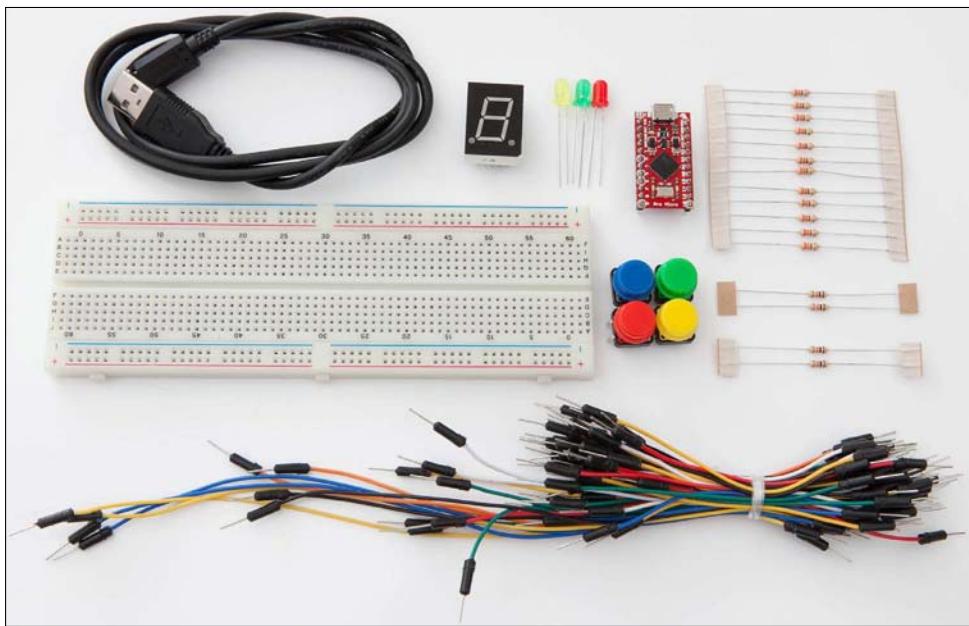


FIGURE 5-3 The parts you need to do this adventure with a PC or a Mac

There are plenty of online component suppliers where you can buy all of these parts, such as www.skpang.co.uk, www.sparkfun.co.uk, www.coolcomponents.com and www.protopic.com, as well as most high street electronic stores like Maplin Electronics (www.maplin.co.uk). Some stores sell bags of components with a collection of LEDs and resistors, as well as bags of jumper leads. If you are finding it hard to find jumper leads of the pin-to-pin type, you can buy a length of solid-core wire and some wire strippers to make your own jumper leads quite simply. This adventure assumes you have bought jumper leads, but the instructions are the same for solid-core wire.

For the Raspberry Pi:

You can use jumper leads of the socket-to-pin type to connect directly from the pins on the Raspberry Pi to the holes on the breadboard. This can sometimes be quite fiddly, however, and it's a bit tricky to count through all the pins. In this adventure I use a pre-soldered Adafruit Pi-T-Cobbler, which is a ribbon cable that runs from the Raspberry Pi and plugs directly into the breadboard, complete with labelled pins. If you know how to solder, you can pick up a Pi T-Cobbler kit and solder it together yourself, which is much cheaper. Figure 5-2 shows you all the parts needed to do this adventure using the Raspberry Pi.

You can buy the Adafruit Pi-T-Cobbler direct from www.adafruit.com or from www.skpang.co.uk. Make sure you buy the pre-assembled Pi-T-Cobbler (unless you want to have a go at soldering as well!).



Just as this book was being written, the Raspberry Pi model B+ was released. All of the adventures in this book will work with the Raspberry Pi B+, however you should buy an additional lead with your Pi-T-Cobbler, so that you can correctly connect it to your Raspberry Pi B+ 40-way connector. The lead you need has a 40-pin socket at one end that plugs into the Raspberry Pi B+, and a 26-way connector at the other end that plugs into your Pi-T-Cobbler. You can get one of these from www.skpang.co.uk at the same time that you buy the Pi-T-Cobbler.

For the PC and Mac:

Because the PC and Mac don't have their own hardware pins like the Raspberry Pi, I have put together a small hardware platform that you can use on these computers which is based around an Arduino. You can use any Arduino for this purpose, providing that you pre-load my special open source software that makes it function just like the Raspberry Pi input/output pins, and you can read about the Arduino here: <http://arduino.cc>. In this adventure you will use an Arduino called the Arduino Pro Micro. You will also need a USB lead to connect this to your computer.

You can buy the Arduino Pro Micro from <https://www.sparkfun.com/products/12587> product number: **DEV-12587**, but it needs some soldering and you need to program it with the special software to make it work as an input/output board. To save you the time and bother of doing this, you can buy a pre-soldered and pre-tested board with the software preloaded from <http://skpang.co.uk/catalog/pro-micro-33v8mhz-with-headers-and-anyio-firmware-p-1327.html>, product number AR-PROMICRO-ANYIO. All you then need to do is to plug the board into your breadboard and connect it via a USB cable to your PC or Mac, and it will work as an input/output board. Figure 5-3 shows you all the parts you need to set off on this adventure using a PC or a Mac.

If you have a Mac, it must run at least OS X 10.6 for the Arduino board to work.



There is a small collection of software source files you need to complete this adventure. All of these files are included in your starter kit's downloadable from the companion website. I have also provided links later on to where you and your friends can download the latest version of these software source files, in case you also want to use them in your non-Minecraft projects and experiments too!



Once you have bought all the bits and pieces you need, let's get started and build your first electronic circuit!

Prototyping Electronics with a Breadboard

When an electronics engineer designs the circuit for a new product, they will normally **prototype** the circuit board before forging ahead and building thousands of them.

A **prototype** is a small model of something to allow you to test that it will work. It is far easier to fix problems if you have built only one of something, than if you have built thousands—or even millions.





There are some fantastic pictures on the Internet of the very first prototypes of the Raspberry Pi computer that Eben Upton at the Raspberry Pi Foundation built back in 2006. Most complex circuit boards start off as a prototype like this, and may go through many revisions and modifications before they end up looking like the circuit board you have in front of you now. You can see these pictures here: <http://www.raspberrypi.org/raspberry-pi-2006-edition/>

Most electronic circuits are soldered together so that the components and the circuit boards have a very strong and long lasting electrical connection. But while soldered circuit boards provide a really long-lasting product, they are not very helpful when all you want to do is experiment with components and design and test circuits, because you would spend a lot of time de-soldering and re-soldering the components. For this, engineers will typically use a device called a solderless **breadboard**.



A **breadboard** is a reusable device that allows you to create circuits without needing to solder all the components. Breadboards have a number of holes into which you push wires or jumper cables and components to create circuits. The two rows of holes at the top and bottom of the breadboard are for power. Normally there is a red line, which is used for the positive connection (such as 3.3 volts, sometimes labelled as + or VCC), and a black or blue line, which is used for the 0 volts (or ground connection, sometimes labelled as - or GND).

Figure 5-4 shows two pictures of a solderless breadboard. On the left is a breadboard with some components plugged into it to make a circuit, and on the right is a picture of the insides of the breadboard. If you look closely, you can see the metal strips that make up the connections. Understanding how a breadboard is wired is very important because it means that, when you push components into the holes of the breadboard, you will know how things are going to connect together to make an electrical circuit.

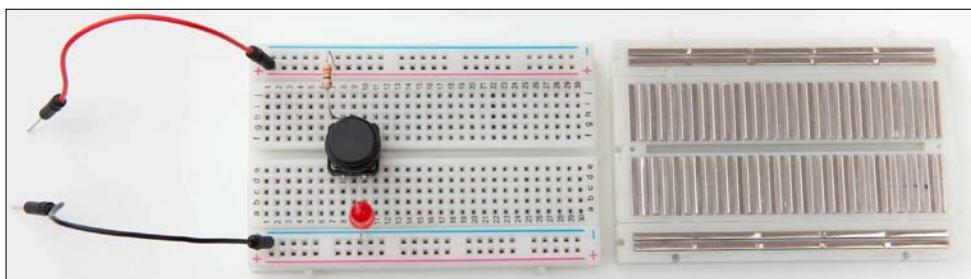


FIGURE 5-4 Left: a breadboard; right, the insides of a breadboard showing its internal wiring

Building a Circuit that Lights an LED

The circuit in Figure 5-4 is a simple circuit that can be used to test components. For any circuit to work, a **voltage** must be applied to it. A 3-volt battery needs to be connected to the strips at the top and bottom of the breadboard, which are called power rails. These are long rails that connect the full length of the breadboard, and are a convenient method of providing electrical power to lots of components at the same time. The red component is a **light-emitting diode (LED)**, which is a type of light that you will meet in a moment. The component with coloured bands on it is a **resistor**, and the black device at the bottom of that is a button. If you follow the wires from the battery right round through all the components in the circuit, you eventually end up back at the battery again. This is a complete electrical circuit, and when you press the button, electrical **current** will flow and the LED will light up.

Voltage is the difference in electrical energy between two points in a circuit. It is the electrical equivalent of water pressure in pipes, and it is this pressure that causes a current to flow through a circuit. Voltage is measured in *volts* (V).



A **Light-emitting diode (LED)** lights up when electricity passes through it. A diode only allows current to pass in one direction. An LED will only light up if you pass current through it in the correct direction. LEDs come in a variety of colours and have one short leg (the cathode or negative) and one long leg (the anode or positive), which helps you to determine which way they need to be placed in a circuit for current to flow through them.



A **resistor** is an electrical component that resists current in a circuit. For example, LEDs can be damaged by too much current, but if you add the correct value resistor in your circuit with the LED to limit the amount of current, the LED will be protected. Resistance is measured in *Ohms*, and often indicated by the use of the Omega symbol Ω . You need to pick a resistor with the correct value to limit the current through a circuit; the value of a resistor is shown by coloured bands that are read from left to right. You can find out about the resistor colour code here: http://en.wikipedia.org/wiki/Electronic_color_code



DIGGING INTO THE CIRCUIT

In the circuit in Figure 5-4, a resistor is used. An LED is a delicate component that can only cope with a small amount of current, usually no more than about 20mA (that's 0.02 of an amp, which is quite small). If an LED is connected directly to a battery, too much current will flow around the circuit and it will damage the LED. To prevent this from happening, a resistor is added that limits the flow of current around the circuit. The value of the resistor is indicated by the coloured bands on its body.

If a higher battery voltage is used, you will have to use a higher value of resistance to limit the amount of current that flows in the circuit. I have calculated the resistor values in this project for you, but there are plenty of websites that will help you recalculate the resistor value for different battery voltages, such as www.ohmslawcalculator.com/led_resistor_calculator.php.



A **current** is the rate at which electrical energy flows past a point in a circuit. It is the electrical equivalent of the flow rate of water in pipes. Current is measured in *amperes* (A), often abbreviated to Amps. Smaller currents are measured in *milliamperes* (mA). There are some interesting theories about the direction that current actually flows in a circuit, which you can read about here: www.allaboutcircuits.com/vol_1/chpt_1/7.html

CHALLENGE



See if you can build the circuit in Figure 5-4 with your electronic components, so that when you press the button the LED lights up. You will need a small battery pack with two AA or AAA batteries. Make sure that you put the LED in the circuit the right way. If the LED does not light, check your wiring very carefully, and make sure that you have the LED in the circuit the right way.

Connecting Electronics to Your Computer

Soon you are going to connect an LED to your computer and control it from a Python program. When you are controlling electronic components from your computer, your computer needs a way to connect and control those circuits, and for that you need some **general purpose input outputs (GPIOs)**.



General purpose input outputs (GPIOs) are signals connected to the central processor of a computer system. They are designed with no specific purpose in mind but are general purpose and can be assigned a use by the designer of a circuit, usually to control and sense external electronic circuits.

On some computers, like the Raspberry Pi, the GPIO circuitry is brought out to special pins on the side of the computer. A GPIO pin can either be set up as an input to sense some external voltage, or it can be set up as an output to control some external voltage. The voltages used by your Raspberry Pi or your Arduino board are 3.3 volts, which is quite small, but there is enough current provided by these pins in order to power an LED or sense a button press. By comparison, the UK mains electricity supply is 240 volts, which is a very high (and dangerous) voltage, and the electricity pylons you see in the country use 230,000 volts! By comparison, the voltages used in computer circuits are very small.

A computer is a digital machine, meaning that it uses numbers internally. All modern computers are digital and use the numbers 1 and 0. When using GPIO pins with a computer, a 0 normally represents no voltage (off), or 0 volts, and a 1 normally represents a voltage being present (in this case, 3.3 volts). When you use GPIO pins to sense external voltages, the computer can only internally represent the voltage on these pins as a digital 0 or 1. A voltage that is close to 0 volts is a 0, and a voltage that is close to 3.3 volts is a 1. Anything in the middle is a bit of a grey area and might be seen as a 0 or a 1 by the computer.

Setting Up the PC or Mac to Control Electronic Circuits

If you have a Raspberry Pi, you can skip this section and go onto the next section, “Controlling an LED”. This is because the Raspberry Pi has its own GPIO pins already built in, which makes it a really good computer for controlling electronic circuits.

PCs and Macs don't have GPIO pins built in, so you have to add them with an extra circuit board. There are lots of ways you could do this, but for this book I have chosen to use a small, pre-programmed Arduino board. You won't be learning anything about the Arduino in this book but if you want to know more about the little computer, check the links in the Where Next section in the appendix at the back of this book.



In the instructions that follow, I am assuming that you have bought the pre-soldered and pre-programmed board listed at the beginning of this adventure. If you have decided to use the bare board and solder and program it yourself, you will have to follow the instructions in the hookup guide on this website: <https://www.sparkfun.com/products/12587> and load in the special open source GPIO software I have written for it. This software, along with a Python module called `anyio` that is included in the starter kits, makes the Arduino board work just like the GPIO pins on a Raspberry Pi.

This will mean that the rest of the programs in this adventure will work by only changing a single line in your Python programs. The software that needs to be loaded into the Arduino board is stored inside the starter kit in the `anyio/arduino/firmware` folder if you need to program it yourself. You can also always find the latest version of the `anyio` package on my github page here if you or your friends also wanted to use it in your non-Minecraft projects and experiments: <https://github.com/whaleygeek/anyio>

Configuring the Drivers

When devices are plugged into computers, the computer has to have special software installed that allows it to communicate with that device—this software is called a device driver. You have probably had to install device drivers before on your computer when plugging in a new printer or other device, and you have to do the same here too.

1. Plug the USB lead into the Arduino, and the other end into your computer.
2. Follow the onscreen instructions and prompts (which, unfortunately, vary among different versions of the operating systems), and the following steps to install the drivers so that you can use the Pro Micro with your computer.

On the Mac:

You will see a message pop up saying that a new keyboard has been added. Just cancel that box by clicking the red cross. That's it—your installation on the Mac is now done!

On the PC:

You will be asked to download a driver. You don't need to download a driver, however, as Windows includes all the necessary drivers. But Windows is not very good at recognising an unknown device, so I have provided a file inside the starter kit that helps Windows detect the device properly. It is called `ProMicro.inf` stored in the `anyio/arduino/firmware` folder of the starter kit, which is inside your `MyAdventures` folder. Browse for a file, choose the `ProMicro.inf` file and click OK. Windows should now recognise the device as a USB serial port.

If at this stage your computer doesn't seem to recognise the device, go to the following sparkfun page and work through the instructions in the hookup guide, which is quite detailed, and also is updated when each new version of an operating system is released: <https://www.sparkfun.com/products/12587>. You might also like to look at SKPang's product page for the ProMicro as he has more instructions on there to help you out: <http://skpang.co.uk/catalog/pro-micro-33v8mhz-with-headers-and-anyio-firmware-p-1327.html>



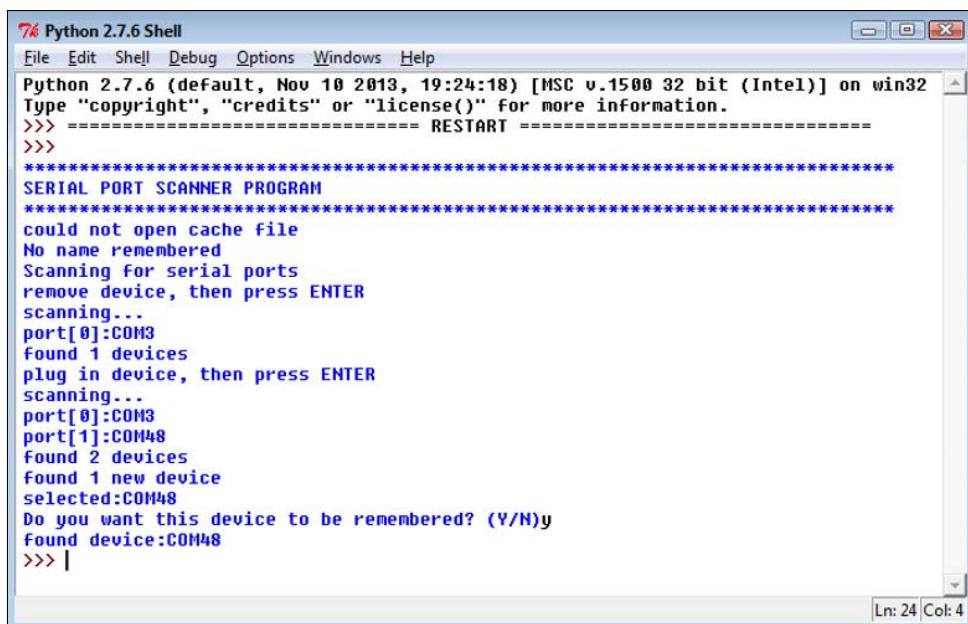
Finding the Serial Port Number

The Arduino you have just connected to your computer appears to the computer as a device called a serial port. I won't go into the details of what that means in this book, but you do need to make sure that the `anyio` package selects the correct port before it will work as a GPIO extender.

I have written a little utility to do this for you, and it is included in the `anyio` package. Figure 5-5 shows a sample session using this utility, so you can see what it should normally look like.

To make sure the correct port is selected:

1. Open IDLE and click File ➔ Open to browse for an existing Python program.
2. Inside the `MyAdventures` folder, open the file called `findPort.py` and it will load into the Python editor window.
3. Run the program by choosing Run ➔ Run Module from the menu. You will get some onscreen instructions.
4. Unplug the USB lead from the Arduino end, then press Enter to start the scan. This step removes the USB lead from your computer so that the program can scan all available ports in your computer and build a list of any ports except the new one you are adding.
5. Next you will be asked to plug in the device so that the program can scan all available ports in your computer again, and use that to detect which new device has been added. Plug in your USB lead again, wait a couple of seconds, then press Enter again.
6. If this has been successful, you should get a message indicating the port name or number that has been found, and you should be asked if you want to remember this. Type a Y then press Enter. The `findPort.py` program (shown in Figure 5-5) then creates a file called `findport.cache` that remembers the port number of your Arduino. When you use any of the GPIO functions later it will be able to find the Arduino port correctly.



The screenshot shows a Python 2.7.6 Shell window. The title bar reads "Python 2.7.6 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main window displays the following text:

```
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
*****
SERIAL PORT SCANNER PROGRAM
*****
could not open cache file
No name remembered
Scanning for serial ports
remove device, then press ENTER
scanning...
port[0]:COM3
Found 1 devices
plug in device, then press ENTER
scanning...
port[0]:COM3
port[1]:COM48
Found 2 devices
Found 1 new device
selected:COM48
Do you want this device to be remembered? (Y/N)y
Found device:COM48
>>> |
```

At the bottom right of the window, there are status indicators: "Ln: 24 Col: 4".

FIGURE 5-5 The output from the `FindPort.py` program



If you plug other USB devices into your computer, you might find that the next time you restart your computer or plug in your Arduino board, the serial port number changes and your program stops working. If this happens, just re-run the `findPort.py` program, or delete the `findport.cache` file and run your program to perform another scan for ports.



Writing a package that works reliably on every possible version of an operating system is quite hard to achieve, and in future newer operating systems may be released. If you have problems with your operating system not installing or not finding the port properly, take a look at the companion Wiley website (www.wiley.com/go/adventuresinminecraft) to see if there is an updated version of the package, or you can always find the latest news and package updates on my github page here: <https://github.com/whaleygeek/anyio>

Controlling an LED

In Adventure 1 you wrote a program called `helloMinecraftWorld.py`, which printed a message on the Minecraft chat. When you are controlling electronics, there is an equivalent “hello world” program that you can write. Instead of showing a message

onscreen, this program flashes an LED. If you can program the computer to flash a single LED when everything is set up, then you know that everything is working and you can expand to bigger and more exciting projects.

This is going to be a special LED, though, because it will be linked to the Minecraft game. You are going to extend an idea that you experimented with in Adventure 2 when you wrote the magic doormat. This time, your doormat is going to be connected to your circuit board and the LED will flash when your player stands on the doormat. This electronic circuit can be connected to anything, though, not just an LED. Just imagine what you could program to happen when your player stands on the doormat—it could turn the lights on or open the curtains in your real bedroom, and more!

In this project, if you are using a PC or a Mac, you will use the `anyio` package. This package is provided in the starter kit, and is also downloadable from the links listed at the start of this adventure.



Lighting Up an LED from your Computer

First you are going to wire up the circuit on the breadboard. Figures 5-6 and 5-7 show a diagram of how the LED is wired to your computer.

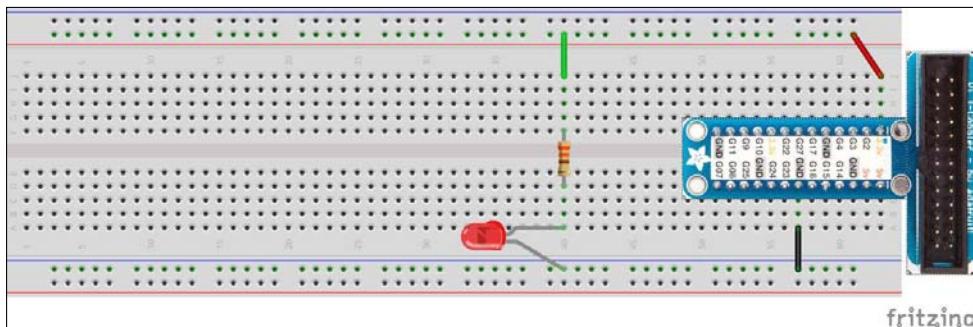


FIGURE 5-6 An LED and a resistor wired to a Raspberry Pi

1. Look at the LED and find the shortest wire. This is the negative side of the LED (the cathode). Connect this to the 0V power rail at the bottom of the breadboard.
2. The other leg of the LED will be longer, and this is the positive side (the anode). Push one leg of the resistor into the same breadboard strip as the LED long lead and the other leg of the resistor into a strip on the top half of the breadboard. Remember that the resistor is needed to limit the amount of current that flows through the LED.

3. So that you can test the LED first, run a wire from the resistor leg on the top half of the breadboard to the positive power rail at the top of the breadboard. You will move this a little later.

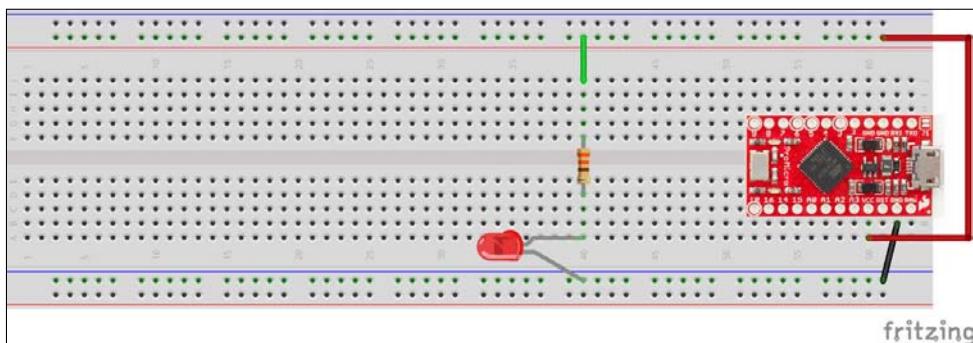


FIGURE 5-7 An LED and a resistor wired to an Arduino

The instructions for the Raspberry Pi and the PC/Mac (using the Arduino) differ here a bit, so work through the section that is relevant to the board you are using.

On the Raspberry Pi:

1. If you have a sticky label in your kit for the Pi-T-Cobbler, stick it on so it looks like Figure 5-8. This will help you to find the correct GPIO pins more easily. Attach the Pi-T-Cobbler to the breadboard so that the black connector is on the right hand just hanging off the edge of the breadboard. Push it into the right hand side of the breadboard so that it lines up with the holes on the far right hand edge of the breadboard. Half of the pins should push into the top half of the breadboard, and half of the pins should push into the bottom half of the breadboard. Push quite hard to make sure it goes all the way in.
2. Connect the ribbon cable between the Pi-T-Cobbler and the Raspberry Pi. There is a notch in the socket and a slot in the plug of the Pi-T-Cobbler that means it will only go in one way. When you connect the ribbon cable to the Raspberry Pi, make sure that the plastic lug on the connector faces away from the edge of the Raspberry Pi circuit board; otherwise all the pins will be connected the wrong way and it won't work! Figure 5-8 shows what this looks like.



The last thing to do is to check that your computer can turn the LED on. Before flashing it with a Python program, you will first test out the LED by using power provided by your computer.

- Run a wire between the positive power rail at the top of the breadboard and the pin on the Pi-T-Cobbler labelled as 3V3.
- Run a wire between the negative power rail at the bottom of the breadboard and the pin on the Pi-T-Cobbler labelled as 0V.

Provided your Raspberry Pi is powered up, your LED should now light up, as it is being powered by the power supply of your computer.

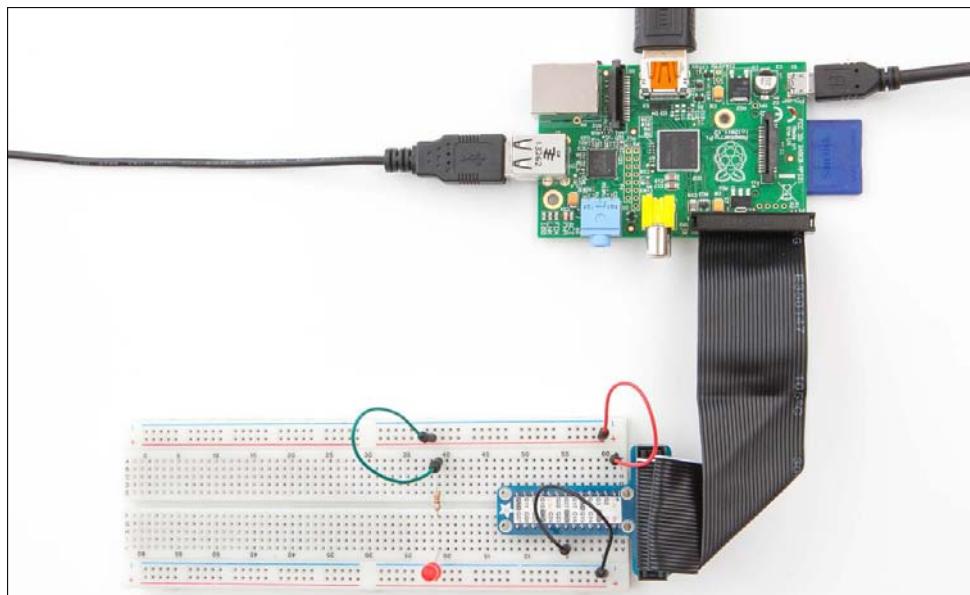


FIGURE 5-8 The Pi-T-Cobbler connected to the Raspberry Pi

On the Arduino:

- Line the Arduino up on the right-hand edge of the breadboard so that the silver USB socket hangs off the right-hand edge. You might have to push quite hard to get the pins to push into the breadboard, but put even pressure across the whole circuit board so that it doesn't break. Figure 5-9 shows what this looks like.
- Connect the USB lead from your computer to the Arduino.

The last thing to do is to check that your computer can turn the LED on. Before flashing it with a Python program, you will first test the LED out by using power provided by your computer.



- Run a wire between the positive power rail at the top of the breadboard, and the pin on the Arduino labelled as VCC. (VCC is just a technical term that means the positive power supply, which on your Arduino board is 3.3 volts.)
- Run a wire between the negative power rail at the bottom of the breadboard and the pin on the Arduino labelled as GND. (GND is just a technical term that means the 0-volt power supply connection.)

Provided your Arduino is powered up, your LED should now light up, as it is being powered by the power supply of your computer.

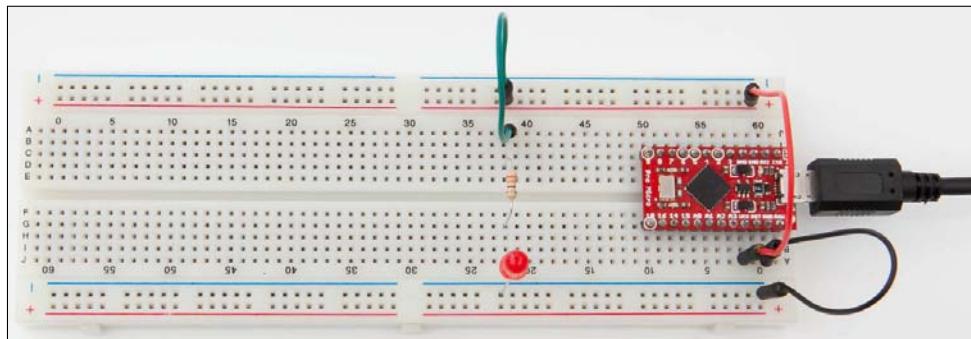


FIGURE 5-9 The Arduino connected to the breadboard



If your LED does not light up, check that it is the right way. Turn it around the other way to see if it works. You can also check back against the diagram of how a breadboard is wired up internally, and follow the circuit from the 3.3 volt power pin, all the way round the circuit until you get to the 0-volt pin, checking that each connection is made correctly. The usual reason for the LED not lighting up is that it is in the wrong way, or you have plugged its legs into the wrong holes in the breadboard so that it is not making a complete circuit.

Flashing the LED

Now that you know that your LED is wired up properly and working, the final thing to do is to write a Python program that makes it flash:

- Start IDLE, and create a new program by clicking **File**→**New** and then save the program as **testLED.py**.
- Import the time module, which you will use to insert a delay between each flash of the LED.

```
import time
```



In all of the programs in this adventure, I use `import RPi.GPIO as GPIO` or `import anyio.GPIO as GPIO` – this is just like you used before with `import mcpi.minecraft as minecraft`. The `as` tells Python to rename the module so that it uses the name on the right hand side, and this is helpful when the module names are very long. It also means that the instructions and programs in this adventure only have to have one or two lines changed and they will work on either the Arduino or on the Raspberry Pi. This is a pretty neat feature of Python!

3. Import the module that allows GPIO to be controlled, and define a constant for the GPIO number that will be connected to your LED, by doing the following:

On the Raspberry Pi:

```
import RPi.GPIO as GPIO  
LED = 15
```

On the Arduino:

```
import anyio.GPIO as GPIO  
LED = 15
```

4. Set the GPIO pin numbering mode. (You'll see the letters BCM included in the code here. For the Raspberry Pi, this stands for "Broadcom". This is the name of the processor chip, and all it means is that you are telling the Raspberry Pi to use the GPIO numbers rather than the pin numbers on the board.) Now configure the GPIO as an output so that your program will be able to change the voltage applied to your LED:

```
GPIO.setmode(GPIO.BCM)  
GPIO.setup(LED, GPIO.OUT)
```

5. Now write a function that will flash the LED once. A parameter `t` is given to this function so that later on you can change the speed at which the LED flashes, and the `output()` function changes the voltage on the GPIO. When you use `True` the voltage will rise to 3.3 Volts, and when you use `False` the voltage will fall to 0 Volts. This will turn the LED on and off:

```
def flash(t):  
    GPIO.output(LED, True)  
    time.sleep(t)  
    GPIO.output(LED, False)  
    time.sleep(t)
```

6. Write a game loop that flashes the LED, and finally call the `GPIO.cleanup()` function when the program finishes, so that the GPIO pins are left in a safe state at the end. You can read about the `try/finally` in the Digging into the Code sidebar later.

```
try:  
    while True:  
        flash(0.5)  
finally:  
    GPIO.cleanup()
```

7. For the LED to flash, it needs to be connected to the GPIO pin on your computer. Do this by following these steps:

On the Raspberry Pi:

Move the wire that connects the resistor and the 3.3 volt power rail so that the end that was connected to the power rail is now connected to GPIO 15 (labelled as G15 on the Pi-T-Cobbler label).

On the Arduino:

Move the wire that connects the resistor and the 3.3 volt power rail so that the end that was connected to the power rail is now connected to GPIO 15 on the Arduino.



The Arduino board has some LEDs on the board itself, and the yellow LED on this board indicates when the Arduino is receiving commands over the USB. You will see this LED on the board flashing at twice the rate of the LED on your breadboard, because each command to turn your LED on and off will cause the Arduino's yellow LED to flash. Don't be put off by this, it is working correctly!

Running a GPIO Program

Finally, it is time to run your program and see if your LED flashes at you!

On the PC/Mac:

You can just run the program as usual from IDLE. The Arduino IO board is connected via the USB and the PC/Mac have full access to connect to this device.

On the Raspberry Pi:

Running the program is slightly more complicated on the Raspberry Pi. You will have to run any program that uses the GPIO differently from other programs, although it is still useful to enter and edit your programs in IDLE. You can read more about why you have to do these extra steps in the Digging into the Code sidebar.

1. Open the LXTerminal program from the desktop or from the menu. This will open a command prompt window that you can use to type commands into the Raspberry Pi.

2. Change into your `MyAdventures` folder by typing the `cd` command.

```
cd MyAdventures
```

3. Run your Python program using `sudo` (read about this in the Digging into the Code sidebar):

```
sudo python testLED.py
```

Did your LED flash? It seems like such a small thing but in fact you have taken a huge step by getting to this point. Now that you can control electronic circuits, you have broken outside the boundaries of the computer!

You may see a warning here “Channel already in use”. Don’t worry, this is normal, and later in this adventure you will learn how to use `GPIO.cleanup()` to prevent this warning message.



The last thing for you to do is to link this program up to the Minecraft world, and this is what you will turn to next.

Remember that flashing an LED from a Python program is the electronics equivalent of “Hello World”. When you link computers up to electronic circuits, it is important always to try to get this working first, before advancing to things that are more involved. If your LED does not flash, check your wiring very carefully, and make sure that you have the wire between the LED and the GPIO pin in the correct hole in the breadboard. You tested earlier that your LED was working by powering it from the 3.3 volt power supply of the computer, so if it does not work, either your Python program is wrong, or the wire between the LED and the GPIO pin is wrong.



DIGGING INTO THE CODE

There are two bits of magic that you have just used that you might like to understand a little better.

First, what is the `try/finally/GPIO.cleanup()` for?

To prevent warnings from appearing every time you run your program, always call `GPIO.cleanup()` before the program finishes. By doing this, it puts all of the GPIO pins back as inputs and disables the GPIO circuitry. “To put back as an

continued

continued

input” means that the pin is not actively trying to control the circuitry any more. If you accidentally short an output pin to another pin when connecting or disconnecting, you can damage the circuitry. To prevent this from happening, it is good practice to make your program call the `GPIO.cleanup()` function, so that accidental short circuits when making changes to your circuit have no chance of damaging the circuitry.

But you have a problem: because you have put `while True:` in your Python program, it means that your program is in an infinite game loop. The only way to break out of this program is to press CTRL+C or to stop the program from the IDLE menu. But when you stop a program that way, it just stops immediately.

Don’t worry—there’s a way to deal with this too, and it involves `try/finally`, which is what is known as an exception handler. You won’t be learning about exception handlers in any detail in this book, but you can’t avoid using them with GPIOs! Basically, when you press Ctrl+C or stop the program from the IDLE menu, Python generates an exception called `KeyboardInterrupt` and the program stops. Using `try/finally` is a little programmer’s trick that allows your Python program to detect that your program has stopped. After the `finally` statement you can put any Python code that needs to run to clean up before it finally finishes. So, you press CTRL+C or stop the program from the IDLE menu, the program jumps to the code in the `finally:` block and runs that, then it finishes. This ensures that `GPIO.cleanup()` is always called and that the GPIO pins are always left in a clean and safe state when your program has finished.

I won’t really go into exceptions in any great detail in this book, but you can read more about them here: <https://wiki.python.org/moin/Han...>

Secondly, what is `sudo` for on the Raspberry Pi, and why do you have to run your GPIO programs from within LXTerminal?

All hardware on the Raspberry Pi is protected so that it cannot normally be accessed by the normal `pi` user that you log in as, but you have to have super-user privileges instead. `sudo` means “substitute user and do,” which basically changes your user from `pi` to `root` (the super user) and then runs your `testLED.py` program inside the Python language. This means that your program can access the protected GPIO hardware. If you don’t run your GPIO programs in this way on the Raspberry Pi, you will get an error.

When writing this book I tried to find a clear explanation of `sudo`, but interestingly, all my Google searches ended up taking me to a page on my blog, which seems quite popular! You can read all about `sudo` and its uses here: <http://blog.whaleygeek.co.uk/sudo-on-raspberry-pi-why-and-why-you-need-to-ask-kids-too>.

Writing the Magic Doormat LED Program

The last thing you are going to do with your LED is to link it up to the Minecraft game. Fortunately, this final step is quite easy and builds on everything you already know how to do.

1. Save your existing `testLED.py` program under a different name by choosing File→Save As from the menu and calling it `welcomeLED.py`.
2. At the top of the file, add the imports for the Minecraft modules you need, and connect to the Minecraft game:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
mc = minecraft.Minecraft.create()
```

3. Below this, set some constants that will be the coordinates of your welcome home mat. You need to decide where you want the mat to appear in your world, and choose these constants accordingly. On the Raspberry Pi you can see your coordinates in the top left of the Minecraft window. On PC/Mac you can press F3 to find out the coordinates of your player:

```
HOME_X = 0
HOME_Y = 0
HOME_Z = 0
```

4. Below these constants, create a block out of wool at the mat home location. This is your doormat—just like the doormat you created in Adventure 2:

```
mc.setBlock(HOME_X, HOME_Y, HOME_Z, block.WOOL.id, 15)
```

5. Change your game loop so it now looks like this. Take special care over the indents. This code repeatedly reads the player position and geo-fences it to see if you are on the doormat. If you are, it flashes the LED to let you know you are home:

```
try:
    while True:
        pos = mc.player.getTilePos()
        if pos.x == HOME_X and pos.z == HOME_Z:
            flash(0.5)
finally:
    GPIO.cleanup()
```

Run the program using the instructions in the section on running a GPIO program earlier in this adventure, and make sure that when you stand on your doormat, the LED flashes to welcome you home!

This is fantastic! You now have all the ingredients you need to break out of the virtual world of the Minecraft sandbox and link up the game to real-world objects. If you can write a program that flashes an LED, you can go on to control any number of other electronic devices—you are limited only by your imagination and willingness to experiment!

CHALLENGE



Get some different coloured LEDs and wire them up to other GPIO pins on your computer. Use a blue LED and add this to your Python program using the `getBlock()` function so that if you are standing on water, it lights the blue LED. Get a green LED and make it light up if you are standing on grass. How many different things can you invent that turn the LEDs on when something happens inside your Minecraft world?

Using a 7-Segment Display

Your next project in this adventure will use a very common type of LED display to show numbers and other symbols when things happen inside Minecraft. You will also be able to use this display in the big game in the final adventure of this book. Before you link up the display to Minecraft, you first have to get the display connected and working. Once that's done, you can link it into your Minecraft game just as easily as you did with your flashing LED.



In this project, if you are using a PC or a Mac, you will use the `anyio` package. This package is provided in the starter kit and is also downloadable from the links listed at the start of this adventure.

What is a 7-Segment Display?

If you look around your house or go into any shop, you will see hundreds of different products that use the 7-segment display format. Digital watches, timers on microwave ovens, CD players, even central heating controllers, all use this pattern of seven segments. Figure 5-10 shows the distinctive pattern of a 7-segment display and also how it is wired internally.

All 7-segment displays have a number of different LEDs in the same device. The most common display, and the one you will use in this next project, actually has eight LEDs inside the same component, but one of them is reserved for the decimal point in the bottom corner of the display. Only seven of the LEDs are dedicated to the actual pattern, which as Figure 5-10 shows, is in the shape of a figure eight.

There are two important things for you to remember about the 7-segment display for this next project:

- Because the display has eight separate LEDs inside it, if you can write a program to turn one LED on and off, then you can easily write a program to drive this display; it just has eight separate GPIO connections.
- When you use this pattern of seven segments, you can display a great many symbols by using different combinations of LEDs. Using just these segments, it is possible to display any digit from 0 to 9.

In Figure 5-10, along with the internal wiring of a 7-segment LED display, you can also see the pin diagram showing which pins connect to which segments. You can see that each LED has its positive (anode) connected to the same common pin, and this is the pin that you will connect to the 3.3 volt power rail of your breadboard. The arrow of the diode symbol tells you which way the current flows, and so it points from the positive side to the negative side.

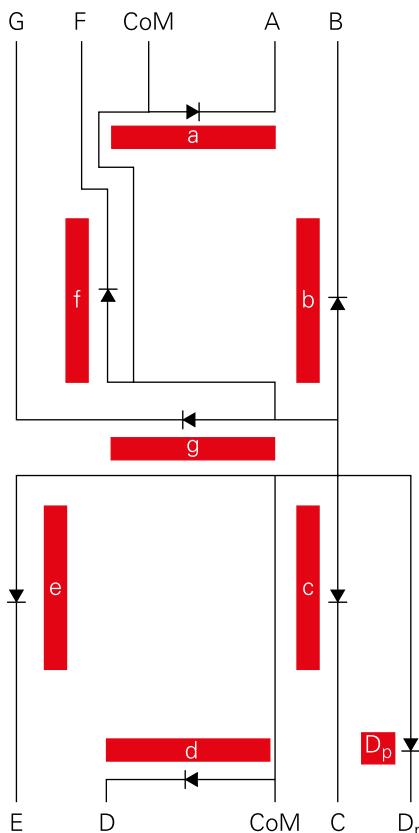


FIGURE 5-10 The segment layout and internal wiring of a common-anode 7-segment display



The labels A B C D E F G are a common convention used by the industry for labelling 7-segment displays. If you look carefully at many displays you will see that the segments are very slightly slanted to the right. This is just a design feature that most display manufacturers have adopted to make the segment display look more visually appealing.



There are lots of different types of 7-segment display that you can buy, and many of them are wired differently. Some displays are common-cathode (which means that all of the negative leads of each LED are connected together) and some are common-anode (meaning that all of the positive leads of each LED are connected together). You will also find that some 7-segment LED modules use different pins for different LED segments. This project uses a particular common-anode 7-segment display from www.skpang.co.uk, but if you buy a different display, check the wiring details carefully before wiring it up. If you have a different (common-cathode) display, I have put comments in all of the programs in this adventure next to the ON constant, so you know what to change if you have a different display.

Wiring Up the 7-Segment Display

Now you are ready to connect your 7-segment display to the breadboard to check that it works properly, by following these steps:

1. Push the 7-segment display into your breadboard so that half of it overlaps the top of the breadboard and half of it overlaps the bottom of the breadboard. There are five pins at the top and five pins at the bottom, and each pin lines up with a hole in the breadboard.
2. Run a wire from the top middle pin of the display (the common anode) to the 3.3 volt power rail at the top of the breadboard. (If you have a different display to me and you are using a common cathode display instead, you will have to run a wire from the bottom middle pin of the display to the 0 volt power rail at the bottom of the breadboard.)
3. Because this display has LEDs inside it, you need to use a resistor like you did before. But there are eight LEDs, so you need to use eight resistors. Take each 330-ohm resistor (colours orange, orange, brown), bend both of its legs over, and place each resistor between a specific pin of the display, and a spare hole in the breadboard. Figure 5-12 shows how the display needs to be wired up. Look back at Figure 5-4 to remind yourself which way the strips on the breadboard are connected, so that you don't short all your resistors together!

- Now you can test your display. Temporarily take a wire from the 0-volt power rail at the bottom of the breadboard and connect it to the unconnected leg of each resistor in turn (the leg of the resistor that is not connected to the display). Every time you touch a different unconnected leg of a different resistor, a different segment should light up. Compare the segments against the segment diagram in Figure 5-10, which shows the display I have used in this project, and make sure your display is internally wired up the same way. If it is different, keep a note of which segment letters are attached to which resistors, as you will need to know that later. (See Figure 5-11.)

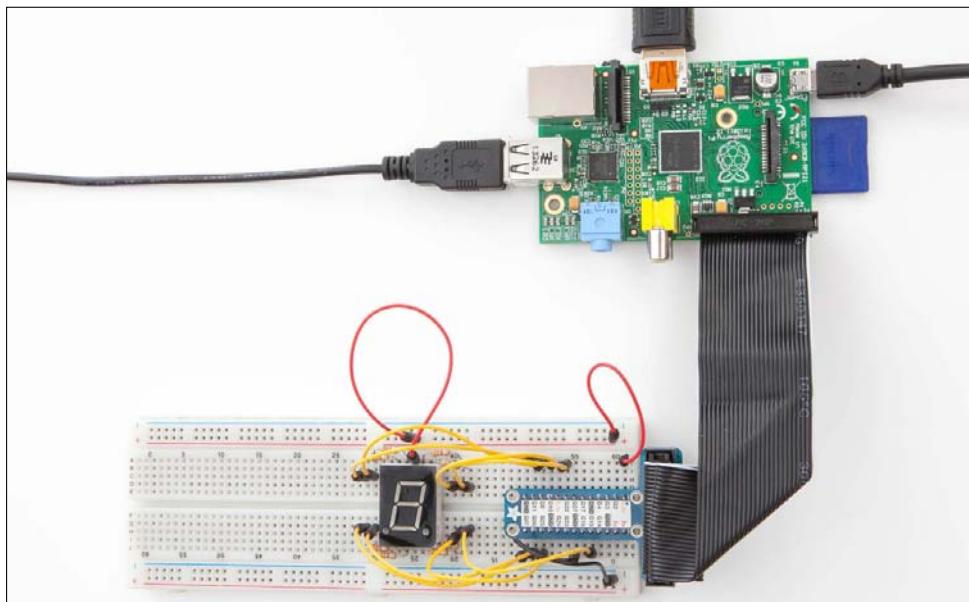


FIGURE 5-11 The 7-segment LED is wired up and ready for testing on the Raspberry Pi.

Once you have tested your display and made sure it's all working properly, the next step is to connect your display up to your computer's GPIOs so that you can control each LED from your Python program. For each resistor, run a wire from the unconnected leg of that resistor and connect it to a different GPIO—follow the diagram in Figure 5-12 for the Raspberry Pi, or use Figure 5-13 if you have an Arduino. If you discovered with your earlier testing that your display works differently from mine, use the notes you took earlier when you touched a wire on each resistor leg in turn, and use this information to wire each segment to the correct GPIO pin of your computer. If you don't do this, you will get some very strange looking patterns on your display when you run your Python program!

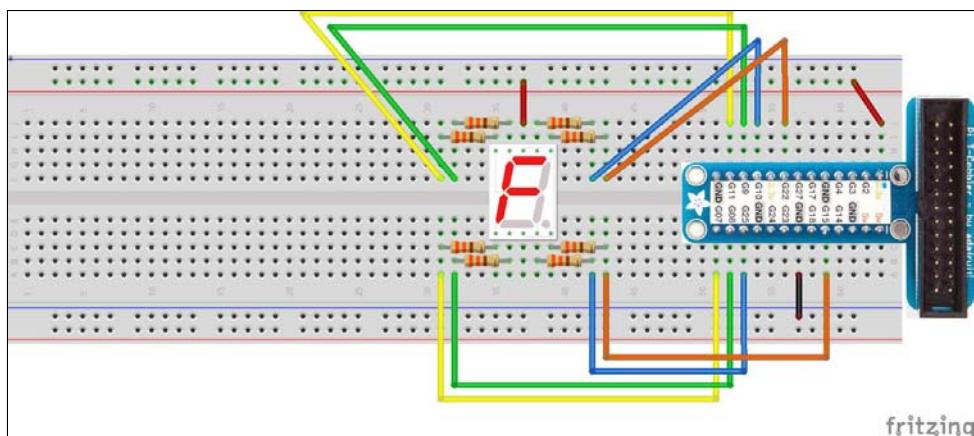


FIGURE 5-12 Wiring diagram for the 7-segment display on the Raspberry Pi

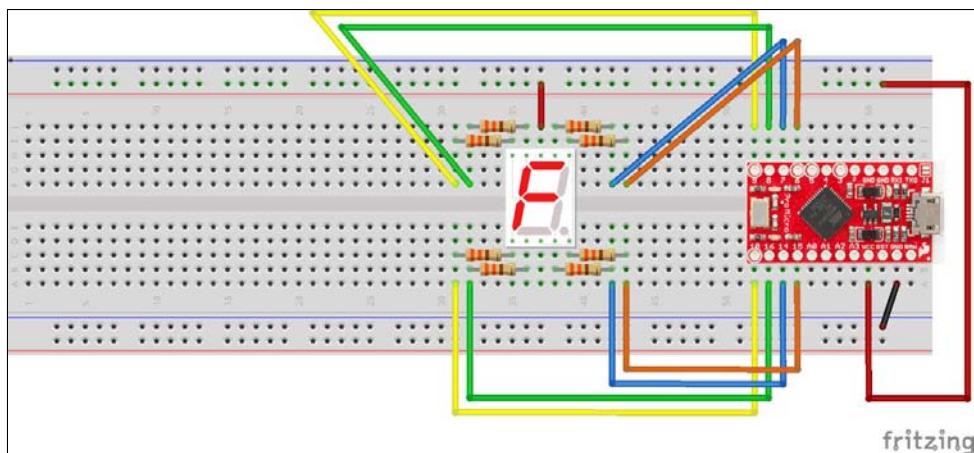


FIGURE 5-13 Wiring diagram for the 7-segment display on the Arduino

Writing Python to Drive the 7-Segment Display

Now that you have tested your 7-segment display and have wired it to all eight GPIO pins of your computer, you are ready to write some Python programs to control it:

1. Create a new program by choosing File ➤ New File and, and save it as `testDisplay.py`.
2. Import all the necessary modules and set some constants for each of the eight GPIO numbers:

For the Raspberry Pi:

```
import RPi.GPIO as GPIO
LED_PINS = [10,22,25,8,7,9,11,15] # order important
```

For the Arduino:

```
import anyio.GPIO as GPIO
LED_PINS = [7,6,14,16,10,8,9,15] # order important
```

- Set the GPIO numbering mode correctly:

```
GPIO.setmode(GPIO.BCM)
```

- Choose your type of display (I used a common anode display in my circuits):

```
ON = False # False=common-anode, True=common-cathode
```

- Write a loop that sets all eight pins to be outputs:

```
for g in LED_PINS:
    GPIO.setup(g, GPIO.OUT)
```

- Each LED of your display will be either on or off. A really nice way to build up a pattern for the LED display, is to store eight **True** or **False** values in a list, each index in that list represents a different LED in the display. Remember you used lists and indexes in Adventure 4? It's just the same as that. The list below will turn on segments A,B,C,D,E,F,G (because the value **True** is used), but the decimal point (DP) will be off because a **False** is stored in the last position. Index **[0]** in the list is for segment A, index **[1]** is for segment B, and so on.

```
pattern = [True, True, True, True, True, True, True,
           False] # ABCDEFG (no DP)
```

- Write a loop that turns on all of the pins in your chosen pattern:

```
for g in range(8):
    if pattern[g]:
        GPIO.output(LED_PINS[g], ON)
    else:
        GPIO.output(LED_PINS[g], not ON)
```

- Now your program will have to wait for a keypress before it finishes; otherwise all the LEDs will go off when the program stops and you won't see anything. Using the **raw_input()** function will wait for you to press Enter on the keyboard before continuing to the next line of the program.:.

```
raw_input("finished?")
```

- Finally, tell the program to clean up the GPIO before the program finishes:

```
GPIO.cleanup()
```

Run the program and see what it does. Remember that on the Raspberry Pi you have to run this program from within a LXTerminal window with `sudo python testDisplay.py`.

CHALLENGE



Try to work out the values to put into the pattern list for each of the digits 0 to 9. Remember that the True/False values in the pattern list are ordered A,B,C,D,E,F,G,DP—test them on your display. How many different letters of the alphabet can you design for your 7-segment display? You can find some useful examples of 7-segment patterns at this Wikipedia page: http://en.wikipedia.org/wiki/Seven-segment_display

Using a Python Module to Control the Display

You might have discovered that there are many different useful patterns that you can display to a 7-segment display. To make it easier to use all of these patterns, I have written a small Python module called `seg7.py`, which is included in the starter kit in the `anyio` folder. You can use and modify this module as much as you like and add it to your own games and programs. By following these steps you will see that it is very easy to link it into an existing program:

1. Start a new program by clicking File ➔ New File and save it as `testDisplay2.py`.
2. Import my display module, and also the time module so you can add delays.

```
import anyio.seg7 as display  
import time
```

3. Copy in every line from your `testDisplay.py` from the start, up to and including the `ON=False` line.
4. After this, set up the display module by giving it access to your GPIO pins and giving it a list of the pin numbers to use for each of the segments:

```
display.setup(GPIO, LED_PINS, ON)
```

5. Write a game loop that counts repeatedly from 0 to 9. The reason we have done this as a loop is so that you can leave the loop running indefinitely and wiggle the wires around if some of the segments don't seem to work, without having to keep re-running your program.

```

try:
    while True:
        for d in range(10):
            display.write(str(d))
            time.sleep(0.5)
finally:
    GPIO.cleanup()

```

Save and run the program. See what happens? The display should count from 0 to 9 repeatedly. Remember that on the Raspberry Pi you have to use `sudo python testDisplay2.py` from a LXTerminal window.

Instead of `time.sleep(0.5)` in the above program, you could replace this with a longer delay (2 seconds), or even with a `raw_input()` so that the displayed pattern will stay solid while you wiggle the wires to make sure they are all in the correct place.



CHALLENGE

Open up the `seg7.py` program, which is stored in the `MyAdventures/anyio` folder, and follow the instructions in the file to add new symbols. See how many of the letters of the alphabet you designed in the earlier challenge you can code up inside this module now. What words can you write on the display, one letter at a time? Can you write your name on the display one letter at a time? Which letters of the alphabet are missing?



Making a Detonator

In the final project in this adventure, you will build a big red detonator button. When you press the button there will be a countdown on the 7-segment display to give your player time to run away from the blast, and then a huge crater will appear at the point where your player was standing when you pressed the button. This will be a handy addition to your toolbox as a quick way to clear some space as you move around your Minecraft world and build things. This project introduces you to another type of GPIO, called an input, that senses when a button is pressed—and you'll put your 7-segment display to good use too!

To watch a tutorial on how to build and play the detonator game, visit the companion website at www.wiley.com/go/adventuresinminecraft and choose the Adventure 5 Video.





In this project, if you are using a PC or a Mac, use the `anyio` package. This package is provided in the starter kit.

Wiring Up a Button

The first step in this project is to wire up a button that you can press to set off the detonator. Choose a big red button to make it look really electrifying!

The button you're going to use here is a non-locking, push-to-make button. This means that the button does not normally make an electrical circuit—it is only when you press it that it completes the circuit. The button does not stick down, however, so as soon as you remove your finger the circuit is broken again.

Figure 5-14 shows the circuit diagram for the Raspberry Pi, and Figure 5-15 shows the circuit diagram for the Arduino. You can either follow these circuit diagrams, or follow the numbered steps, whichever you prefer.

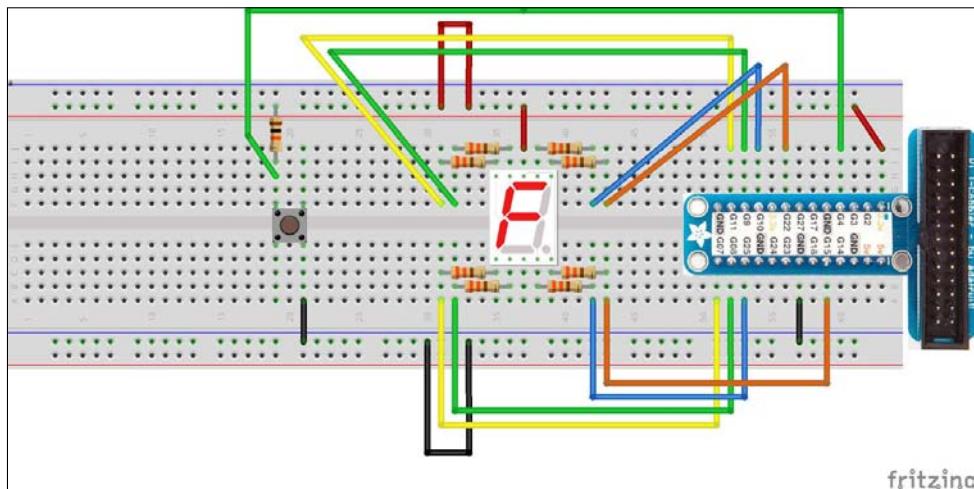


FIGURE 5-14 Circuit diagram of a button connected to the Raspberry Pi

1. Push your button into some spare space on the breadboard. The button that I used (see Figure 5-16) has four pins on it, and it fits nicely across the midway point of the breadboard. Pressing the button joins the pins on the left-hand side together with pins on the right-hand side, as long as you have fitted the button with the pins showing at the top and bottom as shown in Figures 5-14 and 5-15.

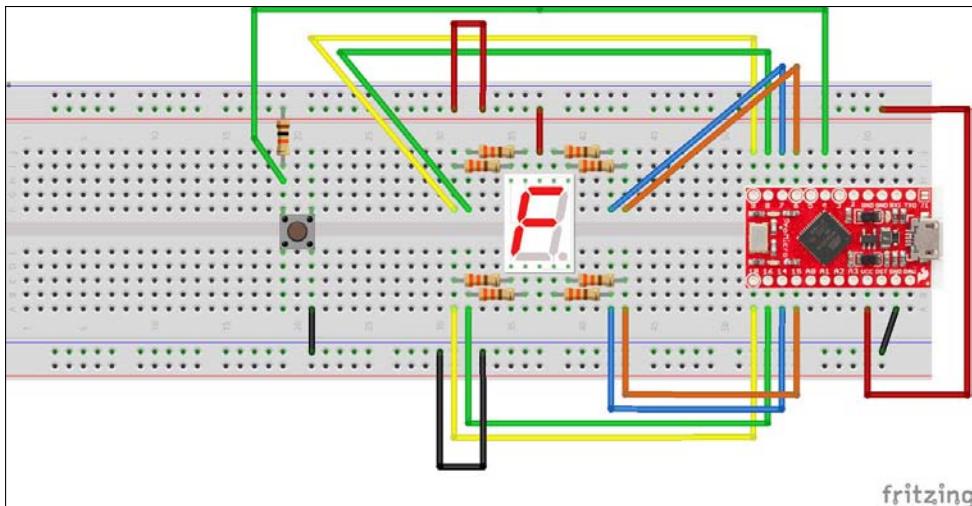


FIGURE 5-15 Circuit diagram of a button connected to the Arduino

2. You need a resistor to pull up the button input to 3.3 volts. If you don't fit a resistor, your button will generate spurious presses and your detonator will keep going off all the time! (Read *Digging Into The Circuit* to find out more about why this resistor is necessary.) Fit the 10K pull-up resistor (with the coloured bands: brown, black, orange) so that one leg is connected to the left pin of the button, and the other leg is connected to the 3.3 volt power rail at the top of the breadboard.
3. Run a wire from the junction between the resistor leg and the button, and connect the other end of the wire to GPIO number 4.
4. Run a wire from the bottom right pin of the button, and connect the other end of the wire to the 0-volt power rail at the bottom of the breadboard. You can read *Digging Into The Circuit* to understand more about how this wiring works, but all buttons are normally wired up to computers in this way. Figure 5-16 shows a button wired to a GPIO pin.

There are many different types of push button. If you use the same sort of button as I did for this project, you will notice that the four pins are slightly bent so that the button fits nicely into a printed circuit board, ready for soldering. Be very careful when you push the button into the breadboard so that the pins don't break off. Push very evenly and firmly all over the button and it should fit. If you have a small pair of pliers, you could use them to squash the pins so that they are completely flat, and they will then fit into the breadboard more easily.



DIGGING INTO THE CIRCUIT

When you wired your button, you fitted a resistor to it. This resistor is called a pull-up resistor because it pulls the voltage on the pin of the button “up” to 3.3 volts.

When the button is not pressed (and no connection made), the wire that connects the GPIO to this point in the circuit is now at 3.3 volts, which is the voltage the GPIO pin will interpret as a digital “1.” When you press the button, the internal mechanics of the button make a circuit between the left-hand and right-hand pins, and the GPIO wire is “pulled hard down” to 0 volts via the 0-volt power rail. In this condition, the GPIO pin will interpret this as a digital “0.” The resistor is quite a high value (10K ohms, which is 10,000 Ohms) because otherwise the button will short together the 3.3 volt power rail and the 0-volt power rail, and your computer might reboot! This is not recommended, as it can in some cases cause damage to your computer. 10K is a short-hand form that engineers use to refer to a 10,000 Ohm resistor.

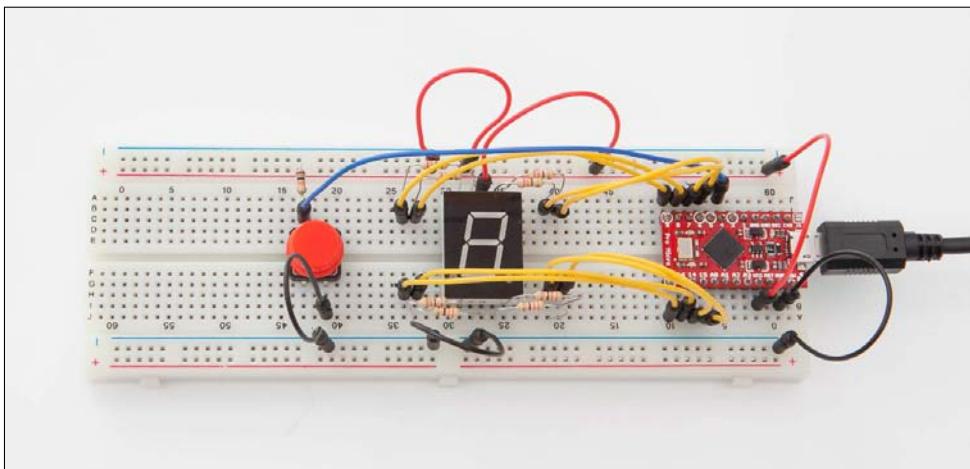


FIGURE 5-16 The button on this breadboard is wired to a GPIO on the Arduino and has a pull-up resistor.

Writing the Detonator Program

Now that you have wired your button, the last step is to write the Python program. This program will monitor the state of the button each time round the game loop, and when it senses a digital “0” on the GPIO pin (in other words, when the button is pressed), it will count from 5 down to 0 on the 7-segment display and then blow a massive crater in the Minecraft world.

1. Start a new program by choosing File ➔ New File and save it as `detonator.py`.
2. Import the necessary modules:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import time
import anyio.seg7 as display
```

3. Configure the GPIO settings for your computer:

On the Raspberry Pi:

```
import RPi.GPIO as GPIO
BUTTON = 4
LED_PINS = [10,22,25,8,7,9,11,15] # order important
```

On the Arduino:

```
import anyio.GPIO as GPIO
BUTTON = 4
LED_PINS = [7,6,14,16,10,8,9,15] # order important
```

4. Setup the GPIO for the button so it is an input, and configure the display GPIOs.

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUTTON, GPIO.IN)
ON = False # False=common-anode, True=common-cathode
display.setup(GPIO, LED_PINS, ON)
```

5. Connect to the Minecraft game.

```
mc = minecraft.Minecraft.create()
```

6. Write a function that drops a block of TNT to mark where the bomb will go off, counts down to zero, then blows a big crater where the TNT block was. The following code builds the TNT block slightly to the side of the player so it doesn't land on top of him! Note that the crater is 20 blocks in size (10 to the left and 10 to the right of your player) and this size is set by the calculations done in `setBlocks()` here:

```
def bomb(x, y, z):
    mc.setBlock(x+1, y, z+1, block.TNT.id)
    for t in range(6):
        display.write(str(5-t))
        time.sleep(1)

    mc.postToChat("BANG!")
    mc.setBlocks(x-10, y-5, z-10, x+10, y+10, z+10,
                block.AIR.id)
```

7. Write the main game loop so that it waits for a button press, then sets off a bomb. Remember that your button connects to 0 volts when it is pressed, which is why you have to compare the GPIO against `False` here to detect a button press.

```
try:  
    while True:  
        time.sleep(0.1)  
        if GPIO.input(BUTTON) == False:  
            pos = mc.player.getTilePos()  
            bomb(pos.x, pos.y, pos.z)  
finally:  
    GPIO.cleanup()
```

Save your program and run it. Remember that on the Raspberry Pi, you have to use `sudo python detonator.py` from within an LXTerminal window.

Run to somewhere in the Minecraft world and press your big red button—then run for your life! Figure 5-17 shows the aftermath of the explosion—a massive crater.

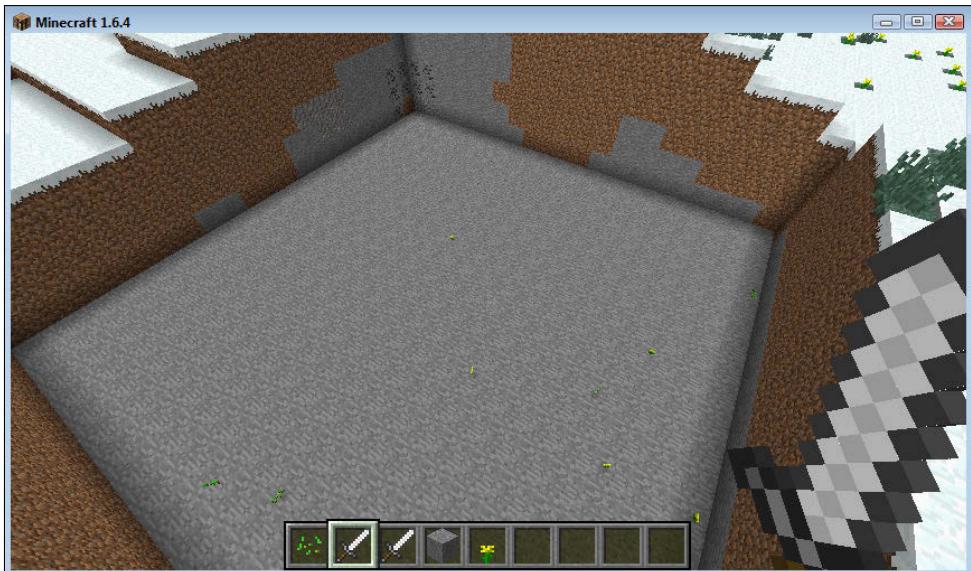


FIGURE 5-17 A crater blown into the Minecraft world

The size of the crater is set by the numbers inside the `setBlocks()` statement. Do you think this is a good way of setting the size of the crater? What if you wanted to make your crater twice the size, how much of the program would you have to change to make this work? Don't you think I'm being a little lazy here?! Why don't you try and improve on my program, so that it is very easy to change the size of the crater?



CHALLENGE

To make the detonator more exciting, try adding some geo-fencing code like in Adventure 2 to detect when your player is still in the blast zone. When the bomb goes off, if you are in the blast radius, use a `mc.player.setTilePos()` to catapult your player up into the sky.



CHALLENGE

Add three more buttons to your circuit, in the same way that you added the first button. Then write some more Python code to make those buttons do different things, like post random messages to the Minecraft chat, teleport your player to a secret location, build blocks of different types where your player is standing, or even build a whole house where you are standing. The following GPIO numbers are safe to use for your three extra buttons.



button	Raspberry Pi	Arduino
BUTTON2	14	5
BUTTON3	23	2
BUTTON4	24	3

Leave your breadboard all wired up when you have finished this Adventure. Martin will use this exact setup of electronic components again in the final Adventure 9 game, and I will use it again in the bonus chapter that you can download from the Wiley website.



Quick Reference Table

Configuring GPIO pins	Reading and writing GPIO pins
<pre>import RPi.GPIO as GPIO # RaspberryPi import anyio.GPIO as GPIO # Arduino GPIO.setmode(GPIO.BCM) GPIO.setup(5, GPIO.OUT) # output GPIO.setup(5, GPIO.IN) # input</pre>	<pre>GPIO.output(5, True) # set on (3.3V) GPIO.output(5, False) # set off (0V) if GPIO.input(6) == False: print("pressed")</pre>
Safely cleaning up after using the GPIO	Using the 7-segment display module
<pre>try: do_something() # put code here finally: GPIO.cleanup()</pre>	<pre>import anyio(seg7 as display LED_PINS = [10,22,25,8,7,9,11,15] ON = False # common-Anode ON = True # common-Cathode display.setup(GPIO, LED_PINS, ON)</pre>
Writing characters to the 7-segment display	Other 7-segment display functions
<pre>display.write("3") display.write("A") display.write("up")</pre>	<pre>display.setdp(True) # point on display.setdp(False) # point off display.clear() display.pattern([1,1,1,1,1,1,1,1])</pre>

Further Adventures in Electronic Circuits

In this adventure you have linked the Minecraft world to the physical world and expanded your horizons into the fascinating world of physical computing by sensing and controlling things in the real world. You have used this new knowledge to flash LEDs, write patterns on 7-segment displays, and sense when buttons have been pressed. But most importantly, you have escaped the confines of the Minecraft sandbox world. With this new found knowledge you can make your own amazing game controllers and display devices!

- There wasn't enough space in this book to do too many games with electronic circuits, but when I started to think of ideas of games, I couldn't stop coming up with new ones! The Minecraft Lift in the bonus chapter was one of the games I just had to let you play with, which is why it is available for download from the companion Wiley website—do try it out. I had hours of fun whizzing up and down the lofty heights and murky depths of the Minecraft world with the lift

and building whole tower blocks around it! One day, I want to build a virtual Shard building (<http://www.the-shard.com>) around the lift, or will you beat me to it?!

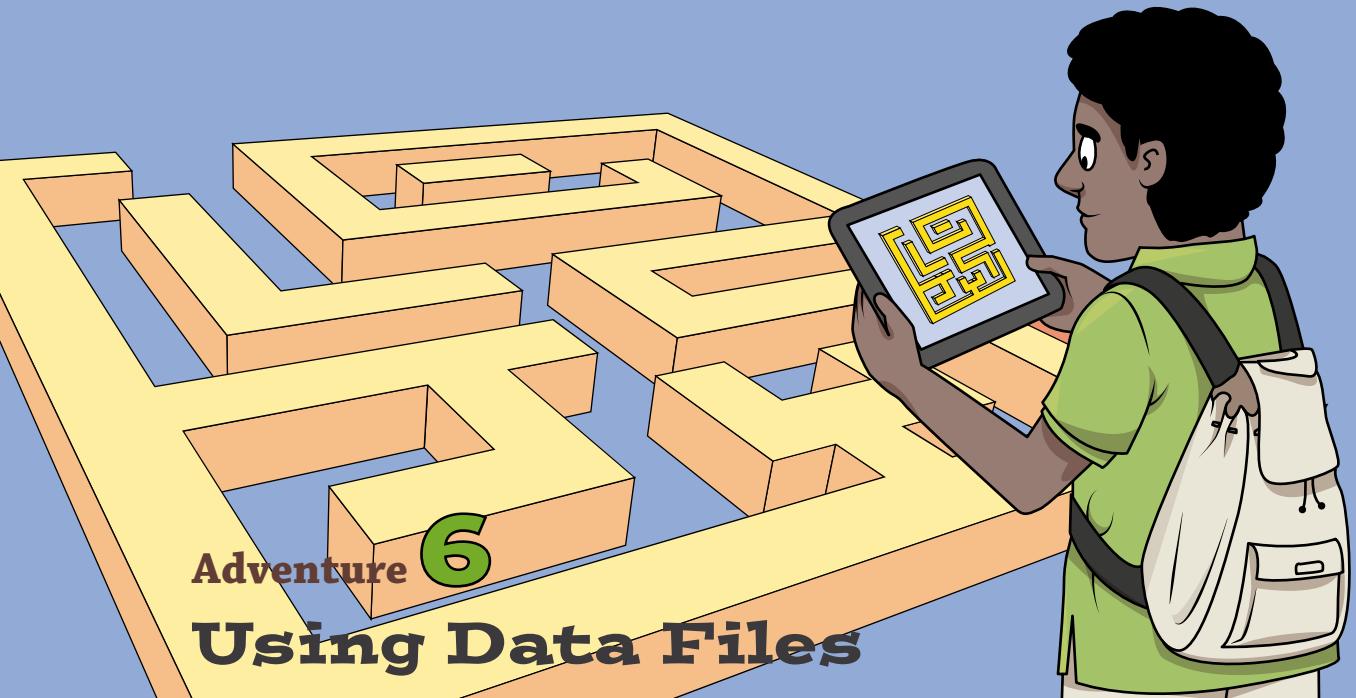
- One of the children at our Raspberry Pi club had a fantastic idea of building an adventure map out of a piece of card and pushing LEDs through holes in it. The idea was to draw a map on the card that represented the Minecraft world that they built, and then each of five LEDs pushed through the card would light up when you completed some challenge in that part of the world. I'd really love to see that game designed, and I hope one day to find a You-Tube video of someone who has built it all—will you be the first person to build that fantastic game?
- Someone worked out how to connect an electronic accelerometer to the computer so that it could sense movements of his real arm and translate these into movements of a robot arm. Look at his video here: http://www.youtube.com/watch?v=_KGc9vlOrNk. See if you can find a way to link up an accelerometer to your computer so that you can control your Minecraft games just by moving your arms! When you've got that working, why not show it off as a project at a local maker faire to inspire others to learn about the fascinating world of Minecraft electronics?

Achievement Unlocked: You have escaped the confines of the Minecraft virtual world into the real world! You're a pioneer in the fascinating new industry of Minecraft Electronics. The boundaries of your gaming experiences are now limitless!



In the Next Adventure...

In Adventure 6, you will learn how to read data from files in order to build large, complex structures like Minecraft mazes automatically. You will use your new skills to build a huge 3D duplicating machine that can duplicate large objects all over the Minecraft world. Trees will never again be safely rooted to the ground!



IT'S WHEN YOU can start to process large amounts of data that computer programming gets really exciting. Your program then becomes a set of rules that govern how that data is read, processed and visually represented—the data becomes the really important part.

In this adventure, you will first look at how to create text files that define mazes. The mazes will then be automatically built in the Minecraft world for you and your friends to solve; you will wonder how such a small Python program can build something so big in the Minecraft world.

You are then going to develop this simple idea into a complete virtual 3D scanning and printing facility called the duplicator room. Anything you build inside the duplicator room can be saved to a file and recalled later, and teleported to any location in your Minecraft world, and even loaded into Minecraft worlds on other computers! You will be able to build up your own library of objects and build them in Minecraft so quickly that your friends will all want duplicating machines of their own!

Reading Data from a File

Computer programs are generally quite unintelligent—they repeatedly follow instructions that you give them. If you don't change any input to your program, it will do exactly the same thing every time you run it. The Minecraft programs you have been making have been quite interactive because they change what they do depending on what happens in the game world, which is the input to the program.

Interesting Things You Can Do With Data Files

Another interesting way to make computer programs do different things every time you run them is to store the input data in a text file and have the computer read that file when it starts. You can then create any number of text files and even have a menu so you can choose which file to open depending on what you want to do with the program. This is called a data-driven program, because it is mostly the data in the external text file that defines what the program does.

If you think about it, many of the programs you already use on your computer use data files. The word processor you use to type up your homework stores your homework in a data file; when you take a photo with a digital camera it is stored in a data file; and your image editor program reads that photo from the data file. Even Minecraft uses data files behind the scenes for tasks like saving and loading the world, and for the texture packs used to build all of the different blocks in the world from. Using data files with a program is a much more flexible way of working, because it means that the program can be used for lots of different things without forcing you, the user, to modify the program every time you want it to do something slightly different. You can also share those data files with your friends, if they have the same programs as you.



An early example program that I wrote that demonstrates the use of data files with Minecraft is a program on my blog, called the Minecraft BMP builder (<http://blog.whaleygeek.co.uk/minecraft-pi-with-python/>), which reads an image from a bitmap picture file (BMP file) and builds it block by block inside the Minecraft world. With it, I built a huge Raspberry Pi logo that is so big it looks a bit foggy when you look up at it! This program can read any BMP image file and then build that image out of blocks inside the Minecraft world, without any need to modify the program.

Your first step in writing a data-driven program is to learn how to use Python to open and read text files from the computer filing system.

Making a Hint-Giver

To learn how to open and read text files, you are going to write a simple hint-giver program. This program will read a file you have prepared of useful Minecraft hints and tips, and display one of the hints on the Minecraft chat at random intervals. You're going to need a text file with tips in it, so your first task is to create this file. You can create this file in the normal Python editor, just like you do with your Python programs.

Start up Minecraft, IDLE, and if you are working on a PC or a Mac, start up the server too. You should have had a bit of practice with starting everything up by now, but refer back to Adventure 1 if you need any reminders.

1. In IDLE, create a new text file by choosing File ➔ New File from the menu. Save the file as `tips.txt`.
2. Now list four or five tips, using one line of text for each tip. Here are a few example Minecraft tips, but it's much more fun if you make up your own. Make sure you press the Enter key at the end of each line, so that this creates a **newline** inside the file for each line. (You will find out more about newlines later in this adventure.)

Build yourself a house before the mobs come and get you
Use flowing water to fill underwater channels
Build up with sand then knock out the bottom block
Use both first-person and third-person views
Double tap space bar to fly into the sky

3. Save this file. You don't run this file, as it is not a Python program but a text file you're going to tell your Python program to use.

A **newline** is a special invisible character that the computer uses to mark the end of a line of text.

It is also sometimes called a *carriage-return*, which is historic from the days of mechanical typewriters. Whenever you wanted to type on the next line of the page, you would use the carriage return, which would make the *carriage return* to the left of the page and then down by one line.



Don't use commas in your `tips.txt` file, or in any messages that you ever use with `mc.postToChat()`. There is a small bug in the Minecraft API, where any comma inside the message text causes the message to be cut off at that position.



Now you have a data file with tips in it—in other words, input data—it's time to write the program that processes that input data. This program will read a random tip from the file and display it after a random interval on the Minecraft chat. Then, while you are playing your game, helpful tips will pop up on the chat.

1. Start a new file with File→New File and then save it as `tipChat.py`.

2. Import the necessary modules:

```
import mcpi.minecraft as minecraft
import time
import random
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. Set a constant for the filename, so that you can easily change it to read a different file later:

```
FILENAME = "tips.txt"
```

5. Open the file in read mode (see the following Digging into the Code for a fuller explanation of this line):

```
f = open(FILENAME, "r")
```

6. Read all the lines of the file into a list called `tips`. Remember that you have already used lists in Adventure 4 (magic bridge builder) and Adventure 5 (electronics):

```
tips = f.readlines()
```

7. Close the file when you have finished with it. It is good practice to always close a file when you have finished with it, because when the file is open it cannot be used by other programs, such as the editor you used to enter the text into the file in the first place:

```
f.close()
```

8. Now write a game loop that waits a random length of time between three and seven seconds:

```
while True:
    time.sleep(random.randint(3, 7))
```

9. Now tell the program to choose a random message from the `tips` list and display it on the chat. You need to use the `strip()` function, in order to strip out unwanted newline characters. You'll be looking at newline characters later on, so don't worry too much about understanding this for now.

```
msg = random.choice(tips)
mc.postToChat(msg.strip())
```

Save and run your program. What happens? As you walk around the Minecraft world and play your game, every so often a tip pops up on the chat, just like in Figure 6-1. Note how the Minecraft chat builds up messages on the screen, and gradually they will disappear over time.



FIGURE 6-1 Random tips pop up on the chat as you play Minecraft.

CHALLENGE

Add more Minecraft tips to your `tips.txt` file based on your experience playing Minecraft. Give your `tips.txt` and your `tipChat.py` program to a friend who is learning Minecraft and let them use it to quickly learn how to do amazing things in the game. You could even write a whole series of `tips.txt` files with different topics and different amounts of detail depending on the ability of the player, and publish them on a small website for others to use along with your `tipChat.py` program.



DIGGING INTO THE CODE

In the `tipChat.py` program, a little bit of magic happens in the following line:

```
f = open(FILENAME, "r")
```

The `open()` function opens the file named in the `FILENAME` constant—but what does that `"r"` at the end mean? When you open a file, you have to tell the `open()` function what level of access you want to that file. In this case, the `"r"` means that you only want to read the file. If you accidentally try to write to the file, you will get an error message. This protects your files by preventing accidental damage. If you want to open a file and write to it, you can use a `"w"` instead (or if you want to read and write a file at the same time, use `"rw"`).

continued

continued

Secondly, what is `f`? `f` is just another variable, but it stores a file handle. A file handle is just something to “grab hold of” the file with—it is a kind of virtual reference to the real file inside your computer’s filing system. Whenever you want to do anything with the open file, use `f` to refer to it. This is very handy, because like any other variable you have used before, you can have multiple file variables to allow multiple files to be open at the same time like this:

```
f1 = open("config.txt", "r")  
f2 = open("levels.txt", "r")  
f3 = open("score.txt", "rw")
```

The variables `f1`, `f2` and `f3` can be used in the rest of your program to correctly identify which file you want to read from or write to.

Building Mazes from a Data File

Now that you know how to make your programs read from data files, you are ready to get a bit more adventurous. You know from earlier adventures that it is easy to place blocks in the Minecraft world. What if you could build blocks using lists of blocks stored in a data file? That’s exactly what you are going to do here: you are going to build a complete 3D maze in Minecraft, where the maze data is stored in an external file! But first, you need to decide how the maze data will be represented inside the file.



VIDEO

To watch a tutorial on how to build and play your maze game, visit the companion website at www.wiley.com/go/adventuresinminecraft and choose the Adventure 6 Video.

The mazes that you design will be 3D because they are built in the Minecraft 3D world, but really they are just 2D mazes built out of 3D blocks—there is only one layer to your maze. This means that the data file needs to store x and z data for each block in the maze, so it will be rectangular.

You need to decide how blocks themselves are represented in the data file. You will be using walls and air. To keep things simple, use a 1 to represent a wall and a 0 to represent air. You can always change the block types inside your Python program that are used for walls later.

Understanding CSV Files

For this program, you use a special type of text file called a **CSV** file, to represent your mazes as files in your computer’s filing system.



A **CSV** file is a comma-separated values file. This is where values are stored in a simple text file, separated by commas. It is possible to represent a simple table or database inside a CSV file. Each row or record in the database is represented by a line in the file, and each field or column is represented by some text on that line separated by commas. Some CSV files use the first line of the file to store the headings or field names. All popular spreadsheet and database programs can import and export data in CSV format.

CSV is a very simple and widely used format. Here is a sample of a CSV file that stores part of a table from a database with details about Minecraft gamer characters. In this CSV file, the first row (or line) in the file has the names of the fields, and all other lines have data separated by commas:

```
Name,Handle,Speciality
David,w_geek,Coding in Python
Roma,physics_gurl,Designing big buildings
Ryan,mr_teck,Minecraft robots
Craig,rrrrrrrr,TNT expert
```

This CSV file has one header row, which has three field names called **Name**, **Handle** and **Speciality**. There are four data rows, each with three fields of data.

When designing complex structures in software, sometimes it is useful to use other tools to help you. As a software engineer, a spreadsheet is one of the tools that I regularly find a use for when designing and representing data. Spreadsheets provide a fantastic way to represent tables of data that can be exported in a CSV file format, and then loaded into programs. You could try designing a maze in your favourite spreadsheet program by setting all the columns to be very narrow, and using formulas to display a white square whenever you put a **0** and a yellow square whenever you put a **1**. Once you have visualised your maze this way, export it as **maze.csv** and run your program and see if you and your friends can solve it!



For your maze data file, you won't need a header row, because each column in the table represents the same type of data; that is, it stores a number 0 for a space and a number 1 for a wall. You can download this sample maze file from the companion website, but here it is if you want to type it in:

1. Create a new text file by choosing File ➔ New File. Use File ➔ Save As and name the file **mazel.csv**.

- Type in the following lines very carefully, making sure there are exactly the same number of columns on each line. If you look very carefully at this data you should already be able to see the structure of this maze! There are 16 numbers per line and 16 lines, so perhaps use a pencil to tick off each line as you type it in so you don't lose your place:

```
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1  
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1  
1,1,1,1,1,1,1,1,0,1,0,1,1,0,1  
1,0,0,1,0,0,0,1,0,1,0,1,0,0,0,1  
1,1,0,1,0,1,1,0,0,0,0,0,1,0,1,1  
1,1,0,1,0,1,1,1,1,1,1,1,0,1,1  
1,1,0,0,0,1,1,1,1,1,0,0,0,0,1,1  
1,1,1,1,1,1,0,0,0,0,0,1,1,1,1  
1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1  
1,0,1,1,1,1,0,0,0,0,1,1,1,1,1  
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1  
1,0,1,1,1,1,1,1,1,0,1,1,1,1,1  
1,0,1,0,0,0,0,0,1,0,0,0,0,0,0,1  
1,0,1,0,1,1,1,0,1,1,1,1,1,1,0,1  
1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1  
1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1
```

- Save this file. You will use it once you have written the Python program that reads and processes the data in this file.

Building a Maze

Now that you have a data file describing your maze, the final step is to write the Python program that reads data from this file and builds the maze in Minecraft using blocks.

- Start a new file with File→New File and then use File→Save As to save it as `csvBuild.py`.
- Import the necessary modules:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block
```

- Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

- Define some constants. The `GAP` constant is the block type that is used in the spaces. This will normally be air but you could experiment with different block types here to make the maze interesting. The `WALL` constant is the block type that the walls of the maze will be built with, so choose this carefully as some block types won't work very well as walls. The `FLOOR` constant is used to build the layer that the maze stands on:

```
GAP = block.AIR.id  
WALL = block.GOLD_BLOCK.id  
FLOOR = block.GRASS.id
```

Don't use **SAND** for the floor layer. If you do, depending on where the maze is built, the floor might start to fall away under your player's feet!



5. Open the file containing your maze data. The **FILENAME** constant is used so that it is easy for you to change the filename later to read in different maze files:

```
FILENAME = "maze1.csv"  
f = open(FILENAME, "r")
```

6. Get the player position and work out the coordinates of the bottom corner of the maze. Adding 1 to the x and z will make sure that the maze doesn't get built on top of your player's head. You could vary the y coordinate to build the maze in the sky if you wanted to:

```
pos = mc.player.getTilePos()  
ORIGIN_X = pos.x+1  
ORIGIN_Y = pos.y  
ORIGIN_Z = pos.z+1
```

7. The z coordinate will vary at the end of each line of data, so start it off at the origin of the maze. It will vary a little later on in the program:

```
z = ORIGIN_Z
```

8. Loop around every line of the file. See Digging into the Code, which explains what the **readlines()** function does. The **for** loop is actually looping through every line in the file, one by one. Each time round the loop, the **line** variable holds the next line that has been read from the file:

```
for line in f.readlines():
```

9. Split the line into parts, wherever there is a comma. See Digging into the Code, which explains what the **split()** function does. Remember that all of the lines that are part of the body of the **for** loop have to be indented one level:

```
data = line.split(",")
```

10. Look at this step very carefully: you now have another `for` loop. This is called a **nested loop**, because one loop is nested inside another. Your program will have to reset the `x` coordinate at the start of every new row of the maze, so this has to be done outside of the `for` loop that draws a whole row:

```
x = ORIGIN_X  
  
for cell in data:
```

11. Each number read in from the row is actually read in as a piece of text (a string), so for this program to work, you will have to put `" "` quotes around the number. This `if/else` statement chooses whether to build a gap or a wall, based on the number just read back from the CSV file. A 0 builds a gap, and anything else builds a wall. Make sure your indentation is correct here: the `if` statement is indented twice because it is part of the `for cell in data` loop, which is part of the `for line in f.readlines()` loop. The `b` variable here is useful as it makes the program a bit smaller and simpler. (Try rewriting this part without using the `b` variable and you will see what I mean!)

```
if cell == "0":  
    b = GAP  
else:  
    b = WALL  
mc.setBlock(x, ORIGIN_Y, z, b)  
mc.setBlock(x, ORIGIN_Y+1, z, b)  
mc.setBlock(x, ORIGIN_Y-1, z, FLOOR)
```

12. Update the `x` coordinate at the end of the `for cell` loop. This must be indented so that it lines up with the previous `mc.setBlock()` as it is still part of the body of the `for cell in data` loop.

```
x = x + 1
```

13. Update the `z` coordinate at the end of the loop that processes each row of data (this must be indented one level only, as it is part of the `for line in f.readlines()` loop).

```
z = z + 1
```

Save your program, and double check that all the indentation is correct. If the indentation is wrong, the program will not work correctly and you will get some very strange-shaped mazes!

Run your program and a fantastic (and quite hard to solve) maze will be built in front of you. Walk around the maze and see if you can solve it without breaking down any walls or flying. Figure 6-2 shows what the maze looks like from ground level.



FIGURE 6-2 A ground level view of the maze inside Minecraft

If you get stuck, you can always cheat a bit and fly up into the sky to get a 3D bird's-eye view of the maze, as shown in Figure 6-3.



FIGURE 6-3 A bird's-eye view of the maze inside Minecraft



A **nested loop** is a loop inside another loop. The first loop is called the *outer loop*, and the second loop is called the *inner loop*. Loops are nested in Python by indenting the second loop another level. You can nest loops inside loops inside loops any number of times.



If you accidentally leave some blank lines at the end of your data file, these will be read in and interpreted by your Python program. Fortunately it doesn't make your program crash, but you might see some redundant extra blocks at the end of the maze as a result.

DIGGING INTO THE CODE

In earlier adventures, you experimented with Python lists and learned that a list is just a collection of zero or more items stored in numbered slots. Many functions built into the Python language provide data in the form of a list, and you have just met two new ones in the maze builder program.

This line of code is the first one:

```
for line in f.readlines():
```

Remember that `f` is a file handle—something you can grab hold of to gain access to an open file. Usefully, the `file` type in Python has a `readlines()` function built into it. This simply reads every line of the file and appends (adds) each line to the end of the list. You may also remember from earlier adventures that, when it's given a list, the `for` statement in Python will loop through all the items in the list.

All the `for` statement does is read every line in the file into a list, and then loop through them one at a time. Each time around the `for` loop, the `line` variable (the loop control variable) holds the next line of data.

If you have a file with three lines in it like this:

```
line one  
line two  
line three
```

then the `f.readlines()` will return a list that looks like this:

```
['line one', 'line two', 'line three']
```

There is a second point in your program where you used another Python list but might not have realised it. It was in this line:

```
data = line.split(",")
```

The `line` variable holds a string of text that is the complete line read from the CSV file. All string variables have a built-in function called `split()` that will take that string and split it into a list. The `" , "` inside the brackets tells the `split` function which character to split on.

Imagine you have a `line` variable with three words separated by commas like this:

```
line = "one,two,three"  
data = line.split(",")
```

The result of this will be that the `data` variable will contain a list with three items in it, like this:

```
['one', 'two', 'three']
```

CHALLENGE

Design your own maze files in other CSV data files with lots of winding passages, dead ends and loops to make the mazes hard to solve. Challenge your friends to find their way through the maze. You might want to plant some random treasure (e.g. `DIAMOND_BLOCK`) throughout the maze and at the exit so that your friends can prove they have walked through the whole maze, or even build a huge maze that takes up most of the Minecraft world! How could you store the positions of the treasure in the maze file?



When I was designing the programs for this chapter, I had to modify the maze program slightly as sometimes it was very hard to solve. For example, what happens if you remove the line from the program that builds the floor? Try building the walls with some other block types, such as `CACTUS` or `WATER_FLOWING` and see what happens. I went through many versions of this program before finding block types that I was happy with for the walls and the floor!



CHALLENGE



There is a bug hidden in this program for you to find. If the last number on a line is zero, then for some reason the maze builder still builds a gold block wall there. Why do you think this happens? Try to fix this bug yourself. Hint: you already solved a similar problem in the `tipChat.py` program.



In this adventure you have been using the `readlines()` and `split()` functions to process CSV files. Python is a very large and established programming language and has lots of features already built in. You could have a look at the built in CSV reader module that is accessible with `import csv` as it is a more powerful method of what we have learnt here. However, I like to build programs up in small steps as it helps me understand how things work, which is why I have used `readlines()` and `split()` in this book.

Building a 3D Block Printer

Building mazes is great fun, but there is so much more you can do with data files now that you know the basics! Why stop at building with only two block types on a single layer? You are now going to use your maze program as the basis of your very own 3D printer and 3D scanner that will duplicate trees, houses—in fact, anything you can build in Minecraft—and “print” them all around the Minecraft world at the touch of a button! This is a block builder really, but I like to call this a 3D printer, because of the way you can sometimes see the blocks building up row at a time, just like how a computer printer prints onto a piece of paper row at a time, or how a 3D printing machine builds up structures layer at a time.

You’re getting quite good at building programs now and each new program you build is larger than the last one. Just like in the previous adventure, where you built your program out of functions, you are going to build this program up in steps.



The technique of writing the different features of a program as functions and then stitching them all together into a bigger program is a very common development technique used by professional software engineers. It relies on the principle that a big program is just a collection of small programs. Providing that the design of the overall program is correct from the start, the detail inside each of the support functions can be filled in gradually as you build and test your program. I always like to have demos handy so that if someone comes up to me and says, “Wow, what’s this great program you are writing?” I can very quickly show them something that works.

Hand-Crafting a Small Test Object to 3D Print

In your maze program, data was stored in a CSV file, where each line in the file represents a line of blocks inside the Minecraft world. Each column inside that line (separated by commas) corresponds to one block in that line of blocks. This is a 2D structure, because it stores a block for each x and z coordinate of a rectangle region. The mazes you have built here are really only 2D mazes, as they only have one layer. They are just built inside the 3D Minecraft world by using 3D blocks.

To build 3D shapes, you need to build up in the y dimension too. If you think of a 3D object as just multiple layers or slices, then your 3D program is not all that different from your maze program. All you need to do for a cube that is 5 by 5 by 5 blocks in size, is to store five layers of information.

There is a problem with this. Your Python program will not know how many lines to expect until it has processed the whole file. It could assume a certain size, but it would be nice to design a flexible CSV file format that can work with any size of object.

To solve this problem, you will add some **metadata** in the first line of the file that describes the size of the object.

Metadata is data about data. If the data in your file is the block types to build, then the metadata (data about that data) can be things like how big the object is, what it is called and who designed it. In the same way that the metadata for a photo stored on your computer describes the date and time it was taken and the name of the camera, the metadata in your 3D printer will describe other useful information about your 3D objects.



You are now going to create a sample object which is a tiny 3D cave so that you can test the 3D printing capabilities of your program. You can download this sample object as shown in Figure 6-4 from the companion Wiley website, or just type it in here like you did with your maze.

Make sure that you put blank lines between each layer of the object. They are there to help you see where each layer begins, but the Python program you write will expect them to be there and it won't work if you leave them out.



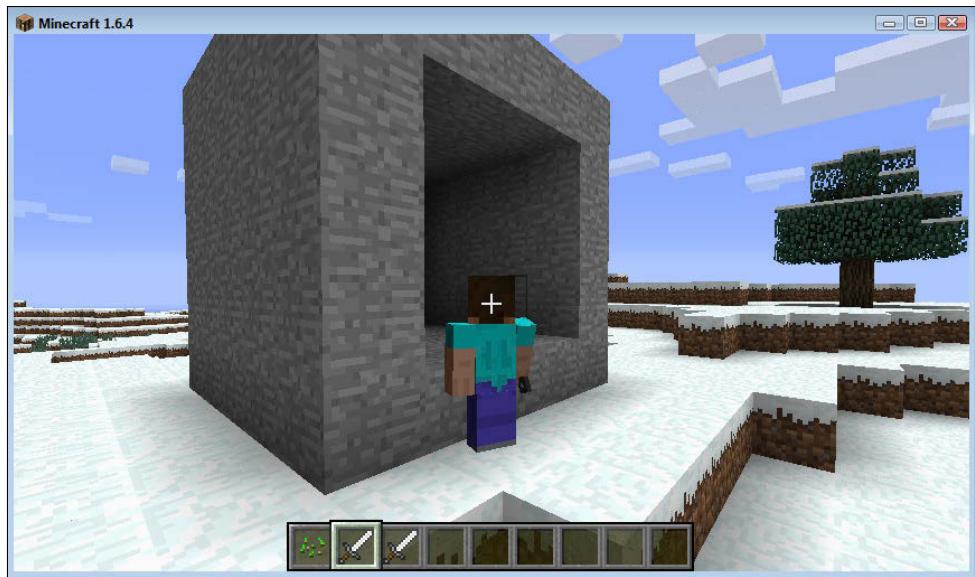


FIGURE 6-4 You build a small stone cave by writing a CSV data file.



Use copy and paste from the Editor menu as much as you can in these step to cut down the amount of typing you need to do and to reduce the possibility of you typing the wrong number of lines.

1. Start a new file with File→New File, and save it as **object1.csv**.
2. Type in the first metadata line that describes the size of the object. Your test object will be 5 by 5 by 5. The first number is the x-size (width), the second number is the y-size (height) and the third number is the z-size (depth):

5,5,5

3. Type in the first layer of the test object, making sure that you leave a blank line before the first line of data. You need all the block positions filled in on the bottom layer, which is why every number is set to a 1:

```
1,1,1,1,1  
1,1,1,1,1  
1,1,1,1,1  
1,1,1,1,1  
1,1,1,1,1
```

4. You are building a hollow cube with an open front, so put in three layers with that format. Make sure you put a blank line before these numbers:

```
1,1,1,1,1  
0,0,0,0,1  
0,0,0,0,1  
0,0,0,0,1  
1,1,1,1,1
```

5. Use copy and paste to put in two more identical hollow layers below this one, making sure you leave exactly one blank line between each square section of numbers.
6. Finally, put a solid top on the object. To do this, use copy and paste to copy the rows of data from step 3, again making sure that there is a blank line before the first row of numbers of this layer.

Save your file, but don't run it—you can't run it as it is not a Python program but a data file that will be used by the program you are now going to write.

Writing the 3D Printer

Now that you have written your test data, it is time to write the 3D printer program that will build the hollow cave inside the Minecraft world. This program is very similar to your maze program, but this one has three nested loops, one loop for each of the x, y and z dimensions.

This is probably one of the most detailed programs you have written while working through this book. Work through it carefully, step by step, and you will be amazed by the final results! If you get stuck, check your indents very carefully, and remember that all of the program listings for all of the programs in this book are downloadable from the companion Wiley website at www.wiley.com/go/adventuresinminecraft.



1. Start a new file with File→New File, and use File→Save As from the menu to save it as `print3D.py`.
2. Import the necessary modules:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. Create a constant that will be the name of your data file. This will be easy to change later on if you want to read different files with different objects in them:

```
FILENAME = "object1.csv"
```

5. Define a `print3D()` function. You will reuse this function later on in the final project so make sure you name it correctly:

```
def print3D(filename, originx, originy, originz):
```

6. Just as you did with your maze builder program, open the file in read mode and read in all the lines into a Python list:

```
f = open(filename, "r")
lines = f.readlines()
```

7. The first item in the list is at index 0. This is the first line of the file that holds the metadata. Split this line into parts by using the `split()` function. Then store those three numbers into the variables `sizex`, `sizey` and `sizez`. You have to use the `int()` function here because when you read lines from a file they come in as strings, but you will need them to be numbers so you can calculate with them later:

```
coords = lines[0].split(",")
sizex = int(coords[0])
sizey = int(coords[1])
sizez = int(coords[2])
```

8. Set a `lineidx` variable to remember the index of the line in the `lines[]` list that you are processing. Because your program will be scanning through the `lines` list reading out data for different layers of the 3D object, you can't use this as the loop control variable:

```
lineidx = 1
```

9. The first `for` loop scans through each of the vertical layers of the data read in from the file. The first few lines of the file are for the layer that will be built at the lowest `y` layer. You can put a `postToChat` here so that the progress of the printing process is visible inside Minecraft:

```
for y in range(sizey):
    mc.postToChat(str(y))
```

10. Skip over the blank line between each layer in the file by adding 1 to the `lineidx` variable. Remember that these blank lines are only in the file so that it is easier for people to read it, but your program has to skip over them.

```
lineidx = lineidx + 1
```

11. Start a nested `for` loop that reads out the next line in the file and splits it whenever there is a comma. Be careful of the indentation here; the `for` has to be indented two levels (one for the function and one for the `for y` loop) and the code inside this `for x` loop needs to be indented three levels.

```
for x in range(sizex):
    line = lines[lineidx]
    lineidx = lineidx + 1
    data = line.split(",")
```

12. Now start your third `for` loop that scans through each of the blocks in the line just read and builds them inside Minecraft. The `for z` loop is indented three levels and the body of the loop is indented four times so take very special care of the indentation; otherwise you will end up with some very strange-shaped objects!

```
for z in range(sizez):
    blockid = int(data[z])
    mc.setBlock(originx+x, originy+y, originz+z, blockid)
```

13. Finally, type in the lines of the main program, which are not indented at all. These lines read the player position and then ask the `print3D()` function to print your 3D object just to the side of where you are standing:

```
pos = mc.player.getTilePos()
print3D(FILENAME, pos.x+1, pos.y, pos.z+1)
```

Save your program and run it to see what happens! Run around to different locations inside Minecraft and run the program. Every time you run the program, a hollow stone cave should be built just near your player! Figure 6-5 shows how easy it is to build lots of stone caves very quickly.

CHALLENGE

Create some more CSV files with 3D objects in them, change the `FILENAME` constant in the `print3D.py` program and run it to stamp your objects all over your Minecraft world. As your objects become more sophisticated, think of some different ways that you could design them before entering the numbers into the file. You might find some way of drawing the objects in a spreadsheet program. Or perhaps you could even buy a big box of plastic construction bricks and design your objects in the physical world first, then take them apart layer by layer and enter the block numbers into your CSV file.





FIGURE 6-5 3D printing lots of stone caves inside Minecraft.

Building a 3D Block Scanner

Your 3D printer is already very powerful. You can build up a big library of CSV files for different objects you want to build, then whenever you need to, just run your `print3D.py` program with a different `FILENAME` constant to stamp the object all over your Minecraft world.

However, building larger and more complex objects will become very difficult if you have to enter all the numbers into the CSV file by hand. Minecraft itself is the best program for building complex objects—so, what if you could use Minecraft to create these CSV data files for you? Here, you are going to run up to a tree and hug it, then make an identical copy of the tree so you can duplicate it all over the Minecraft world. Fortunately, 3D scanning works a bit like 3D printing in reverse, so this not quite as hard as you might think.

1. Start a new file with File→New File, save it with File→Save As and call the pro-

gram `scan3D.py`.

2. Import the necessary modules:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. Create some constants for the name of the CSV file you want to scan to and the size of the area that you want to scan:

```
FILENAME = "tree.csv"
SIZEX = 5
SIZEY = 5
SIZEZ = 5
```

5. Define a `scan3D()` function. You will use this function in a later program, so make sure it is named correctly:

```
def scan3D(filename, originx, originy, originz):
```

6. Open the file, but this time open it in write mode using the "`w`" file mode inside the `open()` function:

```
f = open(filename, "w")
```

7. The first line of the file has to contain the metadata, so write the x y and z sizes in the first line. See Digging into the Code to understand what the "`\n`" at the end of the line is and why you need it.

```
f.write(str(SIZEX) + "," + str(SIZEY) + "," + str(SIZEZ)
+ "\n")
```

8. The scanner program is just the reverse of the printing program, again using three nested loops. Here is this part of the program all in one go, so that it is easy for you to get the indents correct. See Digging into the Code for an explanation of how the comma-separated lines are created:

```
for y in range(SIZEY):
    f.write("\n")
    for x in range(SIZEX):
        line = ""
        for z in range(SIZEZ):
            blockid = mc.getBlock(originx+x, originy+y,
                                  originz+z)
            if line != "":
                line = line + ","
            line = line + str(blockid)
        f.write(line + "\n")
    f.close()
```

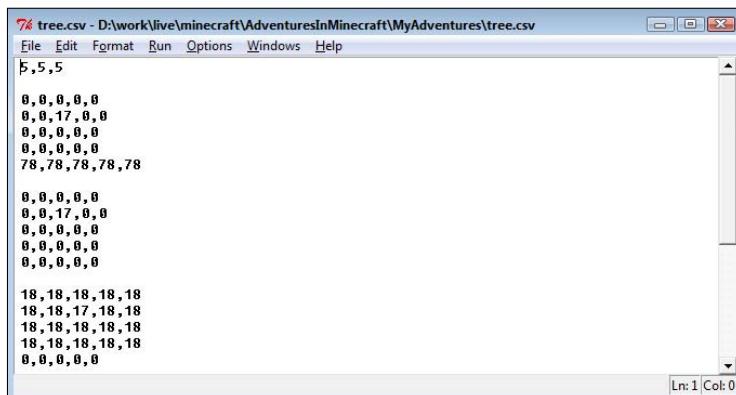
9. Finally, the main program is not indented at all. This just reads the player position, then calculates a cube space, where the player is standing at the centre of that space, and asks the `scan3D()` function to scan that whole space to your CSV file.

```
pos = mc.player.getTilePos()
scan3D(FILENAME, pos.x-(SIZEX/2), pos.y, pos.z-(SIZEZ/2))
```

Save your program. Double check that all your indentation is correct before you run it.

Now run up to a tree and hug it (stand as close as you can to its trunk), and run your `scan3D.py` program. What happens?

Did anything happen at all? Remember that this program scans an object to a CSV file, so you have to look at the CSV file `tree.csv` to see what numbers have been stored in it. Open the file `tree.csv` by choosing **File**→**Open** from the editor window. Figure 6-6 shows a section of the `tree.csv` file that I captured after hugging a tree on my computer! Because the scanning space is quite small, you might regularly only get half a tree scanned, but you can always change the `SIZE` constants to scan a bigger area.



The screenshot shows a Windows Notepad window with the title bar 'tree.csv - D:\work\live\minecraft\AdventuresInMinecraft\MyAdventures\tree.csv'. The menu bar includes File, Edit, Format, Run, Options, Windows, and Help. The main text area contains the following data:

```
5,5,5
0,0,0,0,0
0,0,17,0,0
0,0,0,0,0
0,0,0,0,0
78,78,78,78,78

0,0,0,0,0
0,0,17,0,0
0,0,0,0,0
0,0,0,0,0
0,0,0,0,0

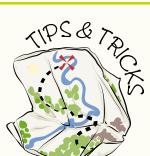
18,18,18,18,18
18,18,17,18,18
18,18,18,18,18
18,18,18,18,18
0,0,0,0,0
```

Ln: 1 Col: 0

FIGURE 6-6 Contents of CSV file after hugging and scanning a tree



Remember that the direction that your player is facing is not necessarily the direction that is scanned, so make sure you look around the area. The 3D scanner will scan from where your player is standing, through increasing values in coordinates. So, if you are standing at 0,0,0, the 3D scanner will scan from there through to coordinates that are larger such as 4,4,4. Look back at the diagram in Adventure 2 where you first learnt about coordinates to understand which direction each part of the coordinate represents.



IDLE normally only shows files that end in the `.py` extension in its Open and Save windows, but you can see your `.csv` files by choosing **Files of Type**→**All Files**.

CHALLENGE

Now that you have a 3D scanner and a 3D printer, modify your 3D printer program to use a `FILENAME` constant of `tree.csv`, then run around in the Minecraft world and keep running `print3D.py`. You should be able to print trees all over the place. Try printing some trees up in the sky and under water to see what happens. How quickly can you build a whole forest of trees this way?



DIGGING INTO THE CODE

In the `scan3D.py` program, you used a special character in the `write()` function that needs to be explained:

```
f.write("\n")
```

The `\n` character (called “backslash n”) is a special non-printable character that Python allows you to use inside quotes. All it means is “move to the next line.” The `n` character is used because it stands for “newline.” The backslash character is used to distinguish this letter `n` from a normal letter `n`.

In the `scan3D.py` program, you designed the file format so that there was a blank line between each layer of the object data to make it easy for people to read and edit the CSV data file. `f.write("\n")` just puts in this blank line for you.

Later in the `scan3D.py` program, there is some interesting code to do with building up the line variable. Here are the important parts of that code:

```
for x in range(SIZEX) :  
    line = ""  
    for z in range(SIZEZ) :  
        blockid = mc.getBlock(originx+x, originy+y,  
                             originz+z)  
        if line != "": # line is not empty  
            line = line + ","  
        line = line + str(blockid)
```

The parts that are underlined are part of a very common coding pattern that is regularly used for building comma-separated values. When the `x` loop starts, the line variable is set to an empty string. Every time the `z` loop generates another value, it first checks if the line is empty, and if it isn’t, it adds a comma. Finally it adds on the blockid of this block. The `if` statement is necessary to prevent a comma appearing at the start of the line before the first number, which would confuse the `print3D.py` program when it read it back in.

Building a Duplicating Machine

You now have all the building blocks of programs you need in order to design and build your own 3D duplicating machine that will be envy of all your friends. With it, you will be able to jump into the Minecraft world and make a magic duplicating room materialise, in which you can build any object you like. You can then save these objects to files, load them back into the room to edit them or magically duplicate them all over the Minecraft world. Finally, you can escape from the world and make the duplicating room vanish completely, leaving no traces of your magic behind you.

Just like in some of the earlier adventures, you are going to use your existing programs and stitch them together into a much bigger program. Because this program has a number of features, you are going to add a menu system to it so it is easy to control.

Writing the Framework of the Duplicating Machine Program

The first thing to do is to write the framework of the program that makes it all hang together. You will start with some dummy functions that just print their name when you call them, and gradually fill in their detail using your existing functions from other programs. You may remember that this is the same way you built up your treasure hunt game in Adventure 4?

1. Start a new file with File→New File and save it as **duplicator.py**.
2. Import the necessary modules:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import glob
import time
import random
```

3. Connect to the Minecraft game:

```
mc = minecraft.Minecraft.create()
```

4. Set some constants for the size of your duplicating machine. Don't set these too big, or the duplicator will take too long to scan and print objects. Also set an initial default position for your duplicating room:

```
SIZEX = 10
SIZEY = 10
SIZEZ = 10
roomx = 1
roomy = 1
roomz = 1
```

5. Define some dummy functions for all the features that this program needs to have. You will fill in the detail of these soon:

```
def buildRoom(x, y, z):
    print("buildRoom")

def demolishRoom():
    print("demolishRoom")

def cleanRoom():
    print("cleanRoom")

def listFiles():
    print("listFiles")

def scan3D(filename, originx, originy, originz):
    print("scan3D")

def print3D(filename, originx, originy, originz):
    print("print3D")
```

6. The dummy menu function is special, because it returns a value at the end of the function. For now, you can generate a random option and **return** it so that it is possible to test the early versions of your program, but soon you will write a proper menu here. You are writing a dummy menu so that you can test the structure of your program first, and you will soon fill this in with a proper menu:

```
def menu():
    print("menu")
    time.sleep(1)
    return random.randint(1, 7)
```

7. Write the main game loop that displays a menu then uses the function required for that feature. See Digging into the Code for more information about how the **anotherGo** variable is used:

```
anotherGo = True
while anotherGo:
    choice = menu()

    if choice == 1:
        pos = mc.player.getTilePos()
        buildRoom(pos.x, pos.y, pos.z)

    elif choice == 2:
        listFiles()
```

```

        elif choice == 3:
            filename = raw_input("filename?")
            scan3D(filename, roomx+1, roomy+1, roomz+1)

        elif choice == 4:
            filename = raw_input("filename?")
            print3D(filename, roomx+1, roomy+1, roomz+1)

        elif choice == 5:
            scan3D("scantemp", roomx+1, roomy+1, roomz+1)
            pos = mc.player.getTilePos()
            print3D("scantemp", pos.x+1, pos.y, pos.z+1)

        elif choice == 6:
            cleanRoom()

        elif choice == 7:
            demolishRoom()

        elif choice == 8:
            anotherGo = False

```

Save your test program and run it. What happens? Figure 6-7 shows what happened when I ran this program on my computer. You can see that the program is telling you what it is doing—it is choosing a random item from the menu, then it is using the function that will handle that menu option. As each function at the moment in your program only prints out its name, that is all that you see. You will see a different sequence of words on your screen to those in Figure 6-7 because the `menu()` function chooses a random choice each time it is used.

```

Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
menu
filename?tree.csv
scan3D
menu
filename?tree.csv
scan3D
menu
filename?tree.csv
scan3D
menu
scan3D
menu
print3D
menu
listFiles
menu
buildRoom
menu
scan3D
print3D
menu
filename?!

```

FIGURE 6-7 The results of running your test program



A **return** is a way of passing back a value from a function when it jumps back to the program that used the function in the first place.

For example, when you ask for a random number, you use the function `random.randint()`. This function returns a value, which you can store in a variable like this: `a = random.randint(1, 100)`. The `return` in Python just allows you to use this same technique of passing back a value from your own functions.



The `raw_input()` function reads a line of text entered at the keyboard. If you put a string between the brackets, that string is used as a prompt, so `name = raw_input("What is your name?")` will both ask a question and get the response.

When you use `raw_input()` to read from the keyboard, it always returns a string of text. If you want to enter a number (for example, in your menu system), you will have to use the `int()` function to convert it to a number. Some Python programmers like to do this all on one line, like this: `age = int(raw_input("What is your age?"))`.



Throughout this book you have been using Python version 2. If you use another computer it might have Python version 3 installed. There are a few differences between the two, and one of these is that in Python version 3, you need to use `input()` instead of `raw_input()` as used in Python 2. This book uses Python 2 because the Minecraft API is written to work with Python 2 only and would require some small modifications to make it work with Python 3.

DIGGING INTO THE CODE

You have used **boolean** variables before in other adventures, but it is worth looking at how the `anotherGo` variable is used in the `duplicator.py` program. Here are the important parts:

```
anotherGo = True
while anotherGo:
    choice = menu()
    if choice == 8:
        anotherGo = False
```

continued

continued

This is a common programming design pattern for a loop that runs at least once, asks you if you want another go and, if you don't, it quits the loop. There are many different ways to achieve this same goal in Python, but using a **boolean** variable is quite a good way to do this, as it is very clear how the program works.

You won't be using it in this book, but Python has a statement called **break** that can be used to break out of loops. You could do some research on the Internet to see how you could rewrite this loop to use a **break** statement instead, which would remove the need for the **boolean** variable.



A **boolean** variable is a variable that holds one of two values—either **True** or **False**. It is named after George Boole, who was a mathematician who did a lot of work in the 1800s with formal logic, much of which underpins the foundations of modern computing. You can read more about Boole here: http://en.wikipedia.org/wiki/George_Boole.

Displaying the Menu

A menu system is a very useful feature to add to your programs if they have lots of options for you to choose from. This menu system prints all the available options and then loops around, waiting for you to enter a number in the correct range. If you enter a number that is out of range, it just prints the menu again to give you another try.

1. Modify the menu function and replace it with the following (make sure that you get the indentation correct):

```
def menu():
    while True:
        print("DUPLICATOR MENU")
        print(" 1. BUILD the duplicator room")
        print(" 2. LIST files")
        print(" 3. SCAN from duplicator room to file")
        print(" 4. LOAD from file into duplicator room")
        print(" 5. PRINT from duplicator room to player.pos")
        print(" 6. CLEAN the duplicator room")
        print(" 7. DEMOLISH the duplicator room")
        print(" 8. QUIT")

        choice = int(raw_input("please choose: "))
        if choice < 1 or choice > 8:
```

```

    print("Sorry, please choose a number between 1 and 8")
else:
    return choice

```

Save the program and run it. What does your program do differently from the one with the dummy menu? Figure 6-8 shows what the real menu looks like.

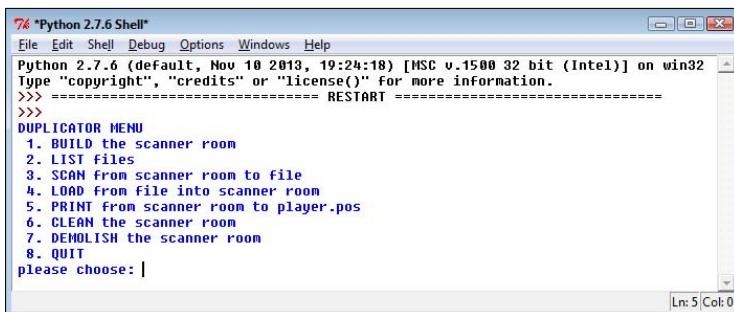


FIGURE 6-8 The menu system

By writing the framework of the program with the menu system and all the dummy functions, and then filling in the details of the functions one by one and re-testing, you are working in exactly same way as a modern software engineer when he or she develops computer programs. It is good practice to write your programs in small steps, making a change and then testing that change, until you eventually have a complete program. This also means that it will never take you more than two minutes to demonstrate your program to anyone who taps you on your shoulder and asks you to show them your new awesome program!



Building the Duplicator Room

The duplicator room is going to be built out of glass, and it will have a missing front on it so that your player can easily climb into it and create and delete blocks.

Replace the `buildRoom()` function with the following code. Be careful of the long lines that use `setBlocks()`, but note that I have not used the `\` arrow here, because Python allows you to split these across multiple lines (see Digging into the Code for an explanation of why you can sometimes split lines and sometimes cannot):

```

def buildRoom(x, y, z):
    global roomx, roomy, roomz

    roomx = x

```

```

roomy = y
roomz = z

mc.setBlocks(roomx, roomy, roomz,
    roomx+SIZEX+2, roomy+SIZEY+2, roomz+SIZEZ+2,
    block.GLASS.id)

mc.setBlocks(roomx+1, roomy+1, roomz,
    roomx+SIZEX+1, roomy+SIZEY+1, roomz+SIZEZ+1,
    block.AIR.id)

```

Save your program and test it again. Test option 1 on the menu to make sure that you can build the duplicator room. Run to another location in the Minecraft world and choose option 1 again to see what happens! Figure 6-9 shows the duplicator room after it has just been built.

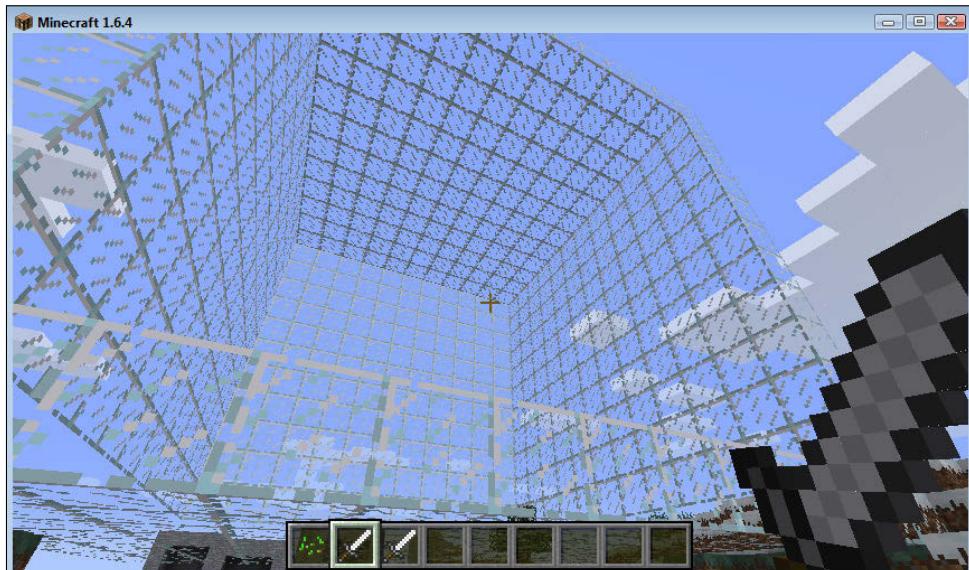


FIGURE 6-9 The duplicator room built in front of you

DIGGING INTO THE CODE

Python uses indentation (spaces or tabs) on the left-hand side, to identify which groups of program statements belong together. Whenever you use loops or `if/else` statements, or even functions, you have been using indentation to group together program statements that belong together. Python is unusual in its use of indents to group statements into blocks, many other languages such as C and C++ use special characters such as `{ and }` to group the

program statements together. You have to be extra-careful with Python indentation as a result; otherwise your program will not work properly!

Most of the time, Python does not allow you to split long lines, which is why in this book you will often see the ↳ arrow instructing you that this is a long line that should not be broken.

However, there are two other ways in Python that you can cope with long lines.

Firstly, you can use a line continuation character—if the last character at the end of a line is a backslash character (like this: \) then you can continue on the next line.

```
a = 1
if a == 1 or \
a == 2:
    print("yes")
```

The second method that you can sometimes use to split long lines is to split them at a point where it is obvious to Python that the line has not finished. For example, if you are setting initial values in a list, or using a function, you can break the lines when Python knows from the other parts of the line that there must be more code to follow. Here are two examples of where Python allows you to split a long line into shorter lines, because the open brackets tell Python that the line has not finished until a matching close bracket is seen:

```
names = ["David",
"Steve",
"Joan",
"Joanne"
]

mc.setBlocks(x1, y1, z1,
x2, y2, z2, block.AIR.id)
```

Demolishing the Duplicator Room

When you have run your duplicator program many times, you will probably end up with lots of old duplicator rooms built all over the Minecraft world. Only the latest one that you have created is actually a working room, and after some time your world will fill up so much that you won't know which room is the right one! To solve this problem, you will now add a feature to your program that demolishes the duplicator room so that your Minecraft world doesn't get cluttered with all these old rooms!

Demolishing the duplicator room is just like using your `clearSpace.py` program from Adventure 3. All you need to know is the outer coordinates of the room. Because the room is one block bigger all around the outside of the duplicating space defined by `SIZEX`, `SIZEY` and `SIZEZ`, this is quite simple to do with a little bit of maths.

Modify the `demolishRoom()` function to look like this.

```
def demolishRoom():
    mc.setBlocks(roomx, roomy, roomz,
                roomx+SIZEX+2, roomy+SIZEY+2, roomz+SIZEZ+2,
                block.AIR.id)
```

Save your program and run it again. Now it's easy to build or demolish your duplicator room. Just use option 1 on your menu to build it and option 7 to demolish it.

Scanning from the Duplicator Room

You have written this part of the program before, in your `scan3D.py` program, so you can use that with some small modifications.

1. Replace the `scan3D()` function with the following code. This code is almost identical to the `scan3D.py` program, but the lines in bold have been added to make it show the progress of the scanning on the Minecraft chat. Scanning a big room can take a long time, so it is nice to have some indication of how far through the process your program has got. You can use copy and paste to bring in the code from your earlier program to save some typing time here:

```
def scan3D(filename, originx, originy, originz):
    f = open(filename, "w")
    f.write(str(SIZEX) + "," + str(SIZEY) + "," + str(SIZEZ)
+ "\n")

    for y in range(SIZEY):
        mc.postToChat("scan:" + str(y))
        f.write("\n")
        for x in range(SIZEX):
            line = ""
            for z in range(SIZEZ):
                blockid = mc.getBlock(originx+x, originy+y,
                                      originz+z)
                if line != "":
                    line = line + ","
                line = line + str(blockid)
            f.write(line + "\n")
    f.close()
```

2. Save the program and test it again by jumping into the duplicating room and building something, then choosing option 3 from the menu. Open up the file that it creates, to check that the object has been scanned properly. Figure 6-10 shows a portion of a scanned file. Note that there are lots of zeros; this is because it has scanned all the `AIR` blocks as well.

```
76 block.csv - D:/work/live/minecraft/AdventuresInMinecraft/MyAdventures/block.csv
File Edit Format Run Options Windows Help
10,10,10

0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,1,0,0,1,0
0,0,0,0,1,1,1,1,1,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,1,0,0,0,0
Ln: 1 Col: 0
```

FIGURE 6-10 The contents of the scanned file

Cleaning the Duplicator Room

Fancy a spring clean? Sometimes it is useful to have a fresh start and just clear the room. You could do this by demolishing the room and rebuilding it, but it's really easy to add a clean feature, as it is only the coordinates that differ from the [demolishRoom\(\)](#) function.

1. Modify the [cleanRoom\(\)](#) function so that it looks like this. Note how the start coordinates are all bigger than the edge of the room, and the end coordinates are not as far over as in the [demolishRoom\(\)](#) function. This way, it clears the inner space of the room without destroying its walls:

```
def cleanRoom():
    mc.setBlocks(roomx+1, roomy+1, roomz+1,
                roomx+SIZEX+1, roomy+SIZEY+1, roomz+SIZEZ+1,
                block.AIR.id)
```

2. Save your program and run it again. Create something inside the room and then choose option 6 to test whether you can clear the room quickly.

Printing from the Duplicator Room

Printing (duplicating) the object that is in the room is really easy, because you have already written the [print3D.py](#) program that does this, and it is identical. Here it is again so you can see it all in one place:

```
def print3D(filename, originx, originy, originz):
    f = open(filename, "r")
    lines = f.readlines()

    coords = lines[0].split(",")
    sizex = int(coords[0])
```

```

sizey = int(coords[1])
sizez = int(coords[2])

lineidx = 1

for y in range(sizey):
    mc.postToChat("print:" + str(y))
    lineidx = lineidx + 1

    for x in range(sizex):
        line = lines[lineidx]
        lineidx = lineidx + 1
        data = line.split(",")

        for z in range(sizez):
            blockid = int(data[z])
            mc.setBlock(originx+x, originy+y, originz+z,
blockid)

```

Save the program and run it again. Test that you can build something in the room, then run to somewhere in the Minecraft world and choose option 5 from the menu. The contents of the room are scanned then printed just to the side of where your player is standing. You should be able to run to anywhere in the Minecraft world and duplicate your objects many times. Try duplicating objects in the sky and under water to see what happens! Figure 6-11 shows the results of running this program.

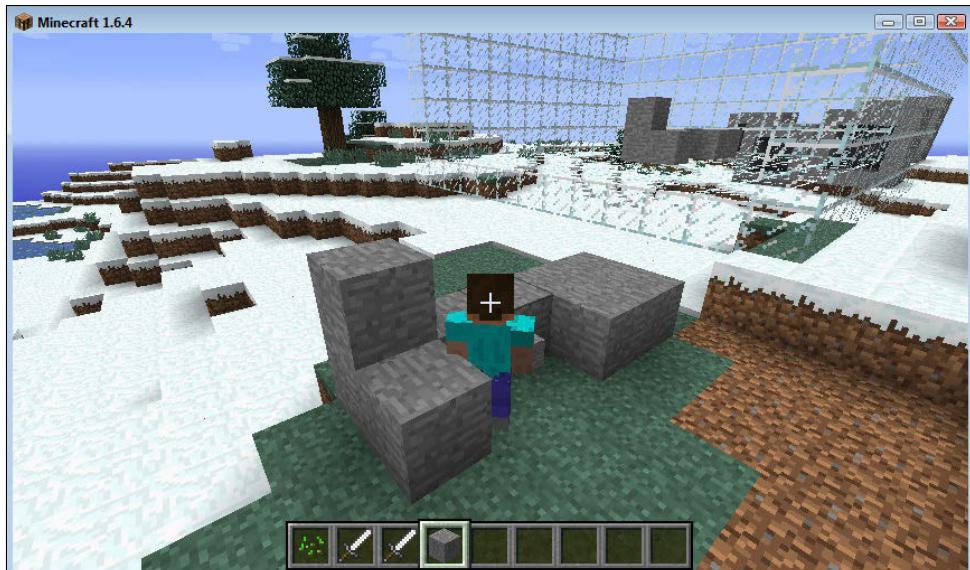


FIGURE 6-11 A stone shape in the room and printed next to your player

Listing Files

Your last task is to write a useful little function that lists all the files in your filing system that are CSV files. You could just use the File Manager, but it is nice to add this feature to your program so that you have everything you need in one place:

1. Modify the `listFiles()` function to use the `glob.glob()` function to read in a list of all files and print them out. See Digging into the Code for an explanation of how this works.

```
def listFiles():
    print("\nFILES:")
    files = glob.glob("*.csv")
    for filename in files:
        print(filename)
    print("\n")
```

2. Save the program and run it again. Scan a few objects into CSV files, then choose option 2 from the menu and make sure that they are all listed. Figure 6-12 shows the files I created when I ran this on my computer.

```
76 *Python 2.7.6 Shell*
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
DUPLICATOR MENU
1. BUILD the scanner room
2. LIST files
3. SCAN from scanner room to file
4. LOAD from file into scanner room
5. PRINT from scanner room to player.pos
6. CLEAN the scanner room
7. DEMOLISH the scanner room
8. QUIT
please choose: 2

FILES:
blocks.csv
maze1.csv
object1.csv
tree.csv

DUPLICATOR MENU
1. BUILD the scanner room
2. LIST files
3. SCAN from scanner room to file
4. LOAD from file into scanner room
5. PRINT from scanner room to player.pos
6. CLEAN the scanner room
7. DEMOLISH the scanner room
8. QUIT
please choose:

Ln: 23 Col: 0
```

FIGURE 6-12 Viewing the list of CSV files you have created

DIGGING INTO THE CODE

`glob.glob()`—what a funny name for a function. So funny that you have to put it twice!

The name `glob` is used twice because the first one is the name of the module (the `glob` module) that it was imported from at the top of your program. The second use of `glob` is the name of the function `glob` inside the module `glob`.

But what does `glob` do, and why is it called `glob`?

It is simply short for the word “global command” and stems from the early days of the design of the Unix operating system. You can read about the history behind the name `glob` on the Wikipedia page at [http://en.wikipedia.org/wiki/Glob_\(programming\)](http://en.wikipedia.org/wiki/Glob_(programming)).

All it does is collect a list of files that match a pattern (or a **wildcard**). When you use `glob.glob("*.csv")` Python searches the current directory in the computers filing system, and generates a list of all the files that end in `".csv"`.

So, if you have the files `one.csv`, `two.csv` and `three.csv` in your `MyAdventures` folder, using `glob.glob("*.csv")` will return a Python list that looks like this:

```
['one.csv', 'two.csv', 'three.csv']
```

Remember that the `for` loop will loop through all items in a list, which is why these next lines of code are really useful:

```
for name in glob.glob("*.csv")
    print(name)
```



A **wildcard** is a special character that can be used to select lots of similar names or words. It's like a joker or a “wild card” in a pack of playing cards, it can represent anything you want it to be.

In Python, the wildcard is often used to select lots of similarly named files in the filing system, such as “all CSV files”. This can be done with `glob.glob("*.csv")`, the `*` character marks the wildcard position, and `glob.glob()` will match any file in the filing system that ends with the letters `.csv`.

CHALLENGE

If in your `duplicator.py` program, you type in a filename that does not exist, the program crashes with an error and leaves your duplicating room in the Minecraft world. Make your program more robust by researching on the Internet how you can detect that a file does not exist so that your program does not crash.

When Martin tested the `duplicator.py` program for me, he found a bug—the bug is that the room is actually bigger than it should be. This means that if you build something the full size of the room, one slither of blocks down one edge will get chopped off when it is scanned to the file, and when you print it, that slither of blocks will be missing. This is quite inconvenient if it's the side of a house or something! See if you can locate the cause of this bug and fix it yourself!



Quick Reference Table

Reading lines from a file	Writing lines to a file
<pre>f = open("data.txt", "r") tips = f.readlines() f.close() for t in tips: print(t)</pre>	<pre>f = open("scores.txt", "w") f.write("Victoria:26000\n") f.write("Amanda:10000\n") f.write("Ria:32768\n") f.close()</pre>
Getting a list of matching filenames	Stripping unwanted white space from strings
<pre>import glob names = glob.glob("*.csv") for n in names: print(n)</pre>	<pre>a = "\n\n hello \n\n" a = a.strip() print(a)</pre>

Further Adventures in Data Files

In this adventure, you learnt how to read from and write to data files. This opens up endless opportunities for saving and restoring parts of the Minecraft world, and even bringing in large amounts of real-world data from other sources, such as websites. You built your own 3D mazes with lots of winding tunnels and dead ends, and finished by building a fully functional 3D scanner and printer, complete with a full menu system. This technique of writing a menu system will be useful for many other programs too!

- There are many sources of “live data” on the Internet. One data source that I found when researching for this book was the Nottingham City Council car park data here: <http://data.nottinghamtravelwise.org.uk/parking.xml>. This data is updated every five minutes and tracks cars as they enter and exit the car parks. I have written a Python program and put it on my github page that processes this data and prints out useful information here: <https://github.com/whaleygeek/pyparsers>. See if you can use this to write a Minecraft game that builds car parks with cars in them, inside your Minecraft world. Your game could be a car-parking challenge where you have to run around the Minecraft world and try to find a spare parking space in a limited time!
- Everything you ever wanted to know about 3D mazes is explained in detail on this fantastic web page here: <http://www.astrolog.org/labyrnth/algorithm.htm>. This page includes lots of example multi-layer 3D mazes that are stored in normal text files. Look through the site and see if you can find a file for a multi-layer maze, and modify your maze program to use the techniques you learned with your 3D duplicating machine to build a huge multi-layered maze. You might even experiment with some of the suggestions on this website to write a Python program that automatically generates mazes for you!



Achievement Unlocked: Shattering the laws of physics and magically materialising and dematerialising huge objects all over the Minecraft world at the push of a button.

In the Next Adventure...

In Adventure 7, you will learn how to build complex 2D and 3D objects by writing Python programs. You will learn how to keep time with the Minecraft clock, build polygons and other multi-sided shapes with just a few lines of Python code—and set off on an exciting expedition to the Pyramids!



Adventure 7

Building 2D and 3D Structures

ONE OF THE great things about programming in Minecraft is that as well as looking at your creations on a 2D screen you can actually bring them to life in a virtual 3D world. You can walk around them, go inside them, make them bigger—even blow them up if you like! By using the ideas originally created to display 3D objects on a 2D screen, you can code 3D objects in Minecraft, turning the ordinary into the extraordinary.

In this adventure you will learn how to use the `minecraftstuff` module to create 2D lines and shapes which when combined with a little maths will allow you to program a clock so big you can stand on its hands as they go round (see Figure 7-1). Once you have mastered creating 2D shapes, you will then learn how to combine them together to create enormous 3D structures in seconds.



FIGURE 7-1 A massive Minecraft clock

The `minecraftstuff` Module

`minecraftstuff` is an extension module to the Minecraft API, which was written for Adventures in Minecraft, and contains all the code you need to draw shapes and control 3D objects. It has a set of functions called `MinecraftDrawing`, which allows you to create lines, circles, spheres and other shapes. Because this complicated code is in a separate **module** it makes it simpler for you to create the code, and it's easier to understand too. Modules are a way of splitting up a program into smaller chunks. When programs get too big they are more difficult to read and harder to understand, and it takes longer to track down problems.

The `minecraftstuff` module is included in the “Adventures in Minecraft” starter kit available on the book’s companion website at www.wiley.com/go/adventuresinminecraft and can be imported to a Minecraft program in the same way as the `minecraft` and `block` modules.

Creating Lines, Circles and Spheres

When you combine a lot of small, simple things, you can create something as large and complex as you want. In this adventure, you’re going to use lots of lines, circles, spheres and other shapes to produce a really big Minecraft structure. In this part of the adventure, you will create a new program, import the modules you need and set up the `minecraft` and `minecraftdrawing` modules. Later, you’ll use the functions in the modules to draw lines, circles and spheres.

Start up Minecraft, IDLE, and if you are working on a PC or a Mac, start up the Bukkit server too. You should have had a bit of practice with starting everything up by now, but refer back to Adventure 1 if you need any reminders.

1. Start by opening IDLE and creating a new file. Save the file as `LinesCirclesAndSpheres.py` in the `MyAdventures` folder.
2. Next, import the `minecraft`, `block`, `minecraftstuff` and `time` modules by typing the following lines:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import mcpi.minecraftstuff as minecraftstuff
import time
```

3. Create the connection to Minecraft:

```
mc = minecraft.Minecraft.create()
```

4. Use the `minecraftstuff` module to create a `MinecraftDrawing` object by typing:

```
mcdrawing = minecraftstuff.MinecraftDrawing(mc)
```

DIGGING INTO THE CODE

`MinecraftDrawing` is a class in the `minecraftstuff` module. Classes are part of a special method of programming called “Object Oriented” which is a way of grouping similar functions and data together—in this case, Minecraft drawing functions and data—so they are easier to understand and use. When you use a class and give it a name (e.g. `mcdrawing`) you create an object. This is known as instantiation. An object is similar to a variable but rather than just holding values (or data), it also has functions!

Object Oriented Programming (OOP) is a complex subject and volumes of books have been written on what it is and how to use it successfully, there is, however, a useful introduction to using classes in Python at en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial/Classes.

You can look at the code in `minecraftstuff` by opening the `minecraftstuff.py` file in the `MyAdventures/mcpi` folder using IDLE.

Drawing Lines

The `MinecraftDrawing` object has a function called `drawLine()`, which when called and passed two positions (x , y , z) and a block type as parameters it will create a line of blocks between those position. Just like the `setBlocks()` function you first learnt about in Adventure 3.



When a function needs information in order to run, such as `setBlock()` which needs an x , y , z and a block type, these values are known as **parameters**. When a program uses that function it is said to **call** it and **pass** parameters.

The following function creates the line of blocks like that shown in Figure 7-2:

```
drawLine(x1, y1, z1, x2, y2, z2, blockType, blockData)
```

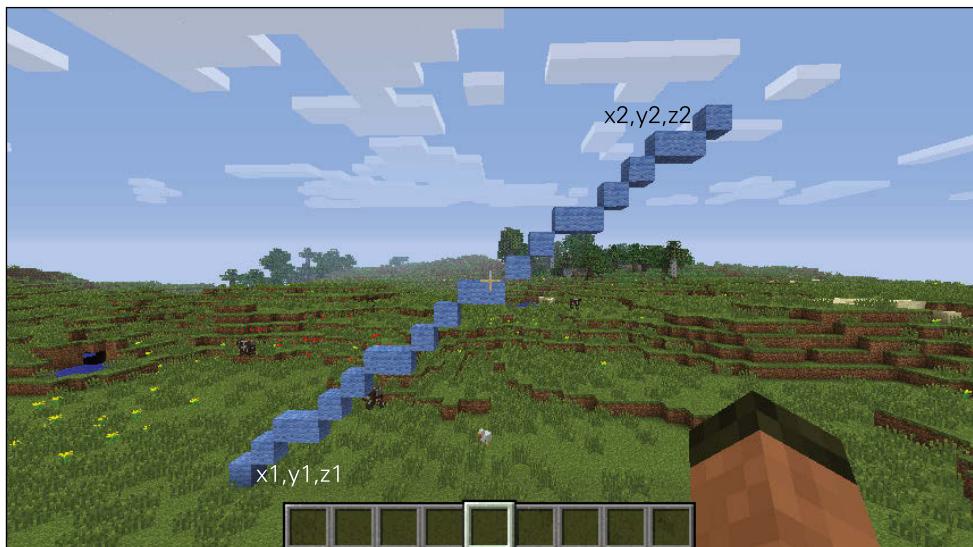


FIGURE 7-2 The `drawLine()` function creates a line of blocks between two sets of x,y,z points.

Now update your program so that it uses the `drawLine()` function to create three lines in Minecraft — one straight up, one across and one diagonal from the players position. Add the following code to the bottom of the `LinesCirclesAndSpheres.py` program:

1. First, you need to find the position of the player by typing the following code:

```
pos = mc.player.getTilePos()
```

2. To draw a vertical line of 20 blocks from the player's position straight upwards, type:

```
mcdrawing.drawLine(pos.x, pos.y, pos.z,  
                   pos.x, pos.y + 20, pos.z,  
                   block.WOOL.id, 1)
```

3. Now draw a horizontal line of 20 blocks straight across by typing:

```
mcdrawing.drawLine(pos.x, pos.y, pos.z,  
                   pos.x + 20, pos.y, pos.z,  
                   block.WOOL.id, 2)
```

4. Next draw a diagonal line of 20 blocks across and up:

```
mcdrawing.drawLine(pos.x, pos.y, pos.z,  
                   pos.x + 20, pos.y + 20, pos.z,  
                   block.WOOL.id, 3)
```

5. Finally, as we will be adding to this program, add a time delay so you can see what is happening:

```
time.sleep(5)
```

6. Now save the file and run the program. You should have created three lines of blocks, each in a different colour of wool: one running vertically from the player's position, one running horizontally and the third running diagonally between them.

DIGGING INTO THE CODE

The `drawLine()` function uses the Bresenham line algorithm to create the line. Have a look at http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm to learn more about the algorithm.

Drawing Circles

You don't have to stick to lines—you can also use `MinecraftDrawing` to create circles, by using the `drawCircle()` function, passing a centre point for the circle, a radius and a block type. You can create a circle by using the function:

```
drawCircle(x, y, z, radius, blockType, blockData)
```

To create the circle shown in Figure 7-3, add the following code to the bottom of the [LinesCirclesAndSpheres.py](#) program:

1. First, find the current position of the player by typing the following code:

```
pos = mc.player.getTilePos()
```

2. Now draw a circle, starting 20 blocks above the player and with a radius of 20 blocks by typing:

```
mcdrawing.drawCircle(pos.x, pos.y + 20, pos.z, 20,  
block.WOOL.id, 4)
```

3. Finally, add a time delay so you can see what's happening in the program and you have the chance to move the player:

```
time.sleep(5)
```

4. Save the file and run the program.



FIGURE 7-3 `drawCircle` creates a circle using a radius around a centre position (x,y,z)

The lines will be drawn again first, you then have five seconds to move the player before the circle is drawn directly above them.

DIGGING INTO THE CODE

The `drawCircle()` function uses the mid-point circle algorithm to create the circle with blocks. This is an adaptation of the Bresenham line algorithm. You can find out more about this at http://en.wikipedia.org/wiki/Midpoint_circle_algorithm.

Drawing Spheres

The `drawSphere()` function is similar to `drawCircle()` in that you work with a centre point, a radius and a block type. You can create a sphere by using the function:

```
drawSphere(x, y, z, radius, blockType, blockData)
```

To create the sphere shown in Figure 7-4, add the following code to the bottom of the `LinesCirclesAndSpheres.py` program:

1. Find the current position of the player by typing:

```
pos = mc.player.getTilePos()
```

2. To draw a sphere starting 20 blocks above the player, with a radius of 15 blocks, type:

```
mcdrawing.drawSphere(pos.x, pos.y + 20, pos.z, 15,  
block.WOOL.id, 5)
```

3. Save the file and run the program.



FIGURE 7-4 `drawSphere` creates a sphere using a radius around a centre position (x,y,z)

The lines and circles will be drawn again, giving you five seconds in between to move the player before the sphere is then drawn.

You can download the complete code for lines, circles and spheres from the Adventures in Minecraft companion website at www.wiley.com/go/adventuresinminecraft.



Spheres are great for creating explosions in Minecraft. By drawing a sphere of **AIR**, you can remove all the blocks around the centre of the sphere, creating a 'hole' in the world.

CHALLENGE



Now that you've seen how simple it is to create basic shapes, try creating your own 3D art masterpiece in code, using lines, circles and spheres.

Creating a Minecraft Clock

Now that you know how to create circles and lines, you might be able to see how you would code the clock in Figure 7-1. The face is simply a large circle, and each of the hands is a line. Now comes the difficult part—working out where to put the lines and how to make them move.

VIDEO



To see a tutorial on how to create the Minecraft Clock, visit the companion website at www.wiley.com/go/adventuresinminecraft.

In this part of the adventure, you will use **trigonometry** to work out where to point the hands by turning the angle of the hand into the x and y coordinates on the clock face (see Figure 7-5). You're going to make the hands seem to move by first drawing them with blocks, then removing them by drawing them with **AIR**, then drawing them again in a new position.

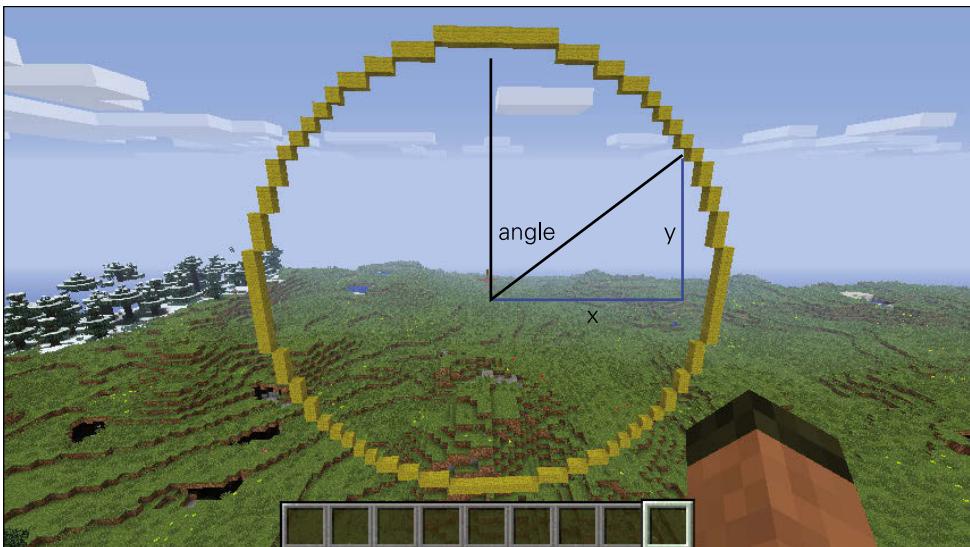


FIGURE 7-5 Finding where to point the hands on a clock face

Trigonometry is the branch of mathematics that deals with the relationships between angles and the lengths of sides in triangles. Visit en.wikipedia.org/wiki/Trigonometry to learn more.



To create your clock, follow these steps:

1. Create a new program for the Minecraft clock by opening IDLE and creating a new file. Save the file as `MinecraftClock.py` in the `MyAdventures` folder.
2. Import the `minecraft`, `block`, `minecraftstuff`, `time`, `datetime` and `math` modules by typing the following code:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import mcpi.minecraftstuff as minecraftstuff
import time
import datetime
import math
```

3. Create a function, `findPointOnCircle()`. When you pass this function the centre of the circle and the angle of the clock's hands, the function returns the position of the clock's hands as shown in Figure 7-5.

```
def findPointOnCircle(cx, cy, radius, angle):
    x = cx + math.sin(math.radians(angle)) * radius
    y = cy + math.cos(math.radians(angle)) * radius
    x = int(round(x, 0))
    y = int(round(y, 0))
    return(x,y)
```

4. Connect to `Minecraft` and create the `MinecraftDrawing` object:

```
mc = minecraft.Minecraft.create()
mcdrawing = minecraftstuff.MinecraftDrawing(mc)
```

5. Find the player's current position by typing:

```
pos = mc.player.getTilePos()
```

6. Now you're going to create variables for the centre of the clock (which will be 25 blocks above the player's position), the radius of the clock face and the length of the clock hands:

```
clockMiddle = pos
clockMiddle.y = clockMiddle.y + 25

CLOCK_RADIUS = 20
HOUR_HAND_LENGTH = 10
MIN_HAND_LENGTH = 18
SEC_HAND_LENGTH = 20
```

7. Next, draw the clock face using `drawCircle()` by typing:

```
mcdrawing.drawCircle(clockMiddle.x, clockMiddle.y,
                      clockMiddle.z,
                      CLOCK_RADIUS, block.DIAMOND_BLOCK.id)
```

8. Start an endless loop. All the code after this point will be inside this loop.

```
while True:
```

9. Next you need to ask your computer what the time is by using the function `datetime.datetime.now()`. You will then split the time into hours, minutes and seconds. Because your clock is a 12-hour clock, not a 24-hour one, you need to specify that if the time is after noon, 12 should be subtracted from the hour (so that, for example, if the time is 14:00, it is shown as 2 o'clock). Do this by typing the following code:

```
timeNow = datetime.datetime.now()
hours = timeNow.hour
if hours >= 12:
    hours = timeNow.hour - 12
minutes = timeNow.minute
seconds = timeNow.second
```

10. Now draw the hour hand. The angle of this will be 360 degrees divided by 12 hours, multiplied by the current hour. Find the x and y position for the end of the hand using `findPointOnCircle()` and draw the hand using `drawLine()` by typing the following:

```
hourHandAngle = (360 / 12) * hours
hourHandX, hourHandY = findPointOnCircle(
    clockMiddle.x, clockMiddle.y,
    HOUR_HAND_LENGTH, hourHandAngle)
mcdrawing.drawLine(
    clockMiddle.x, clockMiddle.y, clockMiddle.z,
    hourHandX, hourHandY, clockMiddle.z,
    block.DIRT.id)
```

11. Next, draw the minute hand, which is one block behind ($z-1$) the hour hand, by typing:

```
minHandAngle = (360 / 60) * minutes
minHandX, minHandY = findPointOnCircle(
    clockMiddle.x, clockMiddle.y,
    MIN_HAND_LENGTH, minHandAngle)
mcdrawing.drawLine(
    clockMiddle.x, clockMiddle.y, clockMiddle.z-1,
    minHandX, minHandY, clockMiddle.z-1,
    block.WOOD_PLANKS.id)
```

12. Now add the second hand, which is one block in front ($z+1$) of the hour hand:

```
secHandAngle = (360 / 60) * seconds
secHandX, secHandY = findPointOnCircle(
    clockMiddle.x, clockMiddle.y,
    SEC_HAND_LENGTH, secHandAngle)
mcdrawing.drawLine(
    clockMiddle.x, clockMiddle.y, clockMiddle.z+1,
    secHandX, secHandY, clockMiddle.z+1,
    block.STONE.id)
```

13. Wait for one second:

```
time.sleep(1)
```

14. Now you need to clear the time by drawing the hands again, only this time you draw them using AIR. Type:

```
mcdrawing.drawLine(  
    clockMiddle.x, clockMiddle.y, clockMiddle.z,  
    hourHandX, hourHandY, clockMiddle.z,  
    block.AIR.id)  
mcdrawing.drawLine(  
    clockMiddle.x, clockMiddle.y, clockMiddle.z-1,  
    minHandX, minHandY, clockMiddle.z-1,  
    block.AIR.id)  
mcdrawing.drawLine(  
    clockMiddle.x, clockMiddle.y, clockMiddle.z+1,  
    secHandX, secHandY, clockMiddle.z+1,  
    block.AIR.id)
```

15. Save the file and run the program to see the result of your efforts. You should see the Minecraft Clock appear directly above the player, who can look up and walk to the side to see the time. Make sure the player is on the correct side of the clock though; otherwise time will be going backward!

You can download the complete code for the Minecraft Clock from the companion website www.wiley.com/go/adventuresinminecraft but I strongly recommend that you type in the code yourself as you read through the steps. You'll learn a lot more that way!

DIGGING INTO THE CODE

The `findPointOnCircle()` function works out a point (x, y) on a circle from the centre position of the circle (cx, cy), the radius of the circle and the angle you've specified (see Figure 7-5).

1. The function is defined with a `cx`, `cy`, `radius` and `angle` as input parameters:

```
def findPointOnCircle(cx, cy, radius,  
                      angle):
```

2. The point on the circle (x, y) is calculated using the `math` functions `sin()` and `cos()`, multiplied by the radius:

```
x = cx + math.sin(math.radians(angle))  
    * radius  
y = cy + math.cos(math.radians(angle))  
    * radius
```

The `math.sin()` and `math.cos()` functions need **radians** to be passed to them. Radians are a different way of measuring angles (rather than the often used 0 – 360 degrees), so the `math.radians()` function is used to convert the angles into radians.

3. The x, y values calculated are decimals, but the function needs whole numbers, so the `round()` and `int()` functions are used to round the decimal number to its nearest whole number and convert it from decimal to integer:

```
x = int(round(x, 0))
y = int(round(y, 0))
```

4. The x and y values are then returned to the program:

```
return(x, y)
```

The function returns both the x and y values at the same time, so the calling program must provide two variables when calling the function.

You create the hands of the clock using a three-step process:

1. Work out the angle of the hand by dividing 360° by 12 or 60 (depending whether it is an hour, minute or second hand) and then multiplying that by the number of hours, minutes or seconds:

```
hourHandAngle = (360 / 12) * hours
```

2. Find the coordinates (x, y) of the end of the hand using the `findPointOnCircle()` function:

```
hourHandX, hourHandY = findPointOnCircle(
    clockMiddle.x, clockMiddle.y,
    HOUR_HAND_LENGTH, hourHandAngle)
```

As the function returns two values, when it's called two variables (`hourHandX`, `hourHandY`) are provided.

3. Draw a line from the middle of the clock to the end of the hand:

```
mcdrawing.drawLine(
    clockMiddle.x, clockMiddle.y, clockMiddle.z,
    hourHandX, hourHandY, clockMiddle.z,
    block.DIRT.id)
```

CHALLENGE



The hour hand on real clocks tracks every minute of the hour. For example, if the time is 11:30, the hour hand will be halfway between the 11 and the 12. The code for the Minecraft clock you've just created doesn't work this way—the hour hand remains pointing at 11 until 11:59:59 ticks over to 12:00:00.

See if you can make your Minecraft clock work like a real clock by changing how you calculate the hour angle.

Drawing Polygons

A polygon is any 2D shape made up of straight connecting sides. It can have any number of sides, from three (a triangle) upward. As you can see from Figure 7-6, you can create any number of interesting polygons in Minecraft.



FIGURE 7-6 Examples of polygons in Minecraft

Although they are 2D shapes, you'll find polygons extremely useful for making 3D graphics, as you can create virtually any 3D object by connecting lots of polygons together. When polygons are used together to create 3D objects, they are known as **faces**. You can create some awesome structures this way. Just look at Figure 7-7, which shows the skyline of Manhattan Island, created by lots of polygons (see how it's done at www.stuffaboutcode.com/2013/04/minecraft-pi-edition-manhattan-stroll.html).



FIGURE 7-7 Minecraft skyline of Manhattan Island, New York

A **face** is a single flat surface that is part of a larger object; for example, one side of a cube or the top of a drum.



You can create polygons (or faces) using the `drawFace()` function in `MinecraftDrawing`. The function expects a list of points (x, y, z) that, when connected together in sequence, will create a complete polygon. Passing a `True` or `False` will determine whether the face is filled, and the final parameter is a what block the face should be made from (see Figure 7-8):

```
drawFace(points, filled, blockType, blockData)
```

Create a new program to experiment with the `drawFace()` function and create the triangle shown in Figure 7-8:

1. First, open IDLE and create a new file. Save the file as `Polygon.py` in the `MyAdventures` folder.
2. Import the `minecraft`, `block` and `minecraftstuff` modules by typing:

```
import mcpi.minecraft as minecraft  
import mcpi.block as block  
import mcpi.minecraftstuff as minecraftstuff
```



FIGURE 7-8 `drawFace()` used to create a triangle from three points

3. Connect to minecraft and create the `MinecraftDrawing` object:

```
mc = minecraft.Minecraft.create()  
mcdrawing = minecraftstuff.MinecraftDrawing(mc)
```

4. Get the player's current position:

```
pos = mc.player.getTilePos()
```

5. Now you need to create a list to hold the points of the polygon. Start by typing:

```
points = []
```

6. Then append three points (x, y, z) to the points list, which, when joined together, will create a triangle:

```
points.append(minecraft.Vec3(pos.x, pos.y, pos.z))  
points.append(minecraft.Vec3(pos.x + 20, pos.y, pos.z))  
points.append(minecraft.Vec3(pos.x + 10, pos.y + 20,  
                           pos.z))
```

7. Use the `MinecraftDrawing.drawFace` function to create the triangle polygon:

```
mcdrawing.drawFace(points, True, block.WOOL.id, 6)
```

8. Save the file and run the program to create the triangle polygon.

DIGGING INTO THE CODE

The points of the polygon's face are created using `minecraft Vec3(x, y, z)`. The `minecraft(Vec3())` is the Minecraft API's way of holding a set of coordinates (x, y, z) together. Vec3 is short for 3D vector.

The points of the face are added to the points list using `append()`; this adds a new item to the end of a list.

CHALLENGE

What other shapes can you make using `drawFace()`? How about a five-sided shape such as a pentagon?



Pyramids

Find a picture of a pyramid and take a good look at it. What do you notice? What shape are its sides? What do all the sides have in common? How many sides does it have?

As you probably know, each side of a pyramid (except the base) is always a triangle. The pyramids in Egypt had four sides (or five if you include the base), but they can have any number of sides from three upward. Did you also notice that the base of any pyramid will fit exactly into a circle! Take a look at Figure 7-9 to see what I mean.

You are now going to create a program, which by using the `drawFace()` and `findPointOnCircle()` functions, will make any size of pyramid, of any height, with any number of sides:

1. First, open IDLE and create a new file. Save the file as `MinecraftPyramids.py` in the `MyAdventures` folder.
2. Import the `minecraft`, `block`, `minecraftstuff` and `math` modules by typing:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import mcpi.minecraftstuff as minecraftstuff
import math
```



FIGURE 7-9 A pyramid made of triangles, which fits exactly into a circle

3. Create the `findPointOnCircle()` function, which will be used to work out where each of the triangles which make up the pyramid will be placed:

```
def findPointOnCircle(cx, cy, radius, angle):
    x = cx + math.sin(math.radians(angle)) * radius
    y = cy + math.cos(math.radians(angle)) * radius
    x = int(round(x, 0))
    y = int(round(y, 0))
    return (x,y)
```

4. Connect to Minecraft and create the `MinecraftDrawing` object:

```
mc = minecraft.Minecraft.create()
mcdrawing = minecraftstuff.MinecraftDrawing(mc)
```

5. Now set up the variables for your pyramid. The middle of the pyramid will be the player's position. The values of these variables will change the size (or radius), height and number of sides of the pyramid. Type:

```
pyramidMiddle = mc.player.getTilePos()

PYRAMID_RADIUS = 20
PYRAMID_HEIGHT = 10
PYRAMID_SIDES = 4
```

6. Loop through each side of the pyramid; all the code after this point will be indented under the `for` loop:

```
for side in range(0, PYRAMID_SIDES):
```

The bigger the pyramid, the longer the program will take to run and the longer Minecraft will take to show the pyramid in the game. If your pyramid is too tall it may also go over the maximum height of the game. So take it slowly and expand your values gradually. You can make enormous pyramids, but you may need to be patient if they are really big they take a while to appear!



- For each side of the pyramid, calculate the angles of the sides of the triangle, then use `findPointOnCircle()` to find the x, y, z coordinates. The angle is calculated by dividing 360 degrees by the total number of sides, multiplied by the number of the side which is being drawn, as you can see in Figure 7-10. Type the code as follows:

```
point1Angle = int(round((360 / PYRAMID_SIDES) * side,0))
point1X, point1Z = findPointOnCircle(
    pyramidMiddle.x, pyramidMiddle.z,
    PYRAMID_RADIUS, point1Angle)
point2Angle = int(round((360 / PYRAMID_SIDES)
                        * (side + 1),0))
point2X, point2Z = findPointOnCircle(
    pyramidMiddle.x, pyramidMiddle.z,
    PYRAMID_RADIUS, point2Angle)
```

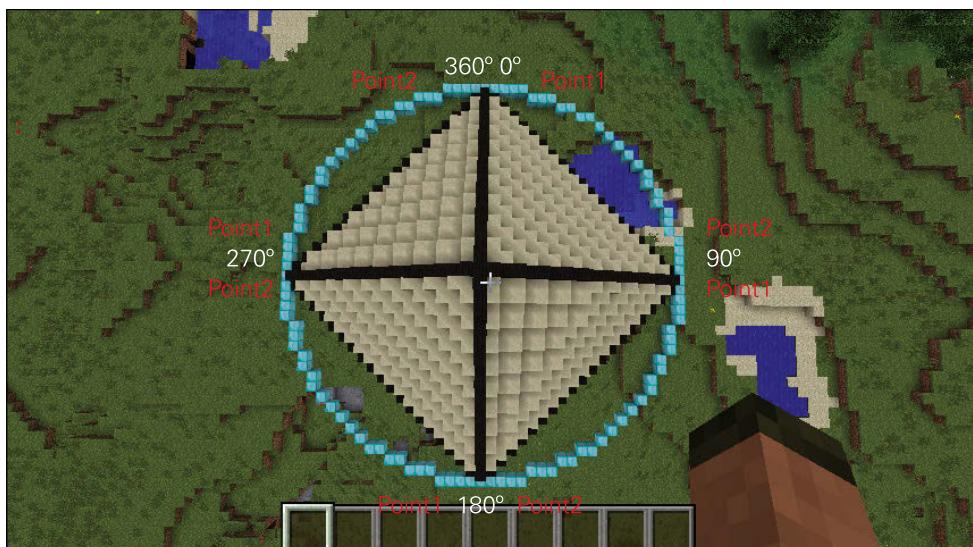


FIGURE 7-10 The angles of a four-sided pyramid

8. Create the points of the triangle and use `drawFace()` to create the side of the pyramid as follows:

```
trianglePoints = []
trianglePoints.append(
    minecraft.Vec3(point1X, pyramidMiddle.y, point1Z))
trianglePoints.append(
    minecraft.Vec3(point2X, pyramidMiddle.y, point2Z))
trianglePoints.append(
    minecraft.Vec3(pyramidMiddle.x,
                  pyramidMiddle.y + PYRAMID_HEIGHT,
                  pyramidMiddle.z))
mcdrawing.drawFace(trianglePoints, True,
                    block.SANDSTONE.id)
```



Sandstone (`block.SANDSTONE.id`) is a really useful block type to use for pyramids as it looks very similar to sand but has a very useful characteristic: it isn't affected by gravity and doesn't fall down if there are no blocks underneath it to hold it up. If you were to make the pyramid out of sand, the player would be buried in it and would have to spend ages digging himself out.

9. Save the file and run the program. You'll see a pyramid appear above the player—and trap him inside!

This program can create pyramids of any size and with any number of sides. Try changing the pyramid variables in the program and re-running it. Figure 7-11 shows a couple of impressive examples.

You can download the complete code for the Minecraft pyramid from the Adventures in Minecraft companion website at www.wiley.com/go/adventuresinminecraft.

CHALLENGE



The pyramid you've created doesn't have a base. Can you create a polygon that fits on the bottom of the pyramid? This should be easy for a four-sided pyramid but if you code it correctly, the same code should also work for a five-, six- or seven-sided pyramid.

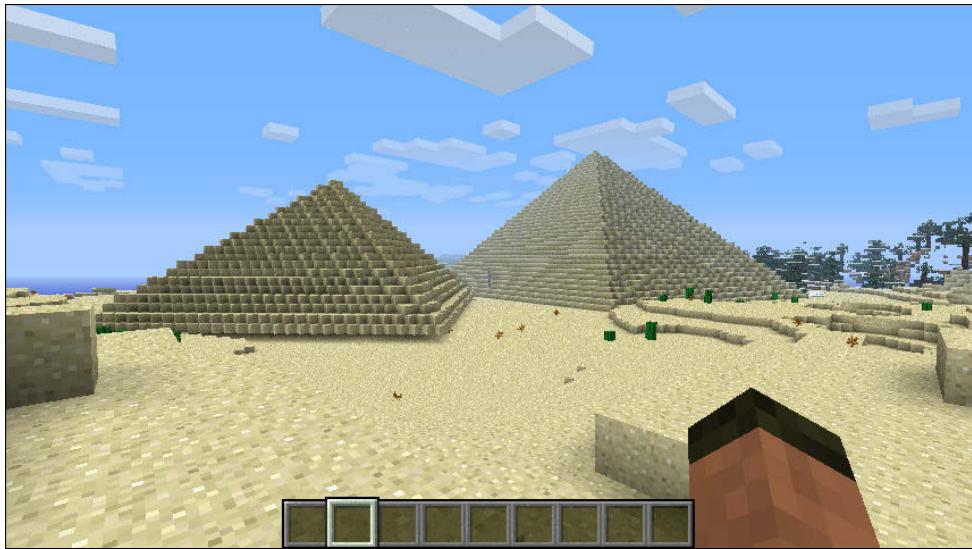


FIGURE 7-11 Minecraft pyramids

Further Adventures with 2d and 3d Shapes

Using the `drawFace()` function, you can create any sort of **polyhedron**, which is a shape with flat faces (just like the pyramids you created earlier), so why not create some more?

You will find lots of polyhedron examples and good ideas at:

- Maths is Fun (www.mathsisfun.com/geometry/polyhedron.html)
- Kids Math Games (www.kidsmathgamesonline.com/facts/geometry/3dpolyhedronshapes.html)

Quick Reference Table

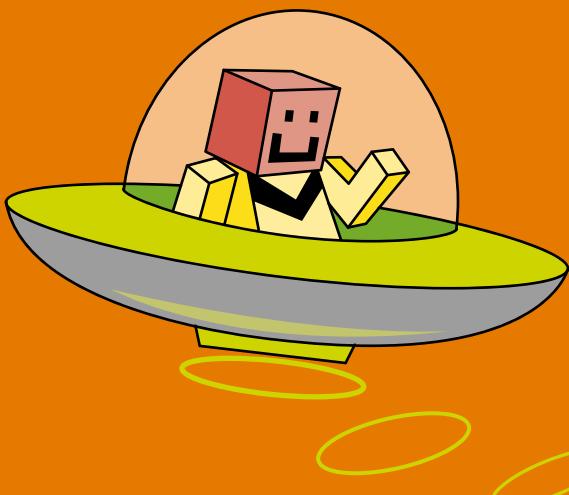
Commands	Description
<pre>mcdrawing.drawLine(0, 0, 0, 10, 10, 10, block.DIRT.id)</pre>	Draw a line between two points
<pre>mcdrawing.drawCircle(0, 0, 0, radius, block.DIRT.id)</pre>	Draw a circle
<pre>mcdrawing.drawSphere(0, 0, 0, radius, block.DIRT.id)</pre>	Draw a sphere
<pre>tri = [] filled = True tri.append(minecraft.Vec3(0,0,0)) tri.append(minecraft.Vec3(10,0,0)) tri.append(minecraft.Vec3(5,10,0)) mcdrawing.drawFace(tri, filled, block.DIRT.id)</pre>	Draw a polygon or face (e.g. a triangle)



Achievement Unlocked: 3D master, creator of massive structures, all hail the pyramid builder!

In the Next Adventure...

In the next adventure, you will learn how to give Minecraft objects a mind of their own, make friends with a block and avoid an alien invasion.



Adventure 8

Giving Blocks a Mind of Their Own

COMPUTERS DON'T THINK. They aren't capable of having any thoughts at all; they can only ever do what programmers tell them to do. You can, however, make computers *look* like they are thinking and making decisions by themselves. By programming your computer to "understand" what's happening and giving it rules to "decide" what to do next, you can open a door to a lot of fun.

In this adventure, you are going to program a block friend who will follow you around, provided you don't get too far away. You are also going to create a flying saucer that chases you until it can get above you and beam you aboard.

You will learn how to make a block move and follow the path it decides is best, as well as how to use the Python `random` module to make it look as if the computer is thinking. You will also create shapes using the `MinecraftShape` functions in the `minecraftstuff` module (which is included in your starter kit).

Your Block Friend

Minecraft can be a lonely world. But your player doesn't have to be alone—you can create a block that will follow him around, talk to him and be his friend (see Figure 8-1).



FIGURE 8-1 Create a Hello block friend to accompany your player on his Minecraft adventures.

To program a block friend you need to think like her! She will be happy if she's near your player, so she will want to follow him around and try to sit next to him. She'll stay happy as long as your player is close to her; if your player walks off, she will walk after him. If your player gets too far away, it will make the block friend sad and she will stop following him until he comes back and gives her a hug (by standing next to her).

The block friend program has two distinct parts:

- **The rules the block uses to decide what it should do next:** This part of the program makes the block friend decide whether to move toward your player (the target) or stay still.
- **The code that moves the block toward a target:** Once the block friend reaches her target, she uses the rules again to work out what to do next.

While the block friend is moving toward the target, she should be travelling “on top” of the land, not floating in the air or burrowing under the ground! You do this by using the function `mc.getHeight(x, z)` and passing it a horizontal position (x,z). It will return the vertical position (y) of the first block down from the sky that isn’t `AIR`.

VIDEO



To see a tutorial on how to create the block friend, visit the companion website at www.wiley.com/go/adventuresinminecraft.

Start by creating a new program for the block friend:

1. Open IDLE, create click File ➔ New File to create a new program and save the file as `BlockFriend.py` in the `MyAdventures` folder.
2. Import the `minecraft`, `block`, `minecraftstuff`, `math` and `time` modules:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import mcpi.minecraftstuff as minecraftstuff
import math
import time
```

3. The first thing you need to do is create a function to calculate the distance between two points. This will be used to work out how far the block friend is from your player:

```
def distanceBetweenPoints(point1, point2):
    xd = point2.x - point1.x
    yd = point2.y - point1.y
    zd = point2.z - point1.z
    return math.sqrt((xd*xd) + (yd*yd) + (zd*zd))
```

4. Now you need to decide how far away your player needs to be for the block friend to stop following him. Create a constant to store the distance that you consider to be “too far away”. I have chosen 15, meaning that when the block friend and the player are 15 blocks apart, the block friend will stop following the player:

```
TOO_FAR_AWAY = 15
```

5. Create the `Minecraft` and `MinecraftDrawing` objects:

```
mc = minecraft.Minecraft.create()
mcstuff = minecraftstuff.MinecraftDrawing(mc)
```

6. Create a variable to store the block’s mood. For this adventure, the block will either be happy or sad. Set it to "`happy`":

```
blockMood = "happy"
```

7. Create the block friend, a few blocks away from the player, by getting the player’s position, adding 5 to the x position and using the `getHeight()` function to find out the y position, so the block is sitting on top of the land:

```
friend = mc.player.getTilePos()
friend.x = friend.x + 5
friend.y = mc.getHeight(friend.x, friend.z)
mc.setBlock(friend.x, friend.y, friend.z,
            block.DIAMOND_BLOCK.id)
mc.postToChat("<block> Hello friend")
```

8. Create a target for the block friend. This will be the position she will move toward. Initially, set the target as the block friend's current position, this is so the block friend doesn't try and move anywhere when the program starts:

```
target = friend.clone()
```



To copy positions in Minecraft, you use the `clone()` function. This is because the positions returned by the Minecraft API are Python objects and they are different to normal variables. For example, if you had used the code `target = friend` and later changed the `friend.x` value in the `friend` object, the `target.x` value would be changed in `target` too.

9. Start an endless loop, so the program will run forever. (Note that all the code after this point is indented under this loop.)

```
while True:
```

10. Get the player's position and calculate the distance between the player and the block friend using the `distanceBetweenPoints()` function:

```
pos = mc.player.getTilePos()
distance = distanceBetweenPoints(pos, friend)
```

11. Apply the rules you want the block friend to use to work out what to do next. If it's "happy", tell it to compare the distance between the friend and the "too far away" constant. If the distance is less than the "too far away" constant, set the target to be the position of the player. If it's greater than the "too far away" constant, change the block's mood to "sad" (see Figure 8-2):

```
if blockMood == "happy":
    if distance < TOO_FAR_AWAY:
        target = pos.clone()
    elif distance >= TOO_FAR_AWAY:
        blockMood = "sad"
        mc.postToChat("<block> Come back. You are too far ↵
                     away. I need a hug!")
```

12. Now you need to tell the program that otherwise (that is, if the distance is not less than the "too far away" constant), the block friend is "sad" and in that case to wait until the player is within one block's distance (a hug) before changing the block's mood to "happy":

```
elif blockMood == "sad":
    if distance <= 1:
        blockMood = "happy"
        mc.postToChat("<block> Awww thanks. Let's go.")
```



FIGURE 8-2 The block friend is sad.

13. The next section of code moves the block friend to his target:

```
if friend != target:
```

14. Next, find all the blocks between the friend and his target by using the `getLine()` function in `MinecraftDrawing`:

```
blocksBetween = mcdrawing.getLine(  
    friend.x, friend.y, friend.z,  
    target.x, target.y, target.z)
```

The `getLine()` function works in the same way as `drawLine()` (see Adventure 7). However, instead of drawing the line in blocks, it returns a list of points (x, y, z) that make a line between the two sets of x, y, z co-ordinates passed as parameters.

15. Directly under the previous code, you need to tell your program to loop through all the blocks between the friend and the target, and move the block friend to the next block between the block friend and the player:

```
for blockBetween in blocksBetween[:-1]:  
    mc.setBlock(friend.x, friend.y, friend.z, block.AIR.id)  
    friend = blockBetween.clone()  
    friend.y = mc.getHeight(friend.x, friend.z)  
    mc.setBlock(friend.x, friend.y, friend.z,  
               block.DIAMOND_BLOCK.id)  
    time.sleep(0.25)  
    target = friend.clone()
```

When the `for` loop finishes, the block friend has reached her target, so set the target to be the block friend's own current position. This way she won't try to move again.

The program moves the block friend by clearing her from her previous last position (which it does by setting the block to `AIR`), updating the block friend's position to be the next block in the line, then recreating the block friend in that position.

Note the difference between `blockBetween` and `blocksBetween`, the first contains the block where the friend is, the second contains a list of all blocks between where the friend started and where she is moving too.



The speed the block friend travels is set by how long the program sleeps between each block move—`time.sleep(0.25)`. If this wait is too short, the block will move too quickly and the player will never be able to get away from the block. If it's too long, the block will move so slowly it will drive you mad with frustration!

16. Put in a small delay at the end of the loop so the program doesn't overload Minecraft by constantly looping and asking for the player's position:

```
time.sleep(0.25)
```

17. Run the program.

The block friend will appear next to your player and follow him until the pair get too far away from each other. When that happens, the block friend will come to a stop and your player will have to go back and stand next to her before she will start following him again.

You can download the complete code for the block friend program from the Adventures in Minecraft companion website at www.wiley.com/go/adventuresinminecraft.

DIGGING INTO THE CODE

To calculate the distance between the block and the player, you created a function called `distanceBetweenPoints(point1, point2)`. This uses Pythagoras' theorem to calculate an accurate distance between two points. You may have investigated how Pythagoras' theorem would work in the challenge in “Adding a Homing Beacon” in Adventure 4?

The function calculates the distance by:

1. Working out the difference between the x,y,z values in `point1` and `point2`:

```
xd = point2.x - point1.x  
yd = point2.y - point1.y  
zd = point2.z - point1.z
```

2. Calculating the squares of the difference between the points:

```
xd*xd
```

3. Adding together all the squares:

```
(xd*xd) + (yd*yd) + (zd*zd)
```

4. Returning the difference between the two points, which is the square root of the sum of the squares using the Python function `math.sqrt()`:

```
return math.sqrt((xd*xd) + (yd*yd) + (zd*zd))
```

Visit www.mathsisfun.com/algebra/distance-2-points.html to learn about Pythagoras' theorem and how to use it to calculate the distance between two points.

The block friend is moved toward the player by finding the blocks between the block friend and the player, and looping through those blocks:

```
for blockBetween in blocksBetween[:-1]:
```

The `[:-1]` tells Python to loop through all the blocks in the `blocksBetween` list until it gets to the last but one block. This way, when the block friend is moved toward the player, she stands next to the player and not on top of him.

The **square root** of a number is the value that can be multiplied by itself to give that number. For example, the square root of 9 is 3, because $3 \times 3 = 9$.



CHALLENGE



1. The block friend always moves at the same speed: one block about every 0.25 seconds, or 4 bps (blocks per second). However, a real friend would speed up the further away her target got. Try changing the program so that the further the block friend is from the player, the quicker she moves.
2. Revisit “Adding a Homing Beacon” in Adventure 4 and complete the challenge again, this time building a better distance estimator using the `distanceBetweenPoints()` function.

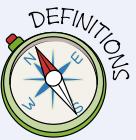
Using Random Numbers to Make Your Block Friend More Interesting

The problem with the block friend program you just created is that it is **predictable**; it’s always going to do the same thing. After you’ve run the program only a couple of times, you’ll know exactly what it’s going to do and when, so it’s going to get boring very quickly. To really give your block friend a “mind”, you need to give it an air of unpredictability.



When something is **predictable**, it means you are able to foresee what is going to happen before it does. This isn’t great if you want it to act as if it’s real. Real things are not always predictable.

By using random numbers, you can simulate unpredictability—in other words, make something look as if it’s acting unpredictably. You do this by making a program choose to do things based on a **probability**; for example, you might tell it to do a particular action 1 every 100 times. By adding more rules based on different odds, you can make the program much more difficult to predict. Before changing the block friend program to use random numbers, let’s explore the code to create random numbers and probability checks. You may remember random numbers being introduced in Adventure 3.



A **probability** is the measurement of how likely it is that something will happen, i.e. when flipping a coin there is a 50% (or 1 in 2) chance that it will land on heads.

The Python `random` module contains the function `random.randint(startNumber, endNumber)` which is used to generate a random number between two specific numbers (`startNumber, endNumber`).

The following code will print a random number between 1 and 10 each time it is run. If you want to see the results, you should create a new python program:

```
import random
randomNo = random.randint(1,10)
print randomNo
```

By adding an `if` statement to check when the random number is 10, you create a probability check that will be true in approximately 1 time in every 10:

```
import random
if random.randint(1,10) == 10:
    print "This happens about 1 time in 10"
else
    print "This happens about 9 times out of 10"
```

If you were to run the program above 100 times you would expect to see “This happens about 1 time in 10” printed about 10 times (see Figure 8-3), but you might only see it 9 or 11 times, or maybe even not at all. Its unpredictable!

If you were to run this program 100 times, how many times would you expect to see “This happens about 1 time in 10” printed? You would expect to see it 10 times, but you *might* not, you *might* not see it all, or you *might* see it 100 times!



If you use random numbers and a probability check in your block friend program, you can make it less predictable. You can even make the block friend “unfriend” your player!

Add some new rules to the block friend program so that if the block friend is “sad” there is a 1 in 100 chance she will decide she has had enough of waiting and will not follow your player if he comes back and gives her a hug:

1. Open IDLE and open the `BlockFriend.py` program from the `MyAdventures` folder.
2. Click File→Save As and save the file as `BlockFriendRandom.py`.

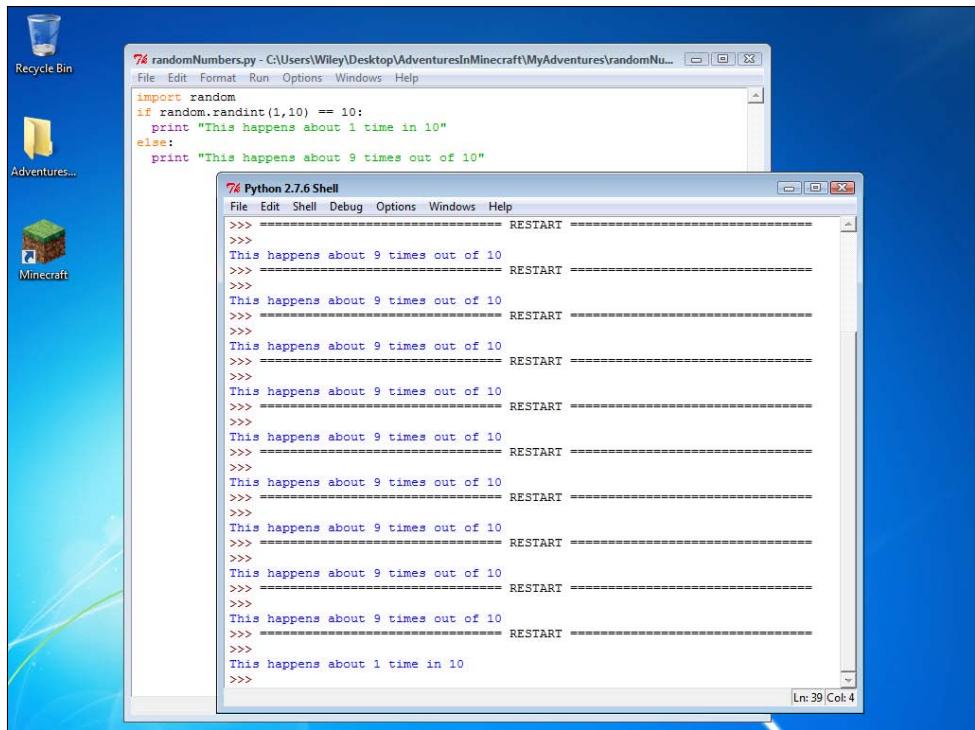


FIGURE 8-3 Creating random numbers to give your block friend an air of unpredictability

3. Add the `random` module to the `import` statements at the top of the program (the code in **bold**):

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import mcpi.minecraftstuff as minecraftstuff
import math
import time
import random
```

4. Add a 1 in 100 random number test to change the block's mood to "had enough" when the block is "sad":

```
elif blockMood == "sad":
    if distance <= 1:
        blockMood = "happy"
        mc.postToChat("<block> Awww thanks. Let's go.")
if random.randint(1,100) == 100:
```

```
blockMood = "hadenough"
mc.postToChat("<block> That's it. I have had ←
enough.")
```

5. Run the program.

When your player gets too far away from the block friend, if you wait long enough (for a 1 in 100 chance to come true), the block friend will decide she has had enough and ‘unfriend’ the player! Once that happens, the block friend will say, “That’s it. I have had enough” and will sit there forever.

CHALLENGE

Change the program again so that there is a 1 in 50 chance that if the block friend’s mood is “**hadenough**” she will forgive the player, if he gives her a hug.



Bigger Shapes

In the block friend program, you made one block to move around and follow the player. But what if you wanted to make a lot of blocks move around? Or how about a shape made out of moving blocks, like a car or an alien spaceship?

This is where things become much more difficult, because you need to keep track of lots of blocks. Each time you wanted to make it move you would need to set all the blocks to **AIR** and then recreate the blocks in their new position. If there are a lot of blocks, the shape will stop moving properly and slow down to a crawl.

The **minecraftstuff** module contains functions called **MinecraftShape**, which has been written specially to create shapes and move them around. It does this by keeping track of all the blocks that make up a shape; when the shape is moved, it only changes the blocks that have changed, not all of them.

To use **MinecraftShape**, you have to tell it what the shape looks like by creating a list of blocks that make up the shape. Each of the blocks in the shape has a position (x, y, z) and a block type.

Figure 8-4 shows a simple shape made up of seven blocks, together with the positions of the blocks. In this case, “position” isn’t the same as the position in Minecraft. You’ll see that the centre of the wooden horse is $0,0,0$ and each block is relative to the centre, so the block to the right is $1,0,0$.

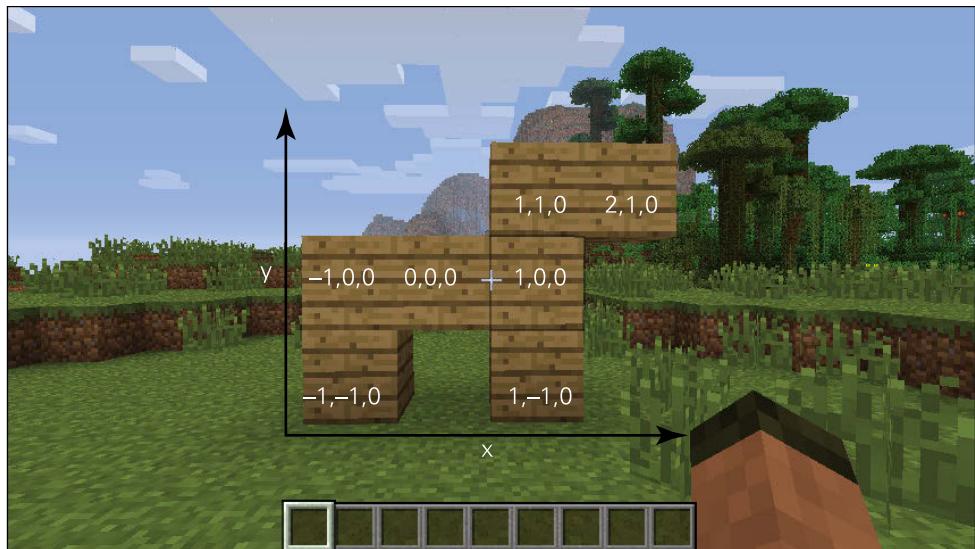


FIGURE 8-4 Block positions of wooden horse Minecraft shape

Now you’re going to create a program that uses `MinecraftShape` to create the wooden horse in Figure 8-4 and make it move:

1. Open IDLE, and click File→New File to create a new program. Save the file as `WoodenHorse.py` in the `MyAdventures` folder.
2. Import the `minecraft`, `block`, `minecraftstuff` and `time` modules:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import mcpi.minecraftstuff as minecraftstuff
import time
```

3. Create the `Minecraft` object:

```
mc = minecraft.Minecraft.create()
```

4. Create the wooden horse shape by creating a list of blocks using `ShapeBlock` to define the relative position and block type of each block:

```
horseBlocks = [
    minecraftstuff.ShapeBlock(0,0,0,block.WOOD_PLANKS.id),
    minecraftstuff.ShapeBlock(-1,0,0,block.WOOD_PLANKS.id),
    minecraftstuff.ShapeBlock(1,0,0,block.WOOD_PLANKS.id),
    minecraftstuff.ShapeBlock(-1,-1,0,block.WOOD_PLANKS.id),
```

```
minecraftstuff.ShapeBlock(1,-1,0,block.WOOD_PLANKS.id),  
minecraftstuff.ShapeBlock(1,1,0,block.WOOD_PLANKS.id),  
minecraftstuff.ShapeBlock(2,1,0,block.WOOD_PLANKS.id)]
```

The position of the blocks is the same as shown in Figure 8-4.

5. You will need to tell `MinecraftShape` where in the Minecraft world to create the wooden horse. Get the player's position and add 1 to the z and y co-ordinates, so it isn't directly on top of the player:

```
horsePos = mc.player.getTilePos()  
horsePos.z = horsePos.z + 1  
horsePos.y = horsePos.y + 1
```

6. Create the wooden horse by using `MinecraftShape`, passing the Minecraft object, the position where the shape should be created and the list of `ShapeBlocks` which make up the shape, as parameters:

```
horseShape = minecraftstuff.MinecraftShape(mc, horsePos,  
                                              horseBlocks)
```

7. Run the program. Voila! You should see a wooden horse appear next to the player.
8. Modify the `WoodenHorse.py` program to make the horse move by adding the following code to the bottom of the program:

```
for count in range(1,10):  
    time.sleep(1)  
    horseShape.moveBy(1,0,0)  
horseShape.clear()
```

The `moveBy(x,y,z)` function tells the shape to “move by” the number of blocks in x, y, z. So in this example, the `horseShape` is moved by one block across (x). The `clear()` function removes the shape, setting all the blocks to `AIR`.

9. Run the program and watch the wooden horse gallop!

As well as `moveBy(x,y,z)` and `clear()`, you can also tell a shape to `move(x,y,z)` to any position in Minecraft. If the shape has been cleared you can recreate it with `draw()`.



You can download the complete code for the wooden horse from the Adventures in Minecraft companion website at www.wiley.com/go/adventuresinminecraft.

You'll find shapes useful now, because you're going to create an alien spaceship! Later, you'll be using them again to create obstacles.

Alien Invasion

Aliens are about to invade Minecraft. A spaceship will come down from the sky directly above your player, who is in mortal danger—these aliens are *not* friendly and they won't give up until they have completed their mission.

In the next program, you will use `MinecraftShape` and the programming techniques you used in the block friend program to create an alien spaceship (see Figure 8-5) which will hover above the surface of the world, chasing your player, trying to get above him. And when it succeeds, it is going to beam him onboard.



FIGURE 8-5 Create an alien spaceship to spice up your Minecraft game!

You create the alien spaceship using `MinecraftShape`, just like in the wooden horse program with each block in the shape having its own relative position and block type. Figure 8-6 shows the positions of the shape's blocks from the side and from above.



FIGURES 8-6 Alien spaceship block positions

The alien spaceship shows you how you can create three-dimensional shapes using the `MinecraftShape` object. The shapes can be as big or small, and as simple or complex as you like. This gives you lots of different options for using shapes in your Minecraft programs.



Like the block friend program, the code for the alien invasion will be in two parts. The first part is the rules to decide what the spaceship will do next; the second part is the code that moves the alien spaceship toward the player.

When the alien spaceship is chasing the player, it will taunt him by posting messages (like “you can’t run forever”) to the chat. The messages are picked at random from a list of taunts (see Figure 8-7).

The rules to decide what the alien spaceship will do next will be based on one of three modes:

- **Landing:** When the program starts, this will be the spaceship’s initial mode and it will come down from sky directly above the player.
- **Attack:** As soon as the spaceship lands, it starts to attack, constantly chasing the player until it is directly above him, from where it will “beam” the him inside.

- **Mission accomplished:** This mode is set after the player has been beamed inside the ship and the aliens are ready to return him to earth. At this point the program will finish and the player will be beamed back.

Once the alien spaceship has captured the player, the program will build a dismal room in which to hold him and change his position to be inside it (see Figure 8-8). The aliens will then post messages to the player before beaming him back by changing his position back to what it was and clearing the room.



FIGURE 8-7 The alien spaceship gives chase.

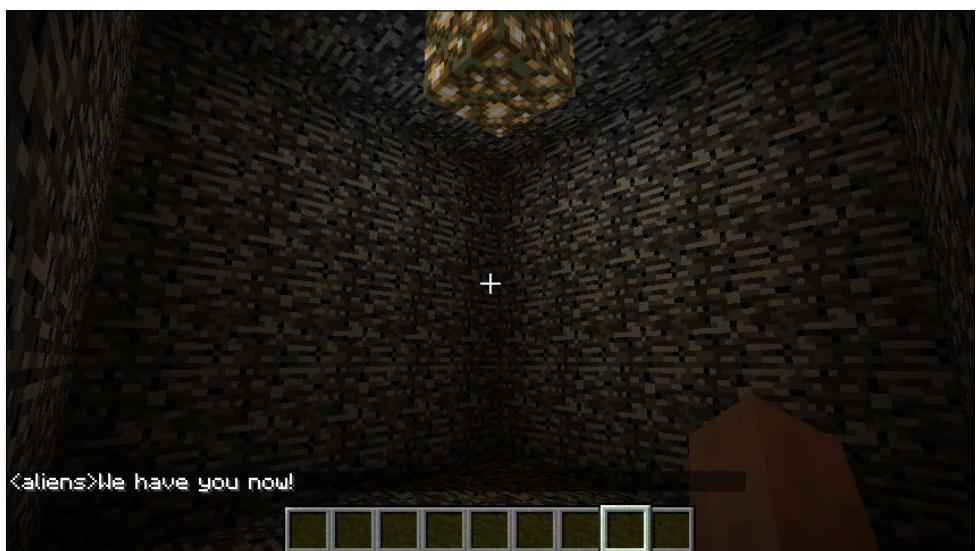


FIGURE 8-8 Inside the alien spaceship

Use the following steps to create the alien invasion program:

1. Open IDLE, click New → New File and save the file as `AlienInvasion.py` in the `MyAdventures` folder.
2. Import the `minecraft`, `block`, `minecraftstuff` and `time` modules:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import mcpi.minecraftstuff as minecraftstuff
import time
```

3. Create the `distanceBetweenPoints()` function:

```
def distanceBetweenPoints(point1, point2):
    xd = point2.x - point1.x
    yd = point2.y - point1.y
    zd = point2.z - point1.z
    return math.sqrt((xd*xd) + (yd*yd) + (zd*zd))
```

4. Create the constants for the program. `HOVER_HEIGHT` is the number of blocks the alien spaceship will hover over the player; `ALIEN_TAUNTS` is a list of the taunts that will be posted to the chat while the aliens are chasing the player:

```
HOVER_HEIGHT = 15
ALIEN_TAUNTS = ["<aliens>You cant run forever",
                 "<aliens>Resistance is useless",
                 "<aliens>We only want to be friends"]
```

You can change the aliens' taunts—see how creative you can be! Or add more if you like.

5. Create the `Minecraft` and `MinecraftDrawing` objects:

```
mc = minecraft.Minecraft.create()
mcstuff = minecraftstuff.MinecraftDrawing(mc)
```

6. Set the aliens' starting position and mode, which will be 50 blocks directly above the player and “landing”:

```
alienPos = mc.player.getTilePos()
alienPos.y = alienPos.y + 50
mode = "landing"
```

7. Create the alien spaceship using `MinecraftShape` (look at Figure 8-6 for a reminder of how this is done):

```
alienBlocks = [
    minecraftstuff.ShapeBlock(-1, 0, 0, block.WOOL.id, 5),
```

```
minecraftstuff.ShapeBlock(0,0,-1,block.WOOL.id, 5),  
minecraftstuff.ShapeBlock(1,0,0,block.WOOL.id, 5),  
minecraftstuff.ShapeBlock(0,0,1,block.WOOL.id, 5),  
minecraftstuff.ShapeBlock(0,-1,0,  
                           block.GLOWSTONE_BLOCK.id),  
minecraftstuff.ShapeBlock(0,1,0,  
                           block.GLOWSTONE_BLOCK.id)]  
  
alienShape = minecraftstuff.MinecraftShape(mc, alienPos,  
                                             alienBlocks)
```

8. Create a `while` loop, which will continue to loop while the mode is not "`missionaccomplished`" or to put it another way, will exit when the mode is "`missionaccomplished`":

```
while mode != "missionaccomplished":
```

9. Get the player's position each time around the loop:

```
playerPos = mc.player.getTilePos()
```

10. The next section of code relates to the rules the program will use to decide what to do next—if the mode is "`landing`", set the alien target (where the alien spaceship will travel to) to be above the player's position and set the mode to "`attack`":

```
if mode == "landing":  
    mc.postToChat("<aliens> We dont come in peace - please ↪  
                  panic")  
    alienTarget = playerPos.clone()  
    alienTarget.y = alienTarget.y + HOVER_HEIGHT  
    mode = "attack"
```

11. Otherwise, if the mode is "`attack`", check to see if the alien spaceship is above the player. If it is, beam her inside the ship and set the mode to "`missionaccomplished`". Otherwise, if the player has got away, set the alien target to be the player's current position and post a taunt to the chat:

```
elif mode == "attack":  
    #check to see if the alien ship is above the player  
    if alienPos.x == playerPos.x and alienPos.z == ↪  
        playerPos.z:  
        mc.postToChat("<aliens>We have you now!")  
  
        #create a room  
        mc.setBlocks(0,50,0,6,56,6,block.BEDROCK.id)
```

```

mc.setBlocks(1,51,1,5,55,5,block.AIR.id)
mc.setBlock(3,55,3,block.GLOWSTONE_BLOCK.id)

#beam up player
mc.player.setTilePos(3,51,5)
time.sleep(10)
mc.postToChat("<aliens>Not very interesting at all - ↪
               send it back")
time.sleep(2)

#send the player back to the original position
mc.player.setTilePos(playerPos.x, playerPos.y,
                      playerPos.z)

#clear the room
mc.setBlocks(0,50,0,6,56,6,block.AIR.id)

mode = "missionaccomplished"

else:
    #the player got away
    mc.postToChat(ALIEN_TAUNTS[random.
        randint(0,len(ALIEN_TAUNTS)-1)])
    alienTarget = playerPos.clone()
    alienTarget.y = alienTarget.y + HOVER_HEIGHT

```

When the player is beamed aboard the spaceship, a room is built 50 blocks over the spawn position and the player's position is set to be inside the room. (Just like Doctor Who's Tardis, the inside is bigger than the outside!) Messages are then posted to the chat, before the player's position is set to be back where he started and the room is cleared.

The room into which the player is beamed is created in the sky above the spawn location, partly for convenience but also because it's likely to be well away from anything else. You could create it anywhere you like, however, because as soon as the player leaves the room it is cleared, returning the Minecraft world to the way it was before.



- If the position of the alien spaceship is not equal to the target (set in the rules above), move the alien spaceship to the target by typing this code into the end of the program, indented under the `while` loop:

```
if alienPos != alienTarget:
```

```
blocksBetween = mcdrawing.getLine(  
    alienPos.x, alienPos.y, alienPos.z,  
    alienTarget.x,alienTarget.y,alienTarget.z)  
for blockBetween in blocksBetween:  
    alienShape.move(blockBetween.x, blockBetween.y,  
                    blockBetween.z)  
    time.sleep(0.25)  
alienPos = alienTarget.clone()
```

13. At this point the program will return to the top of the `while` loop. When the mode has been set to "`missionaccomplished`" and the while loop finishes, the last line of the program is to make the alien spaceship disappear:

```
alienShape.clear()
```

14. Run the program and watch out for the aliens who will come down from the sky directly above you.

You can download the complete code for the alien invasion from the Adventures in Minecraft companion website at www.wiley.com/go/adventuresinminecraft.

DIGGING INTO THE CODE

Each time the aliens chase the player, a random taunt is picked from the constant `ALIEN_TAUNTS` and posted to the chat.

```
mc.postToChat (ALIEN_TAUNTS [  
    random.randint (0, len(ALIEN_TAUNTS) - 1) ] )
```

`ALIEN_TAUNTS` is a list of strings; the `random.randint()` function is used to pick an item from the list, randomly picking a number between 0 and the length of the `ALIEN_TAUNTS` list minus 1.

1 is taken away from the length of the list because, while `len()` returns the actual number of items in the list (i.e. 3), but you have to reference items in the list starting at 0 (i.e. 0,1,2).

CHALLENGE



The alien spaceship is really simple. Try creating an amazing spaceship, one you really like. Change the spaceship so that when it lands it goes into "lurk" mode, where it will stay close to the player but not attack; and then, based on a random probability, switches without warning to attack mode.

Further Adventures in Simulation

In this adventure you have used algorithms and rules to simulate a friend and an alien invasion—how about taking it further and simulating other things such as a flock of birds (or blocks), waves across the oceans in Minecraft in or a complete cellular system such as Conway’s Game of Life made out of blocks.

To find out more about Conway’s game of life visit [env.wikipedia.org/wiki/Conway's_Game_of_Life](http://en.wikipedia.org/wiki/Conway's_Game_of_Life).

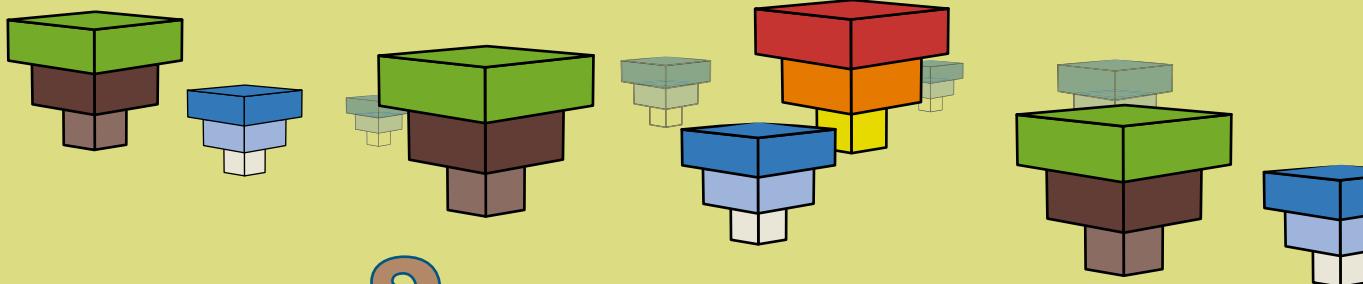
Quick Reference Table	
Command	Description
<code>import random</code>	Imports the Python <code>random</code> module
<code>random.randint(start, end)</code>	Creates a random number between the start and end numbers
<code>import minecraftstuff</code>	Imports the <code>minecraftstuff</code> extensions module, which is included in the Starter Kit
<code>mc.getHeight(x, z)</code>	Gets the height (y co-ordinate) of the land (i.e. the first block down from the sky that isn’t AIR) for a horizontal position (x,z)
<code>mcdrawing = minecraftstuff. ← MinecraftDrawing(mc)</code>	Creates the <code>MinecraftDrawing</code> object
<code>copyOfPosition = position.clone()</code>	Creates a copy (clone) of a Minecraft position
<code>mcdrawing.getLine(x1, y1, z1, x2, y2, z2)</code>	Gets all the blocks in a line between two points
<code>shapeBlocks = [minecraftstuff.ShapeBlock(1, 0, 0, block.DIRT.id), minecraftstuff.ShapeBlock(0, 0, 1, block.DIRT.id)]</code>	Creates a list of <code>shapeBlocks</code> that represent a <code>MinecraftShape</code>
<code>shape = minecraftstuff. ← MinecraftShape(mc, pos, shapeBlocks)</code>	Creates a <code>MinecraftShape</code> of the <code>shapeBlocks</code> passed to it at the position specified
<code>shape.moveBy(x, y, z)</code>	Moves a <code>MinecraftShape</code> by the value in x,y,z
<code>shape.move(x, y, z)</code>	Moves a <code>MinecraftShape</code> to the position x,y,z
<code>shape.clear()</code>	Clears the <code>MinecraftShape</code>
<code>shape.draw()</code>	Draws the <code>MinecraftShape</code>



Achievement Unlocked: Abducted by your own artificial intelligence!

In the Next Adventure...

In the next adventure, you will use all the skills you have learnt throughout your adventures so far to create a game in which you will race against the clock to collect diamonds. But watch out—there are obstacles in the way and they are determined to stop you!



Adventure 9

The Big Adventure: Crafty Crossing

YOU HAVE ARRIVED at your last big adventure! You will be using all the skills you have learned on your journey to create your own fully functional mini-game in Minecraft. You will realise just how versatile the Minecraft world is, as well as how you can achieve remarkable things just by using simple commands to get and set a block, and get and set the players' positions.

You will also learn a new program skill and use *threading* to make your program do more than one thing at once.

This project can be extended in many ways, and once you've finished, it doesn't mean you're at the end of your adventure: instead, you will have arrived at the starting point for creating even more creative, sophisticated and challenging games.

A Game within a Game

The game you're going to create here is called Crafty Crossing. The objective of the game is to collect diamonds and get to the other side of the arena before the timer runs out, all the time navigating past a series of pesky obstacles that are designed to slow you down.

You score points for each diamond you collect, and your points are then multiplied by the number of seconds left on the clock when your player reaches the other side.

To get across the arena in the shortest time possible, your player will need to jump onto a moving platform over a river, get under a wall that moves up and down, and avoid holes that randomly appear in the arena floor (see Figure 9-1).



FIGURE 9-1 Create the Crafty Crossing game.

VIDEO



There is a video of the complete Crafty Crossing game on the companion website at www.wiley.com/go/adventuresinminecraft.

Creating a game is a major mission. You will need to build, test and finally put together a lot of components before your challenging game is ready to go. To make this project easier to follow, I have broken down the instructions into four parts, so you can develop and test the program in sections:

- Part 1: Create the main structure and framework of the program and building the game arena where all the action will take place.
- Part 2: Program the obstacles that will get in the player's way and try to slow him down.
- Part 3: Introduce the 'game play' to the program, creating levels of difficulty, scoring and, of course, the inevitable "Game Over".

- Part 4: Use the skills you learned in Adventure 5 and re-using your button and 7-segment display hardware to add a start button, a diamond countdown and an indicator to alert the player that time is running out.

Once your game is complete, you can take it in whatever direction you want, introduce your own creativity and make it your own.

Suggested challenges are included in each part of the adventure. You should not take up these challenges or change the program until you have completed the whole adventure, otherwise you may find yourself tangled up in a whole lot of complications! Come back to them after you've finished the adventure.



Part 1—Building the Arena

The game arena for Crafty Crossing is where all the action is going to take place. The player will start at one end and try to get to the other—but there will be obstacles littering the arena to slow him down.

Before there are any obstacles in it, the arena is a rectangle of **GRASS** blocks that makes up the floor, and **GLASS** walls that go all the way round (see Figure 9-2). You will use constants to define the width, height and length (or the x, y and z) of the arena. By modifying the constants, you will be able to change the size and shape of the arena and the obstacles will automatically resize to fill the space. The arena floor will need to be three blocks deep as the river obstacle goes down two blocks.



FIGURE 9-2 Your game arena will look like this.

Remember the `setBlocks()` function you used to build a house in Adventure 3? You are now going to use it to build your arena.

Start up Minecraft, IDLE, and if you are working on a PC or a Mac, start up the Bukkit server too. You should have had a bit of practice with starting everything up by now, but refer back to Adventure 1 if you need any reminders.

Start by creating a new program for the Crafty Crossing game. The first step is to set up the initial structure of the game:

1. Open IDLE, create a new file and save the file as `CraftyCrossing.py` in the `MyAdventures` folder.
2. Import the modules you need. You will use a new module `thread`, which will be described in Part 2 when you create the obstacles:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
import mcpi.minecraftstuff as minecraftstuff
import time
import random
import thread
```

3. Create three constants for the game arena, which are its width, height and length as `x, y, z`:

```
ARENAX = 10
ARENAZ = 20
ARENAY = 3
```

4. Now insert the functions for the program. You will build these through the adventure, until you have a complete game:

```
def createArena(pos):
    pass

def theWall(arenaPos, wallZPos):
    pass

def theRiver(arenaPos, riverZPos):
    pass

def theHoles(arenaPos, holesZPos):
    pass

def createDiamonds(arenaPos, number):
    pass
```

The statement `pass` doesn't do anything, but it's useful as a placeholder to mark where code will be created in the future.

5. Create the connection to Minecraft:

```
mc = minecraft.Minecraft.create()
```

6. Create a Boolean variable, which will be set to `True` when the game is over, but at the start of the game is set to `False`:

```
gameOver = False
```

As you continue, you will either be adding more code to the main program or completing the functions you have just created.

You can run the program at this point. Nothing will happen! But it's still a good idea to run the program as it is, because if no errors are displayed in the Python Shell you know that everything is set up correctly.



The next step is to update the `createArena` function, which builds the arena in Minecraft:

1. Start by finding the `createArena` function in the code you have just written, by looking for the following code:

```
def createArena(pos):  
    pass
```

Delete the `pass` statement that is indented under the function.

2. Indented under the `def createArena(pos):` line, create a connection to Minecraft:

```
mc = minecraft.Minecraft.create()
```

3. Now you can create the arena using `setBlocks` by taking the position (`pos`) passed to the function and adding the constants `ARENAX`, `ARENAY` and `ARENAZ` (Figure 9-3 shows how the constants are added to the position to create the arena):

```
mc.setBlocks(pos.x - 1, pos.y, pos.z - 1,  
            pos.x + ARENAX + 1, pos.y - 3,  
            pos.z + ARENAZ + 1,  
            block.GRASS.id)
```

4. Next, create the glass walls by creating a cube of **GLASS** blocks and then clearing the area in the middle by creating a cube of **AIR** inside the **GLASS**:

```
mc.setBlocks(pos.x - 1, pos.y + 1, pos.z - 1,  
            pos.x + ARENAX + 1, pos.y + ARENAY,  
            pos.z + ARENAZ + 1,  
            block.GLASS.id)  
  
mc.setBlocks(pos.x, pos.y + 1, pos.z,  
            pos.x + ARENAX, pos.y + ARENAY,  
            pos.z + ARENAZ,  
            block.AIR.id)
```

5. The **createArena()** function is now complete, but it still needs to be called from the main program. Add the following to the bottom of the program to get the player's position and pass it as a variable to the **createArena()** function:

```
arenaPos = mc.player.getTilePos()  
createArena(arenaPos)
```

6. It's time to run the program! You should see the arena built next to the player, as in Figure 9-3.



FIGURE 9-3 Create the game arena.

CHALLENGE

The arena is functional but it's a bit boring! Can you build a better one? Perhaps add some arrows to the floor to show the direction the player needs to go, or give it some decoration around the sides. Or how about giving it a roof adorned with blazing torches?



Part 2—Creating the Obstacles

Making a game challenging is one of the aspects of good game play—an easy game very quickly becomes a dull game. In the next part of this adventure, you will create some obstacles to slow down the player and make it more difficult for him to get to the other side of the arena.

The Wall

The first obstacle you are going to create is a brick wall that not only goes all the way across the arena but also moves up and down. When it's down it blocks the way, meaning the player has to wait until it goes up before he can pass through (see Figure 9-4).



FIGURE 9-4 The wall you build can slow down the player.

The wall is a simple but very effective way of slowing down the player, but if he gets his timing right he can duck straight under it and move onto the next challenge.

You will create the wall using `MinecraftShape` in the `minecraftstuff` module, which you learned about when you created the Alien Invasion in Adventure 8.

Update the `theWall()` function to create the wall obstacle by following these steps:

1. Find the function in the code you wrote earlier by looking for the following code:

```
def theWall(arenaPos, wallZPos):  
    pass
```

Delete the `pass` statement that is indented under the function.

2. Indented under the `def theWall(arenaPos, wallZPos):` line, create a connection to Minecraft:

```
mc = minecraft.Minecraft.create()
```

3. The `theWall()` function expects two parameters to be passed to it: `arenaPos` (the position of the arena); and `wallZPos` (the Z position along the arena where the wall will be placed). Using these two parameters, create the position for the wall (see Figure 9-5):

```
wallPos = minecraft.Vec3(arenaPos.x, arenaPos.y + 1,  
                         arenaPos.z + wallZPos)
```

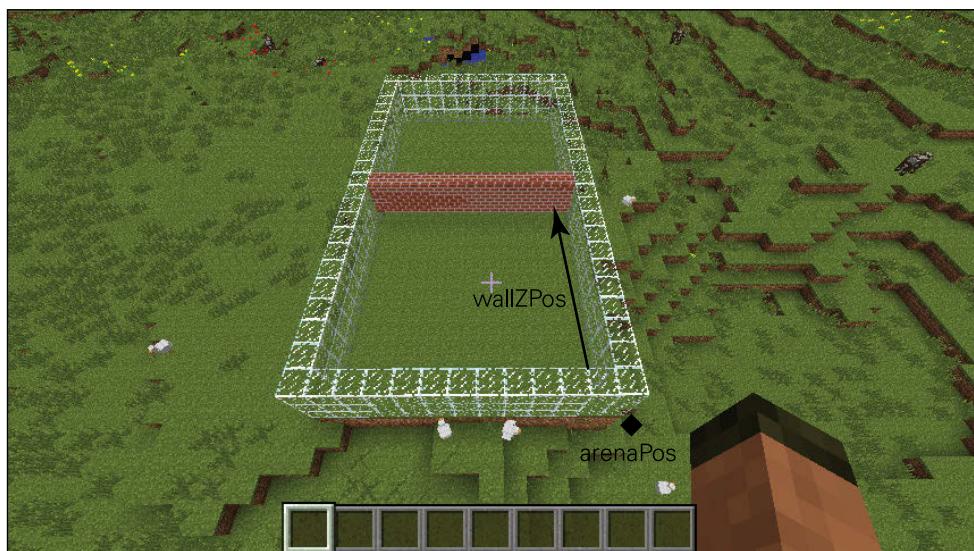


FIGURE 9-5 Create the position of the wall.

4. You will now create the shape blocks that will make up the wall by using two `for` loops—one that creates the blocks across the arena (x) and one that creates the blocks upwards (y)—and adding them to the `wallBlocks` list:

```
wallBlocks = []
for x in range(0, ARENAX + 1):
    for y in range(1, ARENAY):
        wallBlocks.append(minecraftstuff.ShapeBlock(x,
                                                       y,
                                                       0,
                                                       block.BRICK_BLOCK.id))
```

5. Create the shape of the wall by passing in the `wallPos` and `wallBlocks` variables that you created in steps 3 and 4:

```
wallShape = minecraftstuff.MinecraftShape(mc, wallPos,
                                             wallBlocks)
```

6. Your wall is now finished—but wait, it's still static! To move the wall up and down, you use the `MinecraftShape moveBy()` function, putting a small delay in between:

```
while not gameOver:
    wallShape.moveBy(0,1,0)
    time.sleep(1)
    wallShape.moveBy(0,-1,0)
    time.sleep(1)
```

The code inside the `while` loop will run until the variable `gameOver` is set to `True` (or `not gameOver`). This will be set to `True` at the end of the game when the player either completes the game or loses.

7. The `theWall()` function is now complete. All that is left for you to do is call the function in the main program and add the following to the bottom of the program:

```
WALLZ = 10
theWall(arenaPos, WALLZ)
```

The constant `WALLZ` holds the Z position, which is where you want the wall to be built in the arena.

8. Run the program. You should see the wall going up and down in the middle of the arena.

You haven't written the code that will set the `gameOver` variable to `True` yet, so when you run the program it will continue forever! You will need to stop the program by clicking Shell⇒Restart Shell on the Python Shell.



DIGGING INTO THE CODE

When you created the blocks for the wall, you used two `for` loops—one inside the other. This is a really useful programming technique known as nesting. You can also say that the loops are nested. The first `for` loop (`for x in range(0, ARENAX + 1):`) loops once for each block in across the width (x) of the arena. The second loop (`for y in range(1, ARENAY):`) does the same, but for each block from 1 to the height (y) of the arena.

The x and y variables are used to create a `ShapeBlock`, `minecraftstuff.ShapeBlock(x, y, 0, block.BRICK_BLOCK.id)` for each block in the wall.

When you use nested `for` loops and the `ARENAX` and `ARENAY` constants, it means that even if the arena is a different size, the wall will always fill the area.

Running More Than One Obstacle

You now have a problem! The program you created is stuck. It will loop around, moving the wall up and down forever and is never going to do anything else. This is because the program you have written is **sequential**, meaning that each command in your program is run in turn, and the next command will wait for the previous one to finish before running. Your program is stuck, because it never gets to the next command after the `while` loop.



Sequential means to follow a sequence or order, usually one thing after the other.

How can you create more obstacles and run the rest of the game if the program is stuck and won't do anything other than move the wall up and down?

There is a solution. Meet multi-threading! Until now, all the programs you have written in these adventures have been single-threaded; in other words, they have been sequential, with one command running after the other.

If you imagine your program as a piece of string laid out flat and your commands as knots in that string, you can see how your program runs from one end of the string to the other, running a command whenever it gets to a knot. If you include a loop, it will make your program go back to a previous knot, and if you include an `if` statement it might make the program miss a knot—but it can still only run one command at a time.

When you use multi-threading, you are telling your program to create a new thread (or piece of string), which has its own commands (knots). This will run at the same time as your original program, which keeps on running too. Your program is now doing two things at once, rather than just one (see Figure 9-6).

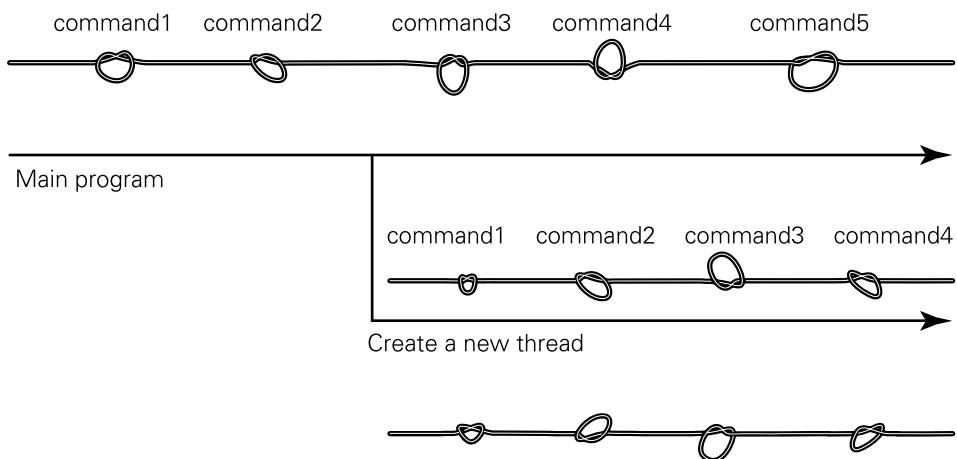


FIGURE 9-6 When you create multiple threads, your program can do more than one thing at a time.

To make all the obstacles in the game run at the same time, you need to create a new thread each time an obstacle is run, meaning that the programs that make the obstacles work are all running at the same time.

Multi-threading is incredibly useful in computer programming, but it is very advanced and it can get complicated very quickly. If you want to know more about multi-threading in Python, visit www.tutorialspoint.com/python/python_multithreading.htm.



Let's return to your wall. To get it running in its own thread, you have to change the line of the program that calls the `theWall()` function, and use `thread.start_new_thread()`:

1. Delete the last line of the program that runs the `theWall()` function, which looks like this:

```
theWall(arenaPos, WALLZ)
```

- Insert the following line at the end of the program to call the `theWall()` function—but this time it's going to be starting in a new thread:

```
thread.start_new_thread(theWall, (arenaPos, WALLZ))
```

- Run the program.

You should see the same result as before, with the wall moving up and down in the arena. The difference is that, this time, it is running in its own thread. Now you can continue programming the rest of the game.



If you want to stop a multi-threaded program running in IDLE, click Shell→Restart Shell in the Python Shell. If you have been pressing Ctrl+C in the Python Shell, this stops only the main program, not the other threads, so you will need to click Shell→Restart Shell to stop the main program and any other threads.

DIGGING INTO THE CODE

To run the `theWall()` function in its own thread, you use the code `thread.start_new_thread(theWall, (arena, WALLZ))`. The first parameter is the name of the function—`theWall`—and the second parameter holds the variables the function expects to be passed—`arena, WALLZ`.

The variables (parameters) the function expects are passed in brackets (`arena, WALLZ`) because `start_new_thread` expects them to be passed as a **tuple**.

Any Python function can be called in this way. If you had written the following function to print a message to the screen:

```
def printMessage(message)
    print message
```

you could run it in its own thread, using:

```
thread.start_new_thread(printMessage,
    ("Hello Minecraft World",))
```

The comma after `"Hello Minecraft World"` tells Python that this is a tuple, but it's a tuple with only one item in it.



Tuples are similar to lists that you have used in previous adventures—the key difference is that once a tuple has been created it can't be changed, unlike a list where you can change it such as adding or removing items. In programming terms things which can't be changed are known as immutable, whereas those that can be changed are known as mutable. If you want to know more about Python tuples visit www.tutorialspoint.com/python/python_tuples.htm.

Building the River

Your next task is to build a river that runs the width of the arena. A river that's too wide for the player to jump! Luckily, there is a bridge over it. Not so luckily, the bridge moves backwards and forwards along the river, so the player needs to use careful timing to jump onto the bridge and off again on the other side (see Figure 9-7).

If the player falls into the river he will be taken back to the start of the arena. You will write the code to move the player back to the start in Part 3; for now, this section is about building the river and the moving bridge.



FIGURE 9-7 The player must cross the river.

First you need to create the river itself by clearing away part of the arena floor and putting a layer of water blocks at the bottom. You create the bridge by using `MinecraftShape` and make it move it from side to side using a similar method to the one you used to make the wall move up and down.

Update the `theRiver()` function to create the river obstacle by following these steps:

1. Find the `theRiver()` function by looking for the following code in the code you wrote earlier:

```
def theRiver(arenaPos, riverZPos):  
    pass
```

Delete the `pass` statement that is indented under the function.

2. Indented under the `def theRiver(arenaPos, riverZPos):` line, create a connection to Minecraft:

```
mc = minecraft.Minecraft.create()
```

3. Create two constants, which are the width of the river (`RIVERWIDTH`) and the width of the bridge (`BRIDGEWIDTH`):

```
RIVERWIDTH = 4  
BRIDGEWIDTH = 2
```

4. Now create the river across the arena position using the parameters passed in, `arenaPos` and `riverZPos` (the Z position along the arena where the river should be placed):

```
mc.setBlocks(arenaPos.x,  
            arenaPos.y - 2,  
            arenaPos.z + riverZPos,  
            arenaPos.x + ARENAX,  
            arenaPos.y,  
            arenaPos.z + riverZPos + RIVERWIDTH - 1,  
            block.AIR.id)  
  
mc.setBlocks(arenaPos.x,  
            arenaPos.y - 2,  
            arenaPos.z + riverZPos,  
            arenaPos.x + ARENAX,  
            arenaPos.y - 2,  
            arenaPos.z + riverZPos + RIVERWIDTH - 1,  
            block.WATER.id)
```

You create the river by using `setBlocks()` to create an area of `AIR` in the arena floor and a layer of `WATER` at the bottom.

5. Create the position where the bridge will be placed. You want it in the middle of the river, so that the player can't just step onto it but has to jump from the bank onto the bridge:

```
bridgePos = minecraft.Vec3(arenaPos.x, arenaPos.y,  
                           arenaPos.z + riverZPos + 1)
```

6. Create the shape blocks that will make up the bridge. This is done in a similar way to the wall by using two nested `for` loops, one for the bridge width (`x`) and one for the width of the river (`z`):

```
bridgeBlocks = []
for x in range(0, BRIDGEWIDTH):
    for z in range(0, RIVERWIDTH - 2):
        bridgeBlocks.append(minecraftstuff.ShapeBlock(x,
                                                       0,
                                                       z,
                                                       block.WOOD_PLANKS.id))
```

When creating the bridge, 2 is subtracted from `RIVERWIDTH` to create a gap of one block between the river bank and the bridge, meaning the player will have to jump on and off the bridge (see Figure 9-8).



FIGURE 9-8 Because the bridge does not completely span the river, your player has to jump.

7. Create the shape of the bridge by passing in the `bridgePos` and `bridgeBlocks` variables you created in steps 5 and 6:

```
bridgeShape = minecraftstuff.MinecraftShape(
    mc, bridgePos, bridgeBlocks)
```

- To move the bridge across the arena one block at a time, you need to calculate the number of steps the bridge would have to go through in order to get from one side to the other; this is the width of the arena minus the width of the bridge:

```
steps = ARENA_X - BRIDGE_WIDTH
```

- Move the bridge from side to side by using two `for` loops (one to go left and one to go right) and by using the `MinecraftShape.moveBy()` function to move the bridge one block to the side for each step, adding a small delay in between steps:

```
while not game_over:  
    for left in range(0, steps):  
        bridgeShape.moveBy(1, 0, 0)  
        time.sleep(1)  
    for right in range(0, steps):  
        bridgeShape.moveBy(-1, 0, 0)  
        time.sleep(1)
```

- Your river function is now complete. It needs to be called from the main program in a new thread, so add the following code to the bottom of the program:

```
RIVER_Z = 4  
thread.start_new_thread(theRiver, (arenaPos, RIVER_Z))
```

- Time to run the program! You should now see the arena with the wall in the middle, the river toward the start of the arena and the bridge going back and forth.



Enter the arena and try to get across the bridge. If you're good with the controls you might find this pretty easy—but wait until you are under pressure to collect the diamonds at the same time *and* do it as quickly as possible! Chances are you will miss the jump a lot more.

Creating the Holes

The final obstacles you need to create in Crafty Crossing are the holes. These are random holes in the arena floor that close up every few seconds and open again in different positions (see Figure 9-9).

You will use the `randint()` function to find random positions for the holes. `BLACK_WOOL` blocks will appear briefly in the floor before the holes open up, giving the player some warning and a chance to get out of the way. You create the holes by turning blocks in the arena floor to `AIR`. They stay that way for a few seconds before being turned back to `GRASS` and new holes are created elsewhere.



FIGURE 9-9 Holes appear, and your player must be careful not to fall into one!

As with the river, if the player falls into a hole they are returned to the start of the arena, but you won't introduce to the code to do that until Part 3 of this adventure.

Now, you're going to update the `theHoles` function to create the holes obstacle:

1. Find the `theHoles` function, by looking for the following code in the code you wrote earlier:

```
def theHoles(arenaPos, holesZPos):  
    pass
```

Delete the `pass` statement that is indented under the function.

2. Indented under the `def theHoles(arenaPos, holesZPos):` line, create a connection to Minecraft:

```
mc = minecraft.Minecraft.create()
```

3. Create two constants, which are the number of holes that will be created (`HOLES`) and the width of the holes obstacle (see Figure 9-10):

```
HOLES = 15  
HOLESWIDTH = 3
```

4. Create the holes `while` loop, which will continue until it's game over:

```
while not gameOver:
```

The rest of the code in the `theHoles` function will be indented under this `while` loop.

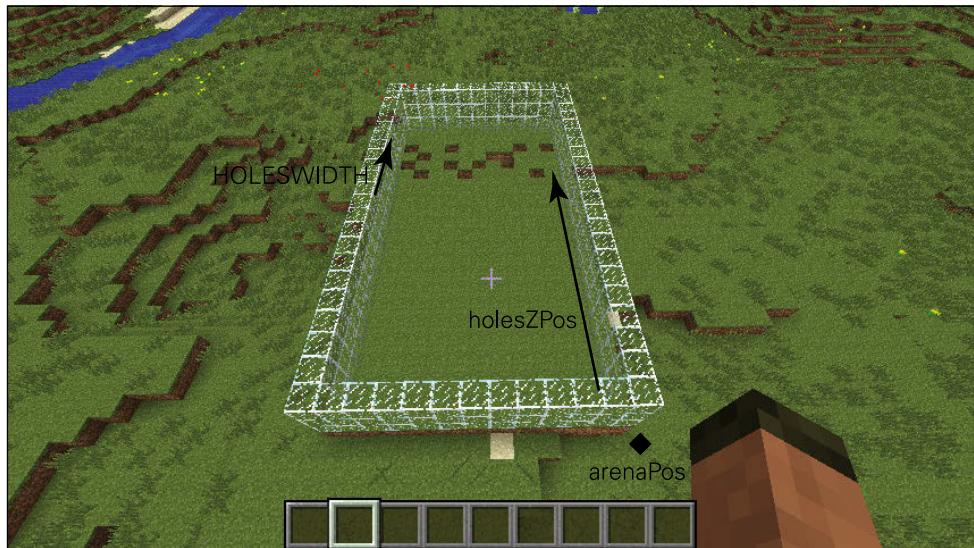


FIGURE 9-10 Create the position and width of the holes.

5. Create random positions for the holes by using the `random.randint()` function to create x and z coordinates, (the y position is the position of the arena) and append them to a Python list:

```
holes = []
for count in range(0,HOLES):
    x = random.randint(arenaPos.x,
                        arenaPos.x + ARENAX)
    z = random.randint(arenaPos.z + holesZPos,
                        arenaPos.z + holesZPos + HOLESWIDTH)
    holes.append(minecraft.Vec3(x, arenaPos.y, z))
```

6. Loop through all the positions in the holes list and turn the blocks to **BLACK WOOL**:

```
for hole in holes:
    mc.setBlock(hole.x, hole.y, hole.z,
                block.WOOL.id, 15)
    time.sleep(0.25)
```

By turning the holes in the arena floor black, you give the player a warning that a new hole is about to appear and they have the chance to get out of the way.

7. Open up the holes by setting the blocks below the hole position to be **AIR** using `setBlocks()`:

```
for hole in holes:  
    mc.setBlocks(hole.x, hole.y, hole.z,  
                 hole.x, hole.y - 2, hole.z,  
                 block.AIR.id)  
  
time.sleep(2)
```

Once the holes are created, a delay is put in so the holes stay there for two seconds.

8. Close up the holes by using the same loop you used to open them, but this time set the blocks back to **GRASS**:

```
for hole in holes:  
    mc.setBlocks(hole.x, hole.y, hole.z,  
                 hole.x, hole.y - 2, hole.z,  
                 block.GRASS.id)  
  
time.sleep(2)
```

The program will now return to the top of the **while** loop and recreate a new set of holes.

9. Your **holes** function is now complete. Add the following code to the bottom of the program:

```
HOLESZ = 15  
thread.start_new_thread(theHoles, (arenaPos, HOLESZ))
```

The **HOLESZ** constant is the z position down the arena where the holes obstacle will be created.

10. Run the program. As before, you should see the arena created with the wall and river obstacles, but now you should also see the holes continually opening and closing in random locations towards the end of the arena.

Try out the arena. See if you can get backward and forward over and through the obstacles without falling down or getting stuck.

You can adjust the constants in the game to make it your own. Perhaps make the arena longer or wider, put the obstacles in different positions or make it more difficult by making them faster or harder to get across.



DIGGING INTO THE CODE

The functions that create the obstacles are like mini-programs, and because they use multi-threading they all run independently. Because of this, if you wanted to have more than one type of obstacle in the arena, it's really easy to create a new one. Perhaps you'd like to have two walls? If you call the `theWall` function again but give it a different z position to the first wall, a second wall will appear and go up and down just like the first one:

```
WALL2Z = 13  
thread.start_new_thread(theWall, (arenaPos, WALL2Z))
```

CHALLENGE



You don't have to stop there. Using the same methods you used to create the wall, river and holes, can you create a new type of obstacle? Perhaps you can conjure up a cage that randomly appears and traps the player, or a series of platforms the player has to jump to reach the end of the arena.



If you find the obstacles too easy or too difficult you can change their difficulty by setting the constants to different values. For example, you can increase or decrease the delays to make the obstacles move slower or faster. To speed up the bridge, change the `time.sleep(1)` code in the move left and move right `for` loops to `time.sleep(0.5)`.

Part 3—Game Play

The next part of your big adventure is to add game play to your program. Your aim is to turn the arena from an obstacle course into a game where the player wants to play again and again, and get to the next level.

To achieve this, the game needs to be exciting and challenging, and include a reward and a goal.

The challenge will be for the player to collect all the diamonds that are randomly placed around the arena, while also trying to get through the obstacles in a set time limit. Points will be given to the player as a reward for collecting diamonds and for getting to the end of the level. The faster the player completes the level, the more points he will get. The goal is to complete all the levels and get as many points as possible.

Your game is going to have three levels, and you are going to make each level more difficult than the last by adding more diamonds and shortening the time limit.

Starting the Game

In this section you will set up the game, and create constants that establish how many diamonds and the amount of time available in each level.

The program has two main loops:

- **The game loop:** This loop will continue until the game is over (`while not gameOver`). This is where each level is set up and started. Points are calculated at the end of each level.
- **The level loop:** This loop will continue until the end of each level—so, either when the level is complete or the game is over (`while not gameOver and not levelComplete`). This loop also returns a player to the start if he falls into the river or a hole, clears diamonds if the player hits them, and checks to see if the time has run out.

The instructions for this part of the adventure refer to indenting code under either the `game` loop or the `level` loop. It is important that you put the code in the right place; otherwise the game won't work properly.



First, create the structure of the game and create the two main loops by following these steps to add the necessary code to the bottom of the program:

1. Create three constants for the number of levels in the game, the number of diamonds that will be created and the amount of time (in seconds) the player will be given to complete each level:

```
LEVELS = 3  
DIAMONDS = [3, 5, 9]  
TIMEOUTS = [30, 25, 20]
```

The `DIAMONDS` and `TIMEOUTS` constants are Python lists. Both have three items, which are the values for each level; for example, in the first level, the player will have to collect three diamonds and will have 30 seconds to do it and get to the other side.

2. Create two variables to hold the points the player has scored and the level they are currently on:

```
level = 0  
points = 0
```

3. Create the `game` loop. Place a comment over it to remind you where it is, as further instructions in this adventure will refer to indenting code under the `game` loop:

```
#game loop  
while not gameOver:
```

The variable `gameOver` is set to `False` at the start of the program; it will be set to `True` when the player runs out of time or completes the game. This variable is also used in the obstacles functions, and when it is set to `True` it will result in all the obstacles stopping.

4. Indented under the `game` loop, change the position of the player so that he is at the start of the arena and ready to begin:

```
mc.player.setPos(arenaPos.x + 1, arenaPos.y + 1,  
                 arenaPos.z + 1)
```

5. Start the clock for the level by getting the current time and putting it into a variable:

```
start = time.time()
```

6. Set the level complete flag to `False` and create the level loop. As you did with the `game` loop, create a comment here to remind you where the `level` loop is, as further instructions will refer to indenting code under the `level` loop:

```
levelComplete = False  
#level loop  
while not gameOver and not levelComplete:
```

7. Indented under the `level` loop, put in a small delay. You need this because the program will loop all the time while the game is playing and, without it, the program would use all the computer's processing power:

```
time.sleep(0.1)
```

8. Run the program. The only change you will see is that the player is automatically put at the start of the arena, but it will give you the chance to check that everything is working properly and no errors have occurred.

CHALLENGE

When the player's is put at the start of the arena, he starts the level in the right hand corner. Would it be better if he started in the middle? Change the code so that the player is put in the middle of the arena rather than the corner at the start of each level.



Collecting Diamonds

The main objective of the game is to collect diamonds. You are now going to program these to appear in random positions in the arena (see Figure 9-11); the player will collect the diamonds by “hitting” them (or rather, right-clicking the block while holding a sword).



FIGURE 9-11 Diamonds appear in random positions.

The diamonds disappear as soon as they have been hit, and, once he has collected all the diamonds, the player can go on to complete the level by getting to the other side of the arena.

Update the `createDiamonds()` function and call it from the `game` loop by following these steps:

1. Find `createDiamonds()` function by looking for the following in the code you wrote earlier:

```
def createDiamonds(arenaPos, number):  
    pass
```

Delete the `pass` statement that is indented under the function.

2. Indented under the `def createDiamonds(arenaPos, number):` line, create a connection to Minecraft:

```
mc = minecraft.Minecraft.create()
```

3. Create the number of diamonds you require by finding random x and z positions in the arena and setting the block to a `DIAMOND_BLOCK`:

```
for diamond in range(0, number):
    x = random.randint(arenaPos.x, arenaPos.x + ARENAX)
    z = random.randint(arenaPos.z, arenaPos.z + ARENAZ)
    mc.setBlock(x, arenaPos.y + 1, z,
                block.DIAMOND_BLOCK.id)
```

4. The `createDiamonds()` function now needs to be called at the start of the `game` loop; every time a new level starts, a new set of diamonds is created. Indented under the `game` loop, directly under the `while` loop, add the following code:

```
#game loop
while not gameOver:
    createDiamonds(arenaPos, DIAMONDS[level])
    diamondsLeft = DIAMONDS[level]
```

The variable `diamondsLeft` is also created; this will hold the number of diamonds remaining for the player to collect.

5. Run the program. Because you are on the first level, you should see three diamonds created at random locations in the arena.

After you have created the diamonds, you can add the code to monitor the player's hit events using the `pollBlockHits` function (which you learned in Adventure 4) and, when if the player hits a `DIAMOND` block, turn it to `AIR`.

Add the following code under the `level` loop to turn the `DIAMOND_BLOCK` to `AIR` if the player hits it:

1. Indented under the `level` loop, call the `pollBlockHits` function to get any block hit events:

```
#level loop
while not gameOver and not levelComplete:
    hits = mc.events.pollBlockHits()
```

2. Loop through the block hit events and get the type of block that was hit:

```
for hit in hits:
    blockHitType = mc.getBlock(hit.pos.x, hit.pos.y,
                               hit.pos.z)
```

- Check to see if the type of block hit was `DIAMOND_BLOCK`. If it was, turn it to `AIR` and subtract 1 from the `diamondsLeft` variable:

```
if blockHitType == block.DIAMOND_BLOCK.id:  
    mc.setBlock(hit.pos.x,hit.pos.y, hit.pos.z,  
                block.AIR.id)  
    diamondsLeft = diamondsLeft - 1
```

The `diamondsLeft` variable will be used to check that all the diamonds have been collected when the player gets to the end of the arena.

- Run the program; when you hit the `DIAMOND` blocks, they should turn to `AIR` and disappear.

Remember that, in order to hit a block, you must be holding a sword and using the right-click button on the mouse.



CHALLENGE

Can you make the diamonds harder to hit? Perhaps you can make them rise up and down and allow the player to hit them only when they are in the air.



Out of Time

If your player was given an infinite amount of time, the game would be really easy (and boring). By introducing a time limit, you make the game challenging; and by making the player get all the diamonds and get to the other side before the time runs out, you give him a goal.

If the time runs out, it's game over. Your next task is to add the code to the level loop that will check to see if the time has run out and, if it has, will set the `gameOver` flag to `True`:

- Indented under the `level` loop, tell the program to calculate the number of seconds left in this level:

```
#level loop  
while not gameOver and not levelComplete:  
  
    secondsLeft = TIMEOUTS[level] - (time.time() - start)
```

2. If there are less than zero seconds left, set the `gameOver` flag to `True` and post a message to the chat:

```
if secondsLeft < 0:  
    gameOver = True  
    mc.postToChat("Out of time...")
```

3. Run the program. After 30 seconds (as this is level 1), the program should end and the message “Out of time...” should appear (see Figure 9-12).



FIGURE 9-12 Let your players know when they run out of time.

DIGGING INTO THE CODE

The number of seconds that are left for the level is calculated using the following code:

```
secondsLeft = TIMEOUTS[level] - (time.time() - start)
```

The variable `secondsLeft` is set by taking the number of seconds that are allowed for this level from the `TIMEOUTS` constant:

```
TIMEOUTS[level]
```

Then you subtract how many seconds the player has been playing the level. This is the time now, minus the time when the level started:

```
(time.time() - start)
```

Tracking the Player

When the player has collected all the diamonds and reached the end of the arena, he has completed the level. If he falls into the river or a hole, he is returned to the start of the arena. To make these things happen, your program needs to know where the player is.

After checking where the player is, the program can then either set the `levelComplete` flag to `True` if he has collected all the diamonds or set his position back at the start of the arena.

Your next task is to add the code to the `level` loop to track the player and either put him back to the start or complete the level:

1. Indented under the `level` loop, get the player's position:

```
#level loop
while not gameOver and not levelComplete:

    pos = mc.player.getTilePos()
```

2. Check to see if the player's height, `y`, is lower than the height of the arena. If it is, he must have fallen into the river or a hole, so put him back to the start:

```
if pos.y < arenaPos.y:
    mc.player.setPos(arenaPos.x + 1, arenaPos.y + 1,
                     arenaPos.z + 1)
```

3. Check to see if the player has reached the end of the arena and has collected all the diamonds. This is done by seeing whether the player's `z` position is the same as the end of the arena and, if it is, setting the `levelComplete` flag to `True`:

```
if pos.z == arenaPos.z + ARENAZ and diamondsLeft == 0:
    levelComplete = True
```

When the `levelComplete` flag is set to `True`, the `level` loop ends, the diamonds are re-created and the game starts again.

4. Run the program. The game should reset when all the diamonds have been collected and the player reaches the end of the area; and, if the player falls into the river or a hole, he should be returned to the start of the arena.

At the moment the game is playing only one level—the first and easiest one. It might be wise for you to practice your moves now, because in the next section the difficulty is going to be cranked up!



Setting the Level as Complete and Calculating Points

When the player completes a level, the program needs to calculate how many points he has scored and put the game on to the next level. The player scores points if he completes the level. He receives one point for every diamond he has collected, multiplied by the number of seconds left on the clock.

To do that, you need to include the following code. The new code needs to be indented under the `game` loop, but it has to come after the `level` loop has finished:

1. Indented under the `game` loop, but after the `level` loop, check to see if the level was completed:

```
#game loop
while not gameOver:
    [code]

    #level loop
    while not gameOver and not levelComplete:
        [code]

        if levelComplete:
```

2. If the level was completed, calculate the points scored and add them to the `points` variable, before posting the results to the chat:

```
    points = points + (DIAMONDS[level] * int(secondsLeft))
    mc.postToChat("Level Complete - Points = " + ↵
                  str(points))
```

3. Set the game to the next level by adding 1 to the `level` variable:

```
    level = level + 1
```

4. If this is the last level, set the `gameOver` flag to `True` and post a message of congratulations to the chat:

```
    if level == LEVELS:
        gameOver = True
        mc.postToChat("Congratulations - All levels ↵
                      complete")
```

The `LEVELS` constant holds the total number of levels in the game.

5. Run the program.

Your game is very nearly complete! When the player has collected all the diamonds and battled his way to the end of the arena before the time runs out, the game will restart on the next level. Here it will get a little bit harder, with more diamonds and less time. If the player manages to complete all the levels, it's game over, and congratulations!

CHALLENGE

Try adding more levels. You could make the game begin at a more leisurely pace by giving the first few levels more time and fewer diamonds, increasing the difficulty gradually to make the later levels really hard.



Adding the Game Over Message

The very last thing you need to do to complete the game play is to add a message right at the end of the program, telling the player it's game over and giving him his final score (see Figure 9-13). Simply add the following code to the very bottom of the program:

```
mc.postToChat ("Game Over - Points = " + str(points))
```



FIGURE 9-13 Tell the player when the game is over and then display the score.



All levels complete—156 points. Beat that!

CHALLENGE



Write the scores to a file on your computer and create a league table—at the end of the game, display the player's position.

You can download the complete Crafty Crossing program on the companion website at www.wiley.com/go/adventuresinminecraft.

It doesn't really have to be game over! The program is made so you can extend it, play around with the settings, introduce new obstacles or create a magnificent arena. It's up to you.

Part 4—Adding a Button and Display

There are a couple of problems with the Crafty Crossing program. The game starts automatically, whether the player is ready or not, and there is no display of useful information such as how many diamonds are left to collect and whether the time is running out.

In the last part of this adventure, you are going to re-use the hardware you created for the detonator in Adventure 5 to add a button for you to press when you want the game to start. You will also use the 7-segment display to show how many diamonds are left; the decimal point on the display will light up when there are only five seconds left to complete the level.

What You Will Need

You will need the same electronic components you used in Adventure 5, connected the same way you connected them in the final exercise, when you made a detonator. You should have a breadboard set up with a 7-segment display and button, connected to either your Raspberry Pi (see Figure 9-14) or to an Arduino (see Figure 9-15). If you need to, look back at Adventure 5 for the set-up.

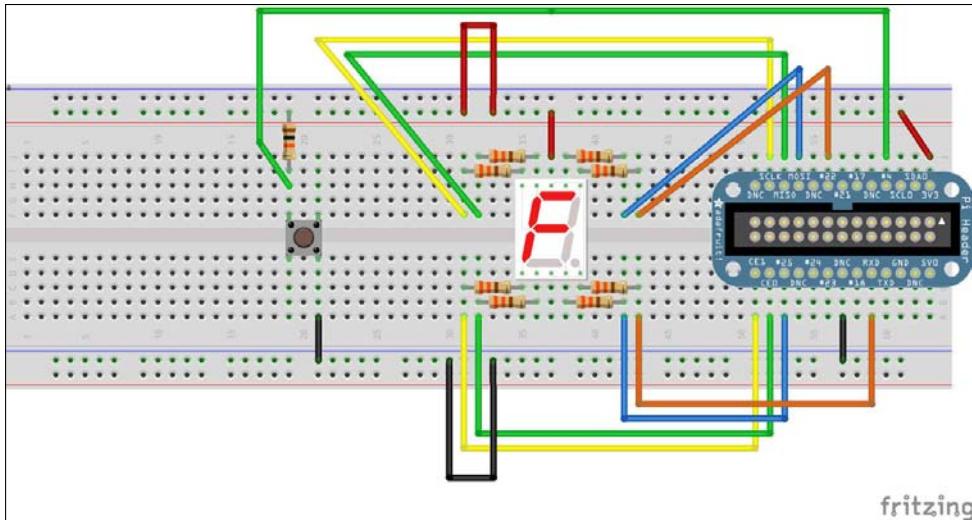


FIGURE 9-14 Circuit diagram for the Raspberry Pi

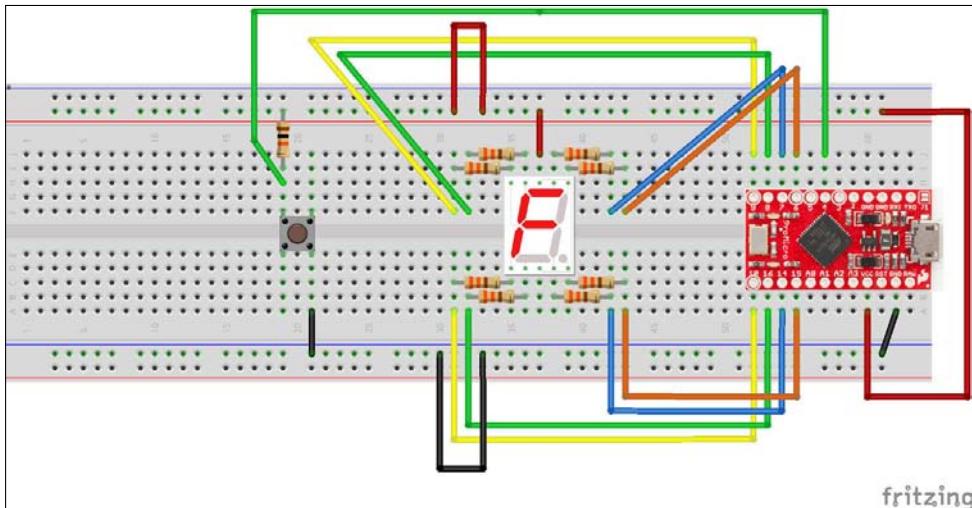


FIGURE 9-15 Circuit diagram for the Arduino

Set Up the Hardware

To include a button and a 7-segment display, the first step is to import the right modules, set up the GPIO and create constants for the pins.

Modify your Crafty Crossing program to set up the hardware and wait for the button to be pressed before starting the game:

1. Import the 7-segment display module under the existing `import` statements:

```
import anyio.seg7 as display
```

2. Import the GPIO module for your hardware and create constants for the pins that will be used:

On the Raspberry Pi:

```
import RPi.GPIO as GPIO
BUTTON = 4
LED_PINS = [10,22,25,8,7,9,11,15]
```

On the Arduino:

```
import anyio.GPIO as GPIO
BUTTON = 4
LED_PINS = [7,6,14,16,10,8,9,15]
```

3. Set up the hardware:

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUTTON, GPIO.IN)
ON = False # common-anode, set to True for common cathode
display.setup(GPIO, LED_PINS, ON)
```

If you are using a common cathode 7-segment display (as described in Adventure 5), use `ON = True`.

4. Just before the `game` loop, add the code to wait for the button to be pressed:

```
mc.postToChat("Press the button to start")
while GPIO.input(BUTTON) == 0:
    time.sleep(0.1)
while not gameOver:
```

The program will now create the arena and the obstacles, then wait for the player to press the button. Only when that happens will the program exit the loop, put the player in the arena and start the game.

5. It is important to finish off your program correctly and clean up the GPIO. Add the following code at the end of the program:

```
GPIO.cleanup()
```

6. Run the program and test the new start button.

CHALLENGE

Can you change the program so that you can press the button at the end of the game to start a new game rather than having to re-run the program?



Diamond Countdown

The 7-segment display will show the number of diamonds left to collect; the display will be updated using the `display.write` function.

To do this, you need to modify the Crafty Crossing program to update the display whenever the diamonds are created in the arena and whenever a player hits a diamond:

1. After the `createDiamonds` function has been called in the `game` loop and the diamonds have been put in the arena, update the display:

```
createDiamonds(arenaPos, DIAMONDS[level])
diamondsLeft = DIAMONDS[level]
display.write(str(diamondsLeft))
```

The `display.write()` method expects a string to be passed to it, so the number of diamonds left is converted to a string using `str()` before being passed to the function.

2. When a player has hit a diamond and the number of diamonds is reduced by 1, update the display:

```
hits = mc.events.pollBlockHits()
for hit in hits:
    blockHitType = mc.getBlock(hit.pos.x, hit.pos.y,
                               hit.pos.z)
    if blockHitType == block.DIAMOND_BLOCK.id:
        mc.setBlock(hit.pos.x, hit.pos.y, hit.pos.z,
                    block.AIR.id)
        diamondsLeft = diamondsLeft - 1
        display.write(str(diamondsLeft))
```

3. At the end of the program, you should make sure the display is cleared. So, before the GPIO is cleaned up, add the code to clear the display:

```
display.clear()
GPIO.cleanup()
```

- Run the program. The number of diamonds should be displayed and should decrease each time the player hits one, until it reaches 0.



The 7-segment display can only display up to the number 9. If you have changed the program so that a level has more than 9 diamonds in it, you should add an `if` statement to the program so that it updates only the display if there are 9 diamonds or less left to collect.

Time-Left Indicator

Your very last task in this adventure is to update the display so the player knows that time is running out. To do this, you will turn on the decimal point on the display. The decimal point LED should come on when there are less than five seconds to go, giving the player a last chance to try and complete the level.

Update the Crafty Crossing program to turn on the decimal point LED in the last five seconds of the level:

- After the number of seconds left is calculated in the `level` loop, check to see if there are less than five seconds left. If there are, turn on the decimal point; otherwise, turn it off:

```
secondsLeft = TIMEOUTS[level] - (time.time() - start)
if secondsLeft < 5:
    display.setdp(True)
else:
    display.setdp(False)
```

- Run the program. When there are only five seconds left, the decimal point will light up.

CHALLENGE



When a new level starts, use the display to flash up the level number before showing the number of diamonds to be collected.

Quick Reference Table

Command	Description
<code>import thread</code>	Imports the Python thread module
<code>thread.start_new_thread(function, (variable1, variable2))</code>	Calls a function in its own thread
<code>import time</code>	Imports the Python time module
<code>timeNow = time.time()</code>	Gets the current time

Further Adventures in Your Continuing Journey with Minecraft

Minecraft gives you a fantastic canvas for creativity and adventure. Add to this the power of being able to control the game through code and the only limitation is your imagination. What will you do next?

Here are some ideas and resources that will hopefully give you inspiration:

- Creating games is a great way to stretch your programming skills. Check out www.classicgamesarcade.com for some ideas.
- Minecraft is multiplayer game. Why not take advantage of that and create programs which many people can use and enjoy?
- Interacting with electronics brings Minecraft into the real world. Take your skills further with Adventures in Arduino eu.wiley.com/WileyCDA/WileyTitle/productCd-1118948475.html.
- There are many open data sources on the Internet. How about integrating Minecraft with websites such as twitter (dev.twitter.com) or weather forecasts from the Met Office (www.metoffice.gov.uk/datapoint)?



Achievement Unlocked: Your big Minecraft project!



Appendix A

Where to Go from Here

WE HOPE THE adventures in this book have given you a whole range of ideas, code snippets and skills, and inspired you to take your Minecraft programming adventures further. Where you go from here is up to you and your imagination! If you’re not sure what you want to do yet, or have an idea for your next project but don’t know where to start, here are some interesting resources you can investigate to get those ideas flowing!

Websites

The Internet has a wealth of useful websites with information about Minecraft, almost too much to sort through, but here are some sites that Martin and David have found useful when learning, playing and programming Minecraft:

Minecraft

- www.wiley.com/go/adventuresinminecraft—This is the companion website for this book. It includes a complete bonus adventure (the Minecraft Lift), downloadable quick-reference sheets, badges, complete program listings, and videos for each of the projects in this book.
- www.stuffaboutcode.com—This is Martin’s very successful blog. It has a whole section about Minecraft with lots of project ideas and experiments that you can try out yourself. Martin also hosts the best Minecraft API reference that you will find anywhere on the Internet!

- <http://arghbox.wordpress.com>—Craig Richardson regularly develops exciting Minecraft programming projects to help support the teaching of the new computing curriculum. Be sure to have a look at his open source Minecraft programming book and his Minecraft controller built from real fruit!
- [www.minecraftforum.net](http://minecraftforum.net)—This is the official Minecraft Forum where you can get help on any topic related to Minecraft, such as new features, setting up servers, and tips for creative and survival mode.
- http://minecraft.gamepedia.com/Minecraft_Wiki—The Minecraft Wiki is a community managed collection of information about Minecraft. And because it is community managed, this means you can contribute to it too!
- [www.scarabcoder.com](http://scarabcoder.com)—The blog of a young coder, Nicholas Harris, who started programming by using Minecraft on the Raspberry Pi. He posts updates on his adventures in coding and technology. Be sure to look at his series on Bukkit plugins.
- www.reddit.com/r/MCPI—Reddit is a social linking website where people post links and then vote them up or down. A wide range of interesting Minecraft programming projects regularly pop up on this Reddit page.
- <http://mcipy.wordpress.com> and <https://github.com/brookscc/mcipy>—This is a collection of Minecraft programming resources. You will find these resources in other places too, but many people (including David) discovered Martin O'Hanlon through this website and the Reddit pages.
- www.minecraftmaps.com—Minecraft allows you to load and save adventure maps, which are complete snapshots of a Minecraft world. This site has a huge repository of community-developed maps that you can load into your Minecraft world, then build on top of them.
- <http://minecraft.curseforge.com>—This is an open source exchange hosting site (a bit like sourceforge, but for Minecraft), where projects are hosted and can then be downloaded from www.curse.com.
- <http://minecraft-seeds.net>—Minecraft worlds are generated by a computer algorithm built inside Minecraft. Random numbers are “seeded” from a start number. If you enter a known seed, you will recreate that world. This site lists seeds you can type in to create specific Minecraft worlds.
- [www.mcedit.net](http://mcedit.net) and www.worldpainter.net—These are free downloadable tools that you can use to design and paint your own custom world maps using onscreen editing tools.
- <http://mcreator.pylo.si>—MCreator is a mod maker for Minecraft, using a simple point-and-click interface with no programming. With it, you can create fantastic Minecraft mods such as new block types, mobs, armour, commands and many other items.

- www.firetechcamp.com—Fire Tech Camp run coding camps for teenagers during every school holiday all around the UK. They have some great courses teaching video games design, Minecraft construction, Python programming, Arduino and many other tech courses tailored for teens interested in tech.

Python

- <https://www.python.org> and <https://docs.python.org/2>—These are the official download and documentation pages for the Python programming language.
- www.codecademy.com/tracks/python—This is a free online course. You can follow the lessons yourself to gradually learn the Python programming language.
- <http://inventwithpython.com/chapters>—This is a free online book with many great Python projects to teach you how to program in Python.
- <https://docs.python.org/2/library/idle.html>—This is the official guide for using the IDLE integrated development environment. But as a guide, it's quite precisely written and there are no pictures. We prefer to use these tutorials from Dr Anne Dawson: www.annedawson.net/Python_Editor_IDLE.htm
- www.geany.org—IDLE is a very small and simple programming environment. After a while, you will probably want something a bit more full-featured. Try Geany instead, it's a much nicer environment to use.
- <http://blog.whaleygeek.co.uk>—This is David's blog. It has a number of Python and Raspberry Pi projects, hints and tips, and some downloadable flashcards to remind you of the important syntax of Python and Minecraft. Print these out and keep them in your top pocket for when you need a reminder!

Bukkit

- http://wiki.bukkit.org/Plugin_Tutorial—The Bukkit server that you used in this book is a community developed server, and it supports plugins. The plugins are written in Java. You can download plugins for almost anything, and you can even write your own plugins! Learn how on this website.
- <http://dev.bukkit.org/bukkit-plugins/raspberryjuice>—The RaspberryJuice that you use in this book is actually a Bukkit plugin, which exposes the inner Minecraft API through the MCPI Python interface used on the Raspberry Pi.
- <http://www.skpang.co.uk>—Mr S.K.Pang is a good stockist in the UK for electronic components and interesting projects. He also stocks a pre-assembled and pre-programmed version of the Sparkfun Arduino Pro-Micro, and a pre-assembled Adafruit T-Cobbler with a helpful sticker that labels all the GPIO pins for you. You might consider buying these pre-assembled so you can get going quicker in Adventures 5, 9 and the Bonus adventure.

- <http://www.sparkfun.com>—Sparkfun have a really good range of electronics and some really well written hookup guides for getting everything connected. Sparkfun designed the Arduino Pro-Micro that you may have used in Adventures 5, 9 and the bonus adventure, but note that many of their products are not pre-programmed or pre-soldered, so you'll need to do a little more work to get going from here.
- <http://www.maplin.co.uk>—Maplin Electronics are a good high-street retailer, and it's great to be able to have an idea on a Sunday afternoon and just walk into a shop and buy the necessary components while you still have the idea fresh in your mind!

Other Ways to Make Things Happen Automatically

Programming Minecraft in Python is not the only way to automate tasks. Minecraft has redstone and command blocks, as well as tags. Each of these can be combined with each other to make parts of your Minecraft world do different things automatically.

- http://minecraft.gamepedia.com/Tutorials/Command_Block—Command blocks can be triggered to automate tasks for you. You can wire them up to a redstone signal that triggers the command to run. There are many things you can build; just take a look at the booby traps and teleporters on this tutorial page.
- www.minecraftforum.net/topic/1969520-17-using-summon-give-datatags-in-map-making-tutorials—This site shows how to use Named Binary Tags (NBT—the internal save format of many Minecraft data types) with commands. Without programming, you can send in commands with tags to create and configure a range of different Minecraft objects, such as blocks and entities.
- www.minecraft101.net/redstone/redstone101.html—Redstone is the Minecraft equivalent of electricity. It is deceptively simple, but out of lots of very simple circuits you can build very complex devices.

Projects and Tutorials

Sometimes it's fun to come up with your own project ideas, but sometimes you need that little extra inspiration to get going. The websites listed here have a very nice collection of example projects, many of which are contributed by hobbyists and developers in the community.

- http://minecraft.gamepedia.com/Tutorials/Advanced_redstone_circuits—This tutorial shows how redstone can be used to piece together lots and lots of small redstone circuits to make a really large automated structure, and walks you through how to build a complete computer out of redstone, step by step!

- <https://learn.adafruit.com/search?q=minecraft>—Adafruit have a really nice collection of tutorials with instructions written by their staff and other people in the community. They have a relatively new area for Minecraft projects that link to electronics, and it looks like they are adding more to this.
- www.stuffaboutcode.com/2013/09/minecraft-ordnance-survey-map-rastrack.html—This is the first project that Martin and David worked on together, and we hadn't even met in person at this point! It's a "mashup" (a collection of programs mashed together). We took data from Ryan Walmsley's Rastrack website (www.rastrack.co.uk) and the Ordnance Survey open maps data, and created the whole of the UK full of Raspberry Pi in the sky! See the video on the website to learn more.
- <http://hackaday.com/2013/01/30/controlling-minecraft-with-a-raspberry-pi>—Here's another way to control real hardware from Minecraft, using Bukkit with redstone. A Bukkit plugin communicates with the Raspberry Pi, making levers and signs inside Minecraft control and monitor real world electronics.
- www.instructables.com/howto/minecraft—The instructables website has a huge collection of really well-written step-by-step instructions on how to build projects, and they now have a Minecraft project area, too.

Videos

Minecraft like many modern games is very visual, and one of the best ways to get ideas and inspiration is to watch what others are doing. Many of these videos are from bloggers and hobbyists that have become famous through their videos alone, and many of them post videos weekly or even daily.

- <http://minecraft.gamepedia.com/Tutorials/Elevators>—In the bonus adventure on the website you can program a fully functional Minecraft lift. There are many different ways that you can build a lift in Minecraft and they are all well documented on this Wiki, along with some great videos of the lifts working.
- www.youtube.com/user/sethbling—Seth Bling has a huge and very active YouTube channel with lots of Minecraft projects and ideas. Be sure to check out his TNT Olympics!
- www.youtube.com/user/stampylonghead—Stampy releases at least one new Minecraft video per day on his YouTube channel and has a lot of fun building awesome structures.
- www.youtube.com/user/scarabcoder—Nicholas Harris has a good YouTube channel with lots of interesting projects. He has also written an e-book, available on Amazon, about how to write Minecraft programs.
- www.youtube.com/user/SimplySarc—Take a look at SimplySarc, especially his well described Vanilla Camera (www.youtube.com/watch?v=NyMHCabq_rs) that uses command blocks to make a camera that takes a photo with no mods at all!

- www.youtube.com/watch?v=7t4bH7Z-Yt4—Our very own Martin O'Hanlon shows a Python program that turns any block you hit into a bomb that goes off after a few seconds.
- www.youtube.com/user/ThatMumboJumbo—MumboJumbo has a regular YouTube series about building with redstone, pistons and levers. Take a look at his version of a Minecraft lift made from pistons: www.youtube.com/watch?v=jQulvbvtYI
- www.youtube.com/user/HiFolksImAdam—Adam creates some interesting visual illusions with command blocks, and builds a TARDIS (www.youtube.com/watch?v=3QpqUCaz8fk) that really is bigger on the inside than it is on the outside!
- www.youtube.com/user/AsdjkeAndBro—Asdjke and Bro have built a number of mini-games inside Minecraft. One of our personal favourites is the battleship game: www.youtube.com/watch?v=6AbPlT-cAm8

Books

There's still something nice about having a printed book by your side, propped open with personalised sticky notes or pencil marks or turned over page corners, when working through your Minecraft adventures. Both Martin and David have learnt a lot from these other authors as well, and think that you'll find many additional projects and ideas that will complement your learning and fun gained from this book. In particular, Becky Stewart's book "Adventures in Arduino" is a really nice follow on if you are using a PC/Mac and enjoyed the electronic circuits you built in Adventures 5, 9 and the Bonus Adventure. The Arduino Pro Micro that we used is compatible with most if not all of the projects in Becky's book, and you can follow the instructions on the Sparkfun website to learn how to re-program the firmware inside the Arduino Pro Micro to make it do anything you want.

- ***Minecraft for Dummies*** by Jacob Corderio (Wiley, 2013)
- ***Minecraft: The Official Beginners' Handbook*** (Egmont, 2013)
- ***Minecraft: The Official Redstone Handbook*** (Egmont, 2013)
- ***Minecraft: Awesome Building Ideas for You*** (Createspace, 2014)
- ***Learn to Program with Minecraft Plugins*** by Andy Hunt (Pragmatic Bookshelf, 2014)
- ***Adventures in Raspberry Pi*** by Carrie Anne Philbin (Wiley, 2013)
- ***Adventures in Arduino*** by Becky Stewart (Wiley, 2014)
- ***Adventures in Python*** by Craig Richardson (Wiley, 2014)
- ***Python for Kids*** by Jason R. Briggs (No Starch Press, 2012)

Index

SYMBOLS AND NUMERICS

: (colon), 52, 69
== (double equals) symbol, 50
(hash symbol), 61
+ (plus symbol), 42
() (round brackets), 98
= (single equals) symbol, 50
[] (square brackets), 98
7-segment display
 about, 138–140
 controlling with Python module,
 144–145
 functions, 152
 using, 152
wiring up, 140–142
writing characters to, 152
writing Python to drive, 142–144

A

A (amperes), 124
a variable, 61, 69, 70
abs() function, 112
absolute coordinates, 66
accessing
 items in lists by index, 115
 last item on list, 115
 Minecraft API, 40
Adafruit
 Adafruit Pi-T-Cobbler, 120
 tutorial, 277
adding
 bridge builder, 113–115
 buttons, 266–270
 carpets, 85–87
 displays, 266–270

to end of list, 115
Game Over message, 265–266
homing beacons, 112–113
adjusting
 blocks, 88
 constants, 255–256
 positions of blocks, 88
administrator account, 21
Adventures in Arduino (Stewart), 278
Adventures in Python (Richardson), 278
Adventures in Raspberry Pi (Philbin), 278
AIR block type
 block friends, 216, 220
 Crafty Crossing game, 242, 252,
 255–256, 260, 261
 creating bigger shapes, 225
 creating blocks, 64, 67, 75, 77, 88
 creating Minecraft clock, 200,
 204–205
 data files, 186
 drawing spheres, 200
 interacting with blocks, 107
alien invasion, 228–234
AlienInvasion.py program,
 228–234
ALIEN_TAUNTS, 228, 234
amperes (A), 124
and keyword, 47
anotherGo variable, 179
anyio package, 129, 138, 144, 146
API (application programming
 interface), 40
append() function, 101, 209
Apple Mac
 Arduino and, 121
 building magic bridges, 96

Apple Mac (*continued*)
configuring drivers, 126
displaying coordinates, 37
installing Python on, 22–24
installing starter kit on, 22–24
requirements for electronic circuits
 adventure, 120–121
 setting up to control electronic
 circuits, 125–128
 starting Minecraft on, 24–27
 website, 9
Apple password, 23
application programming interface
(API), 40
Arduino
 about, 120–121
 flashing LEDs, 133–134
 lighting up LEDs from computers,
 131–132
 writing Python to drive 7-segment
 displays, 143–144
Arduino Pro Micro, 121
arena, building, 239–243
Asdjke and Bro (website), 278
automating tasks, 276

B

baskslash character, 185
BLACK WOOL block type, 252, 254–256
blank lines, 165
Bling, Seth (YouTube user), 277
block module
 alien invasion, 228–234
 block friends, 217
 building 2D/3D structures, 194, 195
 creating Minecraft clock, 201–205
 creating pyramids, 209–213
 drawing polygons, 207–209
 interacting with blocks, 93
blockBetween, 220
BlockFriend.py program,
 217–221, 223
BlockFriendRandom.py
program, 223

blockHit.py program, 105–108,
 111–112
blocks. *See also specific types*
about, 91, 115, 215, 235
alien invasion, 228–234
building magic bridges, 95–97
building more than one, 66–67
changing, 88
changing position of, 88
checking what players are standing
 on, 92–97
checking which are hit, 115
command, 276
creating, 64–65
creating bigger shapes, 225–228
as friends, 215–221
getting type at positions, 115
id numbers, 89
interacting with, 91–116
name constants, 88
random numbers and, 222–225
sensing hits on, 105–108
setting, 88
setting position of, 88
types of, 88
using Python lists, 98–104
writing treasure hunt game, 108–115
blocksBetween, 220
books, as resources, 278
boolean variables, 181, 182
brackets, 98
breadboard, 121–125
break statement, 182
Bresenham line algorithm (website), 197
bridge builder, adding, 113–115
bridgeBlocks variable, 251
bridgePOS variable, 251
Briggs, Jason R. (author)
 Python for Kids, 278
buildBridge() function
 building magic bridges, 95
 building vanishing bridges, 102
calling code with, 97
treasure hunt game, 113

`buildHouse2.py` program, 81
`buildHouse.py` program, 78, 80–81
building. *See also* creating
about, 63, 89
adding carpets, 85–89
arena, 239–243
automatically, 63–90
circuits that light LEDs, 123–124
clearing space, 71–74
creating blocks, 64–65
duplicating machine, 178–191
duplicator room, 183–185
houses, 74–85
lifts, 277
with `for` loops, 67–71
magic bridges, 95–97
magic doormats, 48–49
mazes from data files, 160–168
more than one block, 66–67
multiple houses, 79–85
river, 249–252
with `setBlocks()`, 71–72
3D block printer, 168–174
3D block scanner, 174–177
3D structures. *See* 3D structures
2D structures. *See* 2D structures
vanishing bridges with Python lists,
101–104
Welcome Home game, 45–52
`build.py` program, 66–67
`buildRoom()` function, 183–185
`buildStreet2.py` program, 87, 89
`buildStreet.py` program, 84
Bukkit server
connecting Minecraft to, 26–27
defined, 19
resources, 275–276
stopping, 27–28
tutorials on, 277
website, 19, 275
buttons
adding, 266–270
wiring, 146–148

call parameters, 196
calling, 97
carpets, adding, 85–87
case-sensitivity, in Python,
30, 39
Challenge sidebars, 10
changing
blocks, 88
constants, 255–256
positions of blocks, 88
“Channel already in use” message, 135
charging rent, 53–58
`checkHit()` function
sensing hits on blocks, 105–108
treasure hunt game, 111
checking player location, 47–48
circles, drawing, 197–199
cleaning
after using GPIO, 152
duplicator room, 187
`cleanRoom()` function, 187
`clear()` function, 227–228
clearing space, 71–74
`clearSpace.py` program, 72–74, 75,
185–186
`clone()` function, 218
COBBLESTONE block type, 88
Code sidebars, 10
codeacademy (website), 275
collecting
diamonds, 259–261
treasure, 111–112
colon (:), 52, 69
command blocks, 276
command window (Bukkit), 28
commands. *See also* statements
alien invasion, 235
Crafty Crossing game, 271
drawing, 214
comma-separated values (CSV) files,
160–162
comments, 61
computers
connecting electronics to,
124–125
lighting up LEDs from, 129–132
requirements, 14

C

CACTUS block type, 167
calculating points, 264–265

configuring
 drivers, 126–127
 GPIO pins, 152
 Minecraft launcher, 26

connections/connecting
 creating to Minecraft, 62
 electronics to computers, 124–125
 Minecraft to Bukkit server, 26–27

constants
 about, 76
 adjusting, 255–256
 block name, 88
 defined, 56

controlling
 7-segment display with Python
 module, 144–145
 LED, 128–138

conventions, in book, 9–11

coordinates
 absolute, 66
 corner, 54–55
 defined, 36
 negative, 55
 relative, 66
 using, 36–37

copy and paste, 170

Corderio, Jacob (author)
 Minecraft For Dummies, 278

corner coordinates, 54–55

`cottage()` function, 85

counted loop, 68

Craft-bukkit server (website), 9

Crafty Crossing game
 about, 237–239, 271
 adding buttons and displays, 266–270
 building arena, 239–243
 creating obstacles, 243–256
 game play, 256–266

`CraftyCrossing.py` program, 240–242

`createArena` function, 241–242

`createDiamonds()` function, 259–261, 269–270

creating. *See also* building
 bigger shapes, 225–228
 blocks, 64–65
 circles, 194–200

connections to Minecraft, 62

hint-givers, 156–160

holes, 252–256

lines, 194–200

links, 115

Minecraft clock, 200–205

obstacles, 243–256

polyhedrons, 213

programs, 28–30

pyramids, 209–213

small test objects to 3D print, 169–171

spheres, 194–200

walls, 243–246

Creative mode, 5

CSV (comma-separated values) files, 160–162

`csvBuild.py` program, 162–165

CTRL+C, 45, 248

current, 123, 124

D

data files
 about, 155, 191–192
 building 3D block printer, 168–174
 building 3D block scanner, 174–177
 building duplicating machines, 178–191
 building mazes from, 160–168
 making hint-givers, 156–160
 reading data from files, 155–160

`data` variable, 167

`datetime` module, 201–205

`datetime.datetime.now()` function, 202–205

David Says box, 10

`def`, 80, 97

`def house()` : statement, 81–82

Definitions box, 9

demolishing duplicator room, 185–186

`demolishRoom()` function, 186, 187

`detonator.py` program, 149–151

detonators
 about, 145
 wiring buttons, 146–148
 writing detonator program, 148–151

DIAMOND block type, 167, 260–261
diamonds
 collecting, 259–261
 countdown for, 269–270
DIAMONDS constant, 257
diamondsLeft variable, 260, 261
dice.py program, 67
displaying
 coordinates, 37
 menus, 182–183
displays, adding, 266–270
display.write function, 269–270
distanceBetweenPoints()
 function
 alien invasion, 228–234
 block friends, 218, 220–221, 222
double equals (==) symbol, 50
downloading
 Python, 19, 21–22, 23–24
 Python version 2, 19
 starter kit, 20–21, 22–23
draw() function, 227–228
drawCircle() function
 creating Minecraft clock, 202–205
 drawing circles, 197–199
drawFace() function
 creating polyhedrons, 213
 creating pyramids, 209–213
 drawing polygons, 207–209
drawing
 circles, 197–199
 lines, 196–197
 polygons, 206–209
 spheres, 199–200
drawLine() function
 block friends, 219
 creating Minecraft clock, 203–205
 drawing lines, 196–197
drawSphere() function, 199–200
drivers, configuring, 126–127
duplicating machine
 building, 178–191
 writing framework of, 178–181
duplicator room
 building, 183–185
 cleaning, 187
 demolishing, 185–186

printing from, 187–188
scanning from, 186–187
duplicator.py program, 178–181, 181–182, 191

E

e.face variable, 107

electronic circuits
 about, 117, 152–153
 adventure requirements, 118–121
 controlling LED, 128–138
 making detonators, 145–151
 prototyping electronics with
 breadboard, 121–125
 setting up PC or Mac to control,
 125–128
 using 7-segment display, 138–145

elif statement, 102

else statement

building magic bridges, 95
building vanishing bridges, 102
indentations with, 93
moving players, 59
using with **if** statement, 60, 61

experimenting, with lists, 98–101

extracting starter kit, 20–21, 22–23

F

F3 key, 37, 55

faces, 206, 207

file type, 166

FILENAME constant

building 3D block scanner, 174
building mazes, 163
making hint-givers, 159
making small test objects to
 3D print, 173

filenames, getting lists of matching, 191

files. *See also* data files

CSV, 160–162

listing, 189–190

reading lines from, 191

writing lines to, 191

findPointOnCircle() function

creating Minecraft clock, 202–205

creating pyramids, 209–213, 210–213

`findport.cache` program, 127–128
`findPort.py` program, 127–128
Fire Tech Camp (website), 275
flashing LEDs, 132–134
`for cell` loop, 164
`for` loop
 block friends, 220
 building mazes, 163–164, 166
 building multiple blocks with, 67–69
 building streets of houses with, 83–85
 building towers with, 69–70
 colon with, 80
 Crafty Crossing game, 244–246, 251–252
 creating pyramids, 210–213
 features of, 68–69
 indenting with, 68
 making small test objects to
 3D print, 173
 sensing hits on blocks, 107
 speed when using, 71
framework, writing of duplicating
 machine program, 178–181
`friend` object, 218
functions. *See also* statements
 `abs()`, 112
 `append()`, 101, 209
 `buildBridge()`, 95, 97, 102, 113
 `buildRoom()`, 183–185
 calling, 97
 `checkHit()`, 105–108, 111
 `cleanRoom()`, 187
 `clear()`, 227–228
 `clone()`, 218
 `cottage()`, 85
 `createArena`, 241–242
 `createDiamonds()`, 259–261, 269–270
 `datetime.datetime.now()`, 202–205
 defined, 80
 `demolishRoom()`, 186, 187
 `display.write`, 269–270
 `distanceBetweenPoints()`, 218, 220–221, 222, 228–234
 `draw()`, 227–228
 `drawCircle()`, 197–199, 202–205
 `drawFace()`, 207–209, 209–213
 `drawLine()`, 196–197, 203–205, 219
 `drawSphere()`, 199–200
 `findPointOnCircle()`, 202–205, 209–213
 `getBlock()`, 92, 93, 115, 137
 `getHeight()`, 217
 `getLine()`, 219
 `glob.glob()`, 189–190
 `GPIO.cleanup()`, 133, 135–136
 `holes`, 255–256
 `homingBeacon()`, 112, 114, 115
 `house()`, 81, 82, 84, 85, 87, 97
 `house2()`, 88
 `int()`, 172, 181, 205
 `len()`, 234
 `listFiles()`, 189–190
 `maisonette()`, 85
 `mc.getHeight()`, 216
 `mc.getTilesPos()`, 97
 `menu()`, 180
 `MinecraftDrawing.`
 `drawFace`, 208–209
 `MinecraftShape`, 215, 225–234, 244–246, 249–252
 `MinecraftShape.moveTo()`, 245–246, 252
 `moveBy()`, 227
 `myname()`, 80–81
 `open()`, 159, 175
 `output()`, 133
 `placeTreasure()`, 110–111
 `pollBlockHits`, 260
 `postToChat()`, 43, 80, 172
 `print 3D()`, 172–173
 Python, 80–82
 `randint()`, 86, 181, 252–256
 `random.randint()`, 89, 223, 234
 `random.randint()`, 254–256
 `range()`, 68, 69, 71, 107
 `raw_input()`, 72–74, 143, 145, 181
 `readlines()`, 163–164, 166, 168
 `round()`, 205
 `safeFeet()`, 93–94, 95–97

`scan3D()`, 175, 186
`setBlock() /setBlocks()`,
66–67, 71–75, 78–79, 81, 83, 86,
92, 95–97, 149, 151, 183–185,
196–197, 240–242, 250–252,
255–256
7-segment display, 152
`split()`, 163, 167, 168, 172
`str()`, 42, 73, 269–270
`strip()`, 158
`theHoles`, 253–256
`theRiver()`, 250–252
`theWall()`, 244–246, 247–249,
248, 256
`time.sleep()`, 44, 102, 114, 145,
220, 256
`tower()`, 84
`townHouse()`, 85
`write()`, 177
writing for treasure hunt game,
109–110

G

`game` loop, 43–45, 257–261, 264–265,
268–270
Game Over message, 265–266
game play, Crafty Crossing game,
256–266
`gameOver` variable, 245, 257, 261–262,
264–265
general purpose input outputs (GPIOs),
124–125, 134–136
generating random numbers, 85–86
geo-fencing
defined, 35, 53
using, 53–58
writing program, 56–58
`getBlock()` function
about, 92
in block interactions, 93
treasure hunt game, 115
writing magic doormat LED program,
137
`getHeight()` function, 217
`getLine()` function, 219

`getTilesPos()` statement, 92
`GLASS` block type, 77, 88, 239, 242
global variable, 83
`glob.glob()` function, 189–190
`GLOWING_OBSIDIAN` block type, 65
GPIO pins, 152
GPIO plans, 152
`GPIO.cleanup()` function
flashing LEDs, 133
running GPIO programs, 135–136
GPIOs (general purpose input outputs),
124–125, 134–136
graphical user interface (GUI),
Raspberry Pi, 15
`GRASS` block type, 239, 252, 255–256

H

`Handle` field, 161
hardware setup, 267–269
Harris, Nicholas (programmer),
274, 277
hash symbol (#), 61
Hello Minecraft World program, 14,
28–33
“hello world” program, 14
`helloMinecraftWorld.py`
program, 128
help, 9, 19
hint-givers, making, 156–160
holes, creating, 252–256
`holes` function, 255–256
homing beacons, adding, 112–113
`homingBeacon()` function, 112,
114, 115
`house()` function
adding carpets, 87
building streets of houses with, 84
calling code with, 97
modifying, 85
using, 81, 82
`house2()` function, 88
`HOVER_HEIGHT`, 228
Hunt, Andy (author)
Learn to Program with Minecraft Plugins, 278

I

- .`id`, 79
- IDLE
 - code editor, 13
 - permissions, 29
 - programming IDE, 9
 - showing .csv files in, 176
 - starting, 28
 - website, 275
- `if` statement
 - building 3D block scanner, 177
 - building magic bridges, 95
 - building mazes, 164
 - building vanishing bridges, 102
 - checking player location, 47–48
 - colon with, 69, 80
 - indentations with, 57, 93
 - moving players, 59
 - outcomes of, 47
 - random numbers, 223
 - time-left indicator, 270
 - treasure hunt game, 111
 - using to make magic doormats, 46–47
 - using with `else` statement, 60, 61
 - writing Welcome Home game, 49–50
- `if/else` statement
 - building duplicator room, 184–185
 - building mazes, 164
- `import` statement, 81, 224, 268
 - importing Minecraft API, 62
- indentation
 - in block interactions, 93
 - defined, 44, 45
 - importance of, 45, 57
 - Python, 184–185
 - for Welcome Home game, 50
- “independent video games,” 2
- index
 - accessing list items by, 115
 - defined, 99
- infinite loop, 43
- input, reading from keyboard, 72–74
- `input()` statement, 73, 181
 - installing
 - Minecraft on Raspberry Pi, 16–17
 - Python on Apple Mac, 22–24
- Python on Windows PC, 20–22
- starter kit on Apple Mac, 22–24
- starter kit on Windows PC, 20–22
- instructables (website), 277
- `int()` function
 - creating Minecraft clock, 205
 - making small test objects to 3D
 - print, 172
- writing framework of duplicating machine program, 181
- `int()` statement, 73
- interacting, with blocks, 91–116
- interface, 40
- Internet connection, 6
- Internet resources
 - adafruit, 120
 - advanced controls of Minecraft, 37
 - Apple Mac, 9
 - Arduino, 120–121
 - Asdjke and Bro, 278
 - block id numbers, 89
 - book companion, 8, 11, 20, 38, 194, 212, 266, 273
 - Bresenham line algorithm (website), 197
 - Bukkit, 275
 - Bukkit server, 19
 - circuits, 124
 - classicgamesarcade, 271
 - codeacademy, 275
 - command blocks, 276
 - coordinates, 36
 - Craft-bukkit server, 9
 - design tools, 274
 - electronic accelerometer, 153
 - Fire Tech Camp, 275
 - geo-fencing, 53
 - Harris, Nicholas, 274, 277
 - help, 19
 - IDLE, 275
 - IDLE programming IDE, 9
 - instructables, 277
 - inventwithpython, 275
 - Kids Math Games, 213
 - Mac OS X, 9
 - Maplin Electronics, 120, 276

maps, 274
Maths is Fun, 213
MCreator, 274
Met Office, 271
Microsoft Windows, 9
Minecraft, 9
Minecraft BMP builder, 156
Minecraft extra data values, 89
Minecraft Forum, 274
Minecraft Pi edition, 9
Minecraft Pi Edition, 16
Minecraft Wiki, 274
Minecraft worlds, 274
MumboJumbo, 278
Named Binary Tags (NBT), 276
Nottingham City Council, 192
O'Hanlon, Martin, 274, 278
open exchange hosting, 274
Pang, S.K., 275
ProMicro, 127
Python, 9, 14, 20, 275
Python Wiki, 20
random numbers, 86
Raspberry Juice bukkit plug in, 9, 275
Raspberry Pi, 9, 16, 122
RaspberryJuice plugin, 19
Rastrack, 277
Reddit, 274
Redstone, 276
resistor colour code, 123
resistor values, 124
as resources, 273–275
Richardson, Craig, 274
scarabcoder, 274
The Shard, 88, 153
SimplySarc, 277
skpang, 120, 121
Sparkfun, 126, 127, 276
Stampy, 277
starter kits, 14
stuffaboutcode, 206, 273, 277
TARDIS, 278
3D mazes, 192
twitter, 271
Whale, David, 275
Wiley, 128
inventwithpython (website), 275

K

keyboard, reading input from, 72–74
KeyboardInterrupt, 136
Kids Math Games (website), 213

L

labels (7-segment displays), 140
laying carpets, 86–87
Learn to Program with Minecraft Plugins
(Hunt), 278
LEDs. *See* light-emitting diodes (LEDs)
len() function, 234
level, setting as complete, 264–265
level loop, 257–265, 270
levelComplete, 263
LEVELS constant, 264–265
lifts, building, 277
light-emitting diodes (LEDs)
 building circuits that light, 123–124
 controlling, 128–138
 defined, 123
 flashing, 132–134
 lighting up from computers, 129–132
 troubleshooting, 132
 yellow, 134
lighting up, LEDs from computer,
 129–132
line continuation character, 185
line variable, 163
lineidx variable, 172
lines
 drawing, 196–197
 reading from files, 191
 writing to files, 191
LinesCirclesAndSpheres.py
 program
 drawing circles, 198–199
 drawing lines, 195–196
 drawing spheres, 199–200
links, creating, 115
listFiles() function, 189–190
listing files, 189–190
lists
 accessing items in by index, 115
 accessing last item on, 115
 adding to end of, 115
 defined, 98

lists (*continued*)
getting of matching filenames, 191
looping through, 115
printing contents of, 115
sizing, 115
long lines, splitting, 185
looping through lists, 115

M

mA (milliamperes), 124
Mac. *See* Apple Mac
Mac OS X (website), 9
magic bridges, building, 95–97
magic doormats
building, 48–49
LED program, writing, 137–138
using `if` statements to make, 46–47
`magicBridge.py` program, 95–97,
101, 102
main game loop, writing for treasure
hunt game, 109–110
`maisonette()` function, 85
Maplin Electronics (website), 120, 276
maps, websites for, 274
`math` module
block friends, 217
creating Minecraft clock, 201–205
creating pyramids, 209–213
Maths is Fun (website), 213
`maze1.csv`, 161–162
`maze.csv`, 161
mazes, building from data files, 160–168
`mc.getHeight()` function, 216
`mc.getTilesPos()` function, 97
`mcpi` folder, 40
`mcpi.minecraft` module, 40
`mc.player.setTilePos()`,
62, 151
`mc.postToChat()`
about, 97
alien invasion, 234
commas in, 157
indentation with, 50
MCreator (website), 274
`mc.setBlock()`, 88, 164

`mc.setBlocks()`, 88
`menu()` function, 180
menus, displaying, 182–183
messages, posting to Minecraft chat, 62
Met Office (website), 271
metadata, 169
Microsoft Windows (website), 9
`midx` variable, 77, 81
`midy` variable, 77, 81
milliamperes (mA), 124
Minecraft. *See also specific topics*
about, 1, 13–15
advanced controls, 37
configuring launcher, 26
connecting to Bukkit server, 26–27
creating connections to, 62
installing on Raspberry Pi, 16–17
origins of, 2
resources, 273–275
starting on Raspberry Pi, 17–18
starting on Windows PC/Apple Mac,
24–27
website, 9
Minecraft API
accessing, 40
importing, 62
*Minecraft: Awesome Building Ideas for
You*, 278
Minecraft BMP builder (website), 156
Minecraft chat, posting messages to, 62
Minecraft clock, creating, 200–205
Minecraft For Dummies (Corderio), 278
Minecraft Forum (website), 274
`minecraft` module
alien invasion, 228–234
block friends, 217
blocks, 93
creating lines, circles, and spheres,
194–200
creating Minecraft clock, 201–205
creating pyramids, 209–213
drawing polygons, 207–209
`Minecraft` object
block friends, 217
creating bigger shapes, 226–228

Minecraft Pi Edition (website), 9, 16
Minecraft: The Official Beginner's Handbook, 278
Minecraft: The Official Redstone Handbook, 278
Minecraft: The Story of Mojang (documentary), 2
Minecraft Wiki (website), 274
MinecraftClock.py program, 201–205
MinecraftDrawing class
alien invasion, 228–234
block friends, 217, 219
creating Minecraft clock, 202–205
creating pyramids, 210–213
drawing circles, 197–199
drawing lines, 195, 196–197
drawing polygons, 207–209, 208–209
MinecraftDrawing.drawFace function, 208–209
MinecraftPyramids.py program, 209–213
MinecraftShape function
about, 215
alien invasion, 228–234
Crafty Crossing game, 244–246, 249–252
creating bigger shapes, 225–228
MinecraftShapemoveBy() function, 245–246, 252
minecraftstuff module
about, 48, 193, 194, 215
alien invasion, 228–234
block friends, 217
Crafty Crossing game, 244–246
creating bigger shapes, 225
creating Minecraft clock, 201–205
creating pyramids, 209–213
drawing polygons, 207–209
modes, 5
module, 194
Mojang AR, 2
moveBy() function, 227
moving players, 58–61
MumboJumbo (website), 278

MyAdventures folder, 30, 40, 92, 127, 145, 195, 201, 207, 209, 217, 223, 226, 240
myname() : function, 80–81

N

n character, 177
Name field, 161
Named Binary Tags (NBT), 276
negative coordinates, 55
nested loop, 164, 165
newline, 157
None value, 110
“Notch” gamer tag, 2
Nottingham City Council (website), 192

O

object1.csv file, 170–171
obstacles
 creating, 243–256
 running, 246–249
O'Hanlon, Martin (author), 12, 274, 278
open exchange hosting site, 274
open() function
 building 3D block scanner, 175
 making hint-givers, 159
open source Minecraft server, 19
organization, of this book, 7–8
OS X 10.8+, 24
out of time, 261–262
output() function, 133

P

Pang, S.K. (stockist), 275
parameters, 196
pass parameters, 196
pass statement, 241, 250–256, 260–261
PCs. *See* Windows PC
permissions
 IDLE, 29
 to run Bukkit, 25
Persson, Markus (programmer), 2
Philbin, Carrie Ann (author)
 Adventures in Raspberry Pi, 278

`pi`, 136
`placeTreasure()` function, 110–111
players
about, 35, 61–62
building magic doormats, 48–49
building Welcome Home game, 45–52
charging rent, 53–58
checking locations of, 47–48
corner coordinates of field, 54–55
game loop, 43–45
geofencing, 53–58
getting started, 38
getting tile position, 62
improving output, 41–42
moving, 58–61
`postToChat`, 43
sensing position of, 36–45
setting tile position, 62
showing position of, 38–41
tracking, 263
writing geo-fence program, 56–58
writing Welcome Home game, 49–50
plugin, 19
plus symbol (+), 42
points, calculating, 264–265
`points` variable, 264–265
`pollBlockHits` function, 260
`Polygon.py` program, 207–209
polygons, drawing, 206–209
polyhedrons, creating, 213
pop, 101
`pos` variable, 40, 42, 62, 66, 70, 93
posting messages to Minecraft
chat, 62
`postToChat()` function, 43, 80, 172
predictable, 222
`print 3D()` function, 172–173
`print` statement, 40–44, 80–81, 109–112
`print3D.py` program, 171–177, 187–188

printing
from duplicator room, 187–188
list contents, 115
probability, 222
programming, 2–3
programs
creating, 28–30
defined, 36
running, 30–33
stopping, 33–34
testing, 31–33, 48
projects, resources for, 276–277
`ProMicro.inf` file, 126
prototyping, 121–125
pseudo-random numbers, 86
pull-up resistor, 148
pyramids, creating, 209–213
Python
about, 7, 11
case-sensitivity of, 39
defined, 13
downloading, 19, 21–22, 23–24
downloading version 2, 19
functions, 80–82
indentation, 184–185
installing on Apple Mac, 22–24
installing on Windows PC, 20–22
resources, 275
statements, 36
website, 9, 14, 20, 275
writing to drive 7-segment displays, 142–144
Python for Kids (Briggs), 278
Python lists
about, 98, 101
building vanishing bridges with, 101–104
experimenting with, 98–101
Python module, 144–145
Python Shell, 28–29
Python Wiki, 20



Quick Reference Table, 11

R

- radians, 205
- `randint()` function, 86, 181, 252–256
- `random` module, 111, 215, 223, 224
- random numbers
 - defined, 86
 - generating, 85–86
 - using, 222–225
- `random.randrange()` function
 - alien invasion, 234
 - random numbers, 223
 - stopping on random patterns, 89
- `random.randint()` function, 254–256
- `RANGE` constant, 109, 111, 115
- `range()` function, 68, 69, 71, 107
- Raspberry Juice bukkit plug in (website), 9, 275
- Raspberry Pi
 - about, 6
 - building magic bridges, 96
 - displaying coordinates, 37
 - flashing LEDs, 133–134
 - installing Minecraft on, 16–17
 - lighting up LEDs from computer, 130–131
 - requirements for electronic circuits
 - adventure, 120
 - running GPIO programs, 134–135
 - starting Minecraft on, 17–18
 - website, 9, 16
 - writing Python to drive 7-segment displays, 143
- Raspberry Pi Model B+, 120
- RaspberryJuice plugin (website), 19
- Rastrack (website), 277
- `raw_input()` function, 72–74, 143, 145, 181
- reading
 - data from files, 155–160
 - GPIO plans, 152
 - input from keyboard, 72–74
 - lines from files, 191
- `readlines()` function, 163–164, 166, 168
- README file, 26
- red error message, 51
- Reddit (website), 274
- Redstone, 276
- relative coordinates, 66
- rent, charging, 53–58
- `rent.py` program, 53–58, 58–61
- requirements, 5–6, 14, 118–121
- resistor, 123, 148
- resources
 - books, 278
 - Bukkit, 275–276
 - Minecraft, 273–275
 - projects, 276–277
 - Python, 275
 - tutorials, 276–277
 - videos, 277–278
 - websites, 273–275
- return, 181
- Richardson, Craig (programmer), 274
 - Adventures in Python*, 278
- river, building, 249–252
- `RIVERWIDTH`, 251
- `root`, 136
- round brackets (()), 98
- `round()` function, 205
- running
 - GPIO programs, 134–136
 - obstacles, 246–249
 - programs, 30–33

S

- `safeFeet()` function, 93–94, 95–97
- `safeFeet.py` program, 92–94, 95
- `SAND` block type, 65, 163
- sandbox game, 1–2
- `SANDSTONE` block types, 212
- `scan3D()` function
 - building 3D block scanner, 175
 - demolishing duplicator room, 186
- `scan3D.py` program
 - building 3D block scanner, 174–177
 - demolishing duplicator room, 186
- scanning, from duplicator room, 186–187
- scarabcoder (website), 274

score variable, 111
secondsLeft variable, 262
Security Alert, 27
seg7.py program, 144–145
sensing hits on blocks, 105–108
sequential, 246
serial port number, finding, 127–128
setBlock() /setBlocks()
 function
 building blocks, 66–67, 73
 building duplicator room, 183–185
 building houses with, 75
 building magic bridges, 95–97
 building with, 71–72
 calculating corner coordinates with, 74
Crafty Crossing game, 240–242,
 250–252, 255–256
drawing lines, 196–197
extra number of, 86
extra numbers of, 78–79
in **house()** function, 83
indenting, 81
troubleshooting, 79
using with **getBlock()**
 function, 92
writing detonator program, 149, 151
setting
 blocks, 88
 level as complete, 264–265
 player tile position, 62
 positions of blocks, 88
setup
 Apple Mac to control electronic
 circuits, 125–128
 Apple Mac to program Minecraft,
 19–28
 hardware, 267–269
 PC to control electronic circuits,
 125–128
 PC to program Minecraft, 19–28
 Raspberry Pi to program Minecraft,
 15–18
7-segment display
 about, 138–140
 controlling with Python module,
 144–145
functions, 152
using, 152
wiring up, 140–142
writing characters to, 152
writing Python to drive, 142–144
ShapeBlock, 226–228
shapes, creating bigger, 225–228. *See also*
 specific shapes
The Shard (website), 88, 153
showing player position, 38–41
SimplySarc (website), 277
single equals (=) symbol, 50
SIZE constant, 77, 79, 84, 176,
 185–186
size variable, 73, 172
sizing lists, 115
“Sky Hunt” game. *See* treasure
 hunt game
skyHunt.py program, 109–110
SLATE block type, 88
source files, for electronic
 circuits, 121
space, clearing, 71–74
Sparkfun (website), 276
Specialty field, 161
spheres, drawing, 199–200
split() function
 building mazes, 163, 167, 168
 making small test objects to 3D
 print, 172
square brackets ([]), 98
square root, 221
Stampy (YouTube user), 277
“StartBukkit.Command” message, 25
starter kits
 about, 6, 14–15
 downloading, 20–21, 22–23
 extracting, 20–21, 22–23
 installing on Apple Mac, 22–24
 installing on Windows PC, 20–22
starting
 Crafty Crossing game, 257–259
IDLE, 28
Minecraft on Raspberry Pi, 17–18
Minecraft on Windows PC/Apple
 Mac, 24–27

statements. *See also* functions
about, 36
`break`, 182
`def house()`, 81–82
`elif`, 102
`else`, 59, 60, 61, 93, 95, 102
`getTilesPos()`, 92
`if`, 45–50, 57, 59, 60, 61, 69, 80, 93,
95, 102, 111, 164, 177, 223, 270
`if/else`, 164, 184–185
`import`, 81, 224, 268
`input()`, 73, 181
`int()`, 73
`pass`, 241, 250–256, 260–261
`print`, 40–44, 80–81, 109–112
Python, 36
Stewart, Becky (author)
 Adventures in Arduino, 278
`STONE` block type, 64, 67, 88
stopping
 Bukkit, 27–28
 multi-threaded program running
 IDLE, 248
 programs, 33–34
`str()` function, 42, 73, 269–270
strings
 defined, 42
 stripping white space from, 191
`strip()` function, 158
stripping white space from strings, 191
stuffaboutcode (website), 273, 277
`sudo`, 136
`sudo python detonator.py`
 program, 150
`sudo Python testDisplay.py`
 program, 144
Survival mode, 5
syntax, 52
syntax error, 51–52

T

Tab key, 45
TARDIS (website), 278
tasks, automating, 276
`testDisplay2.py` program,
144–145

`testDisplay.py` program, 142–144,
144–145
testing programs, 31–33, 48
`testLED.py` program, 132–134,
136, 137
`theHoles` function, 253–256
`theRiver()` function, 250–252
`theWall()` function, 244–246,
247–249, 248, 256
`thread.start_new_thread()`,
247–249, 248
3D block printer, building, 168–174
3D block scanner, building, 174–177
3D printer, writing, 171–173
3D structures
 about, 193, 213–214
 creating lines, circles, and spheres,
 194–200
 creating Minecraft clock, 200–206
 drawing polygons, 206–209
 `minecraftstuff` module, 194
 pyramids, 209–213
time delay, 96
`time` module
 alien invasion, 228–234
 block friends, 217
 creating lines, circles, and
 spheres, 195
 creating Minecraft clock, 201–205
time-left indicator, 270
`TIMEOUT` variable, 112
`TIMEOUTS` constant, 257, 262
`timer` variable, 112
`time.sleep()` function
 block friends, 220
 building vanishing bridges, 102
 controlling 7-segment display with
 Python module, 145
 in `game` loops, 44
 modifying, 256
 treasure hunt game, 114
`tipChat.py` file, 158, 159, 168
`tips` list, 158
Tips & Tricks box, 10
`tips.txt` file, 156–160, 159
`tower()` function, 84

`tower.py` program, 69–70, 79
`townHouse()` function, 85
tracking players, 263
treasure, collecting, 111–112
treasure hunt game
 about, 108
 adding bridge builder, 113–115
 adding homing beacon, 112–113
 collecting treasure when it's hit,
 111–112
 placing in the sky in treasure hunt
 game, 110–111
 placing treasure in the sky,
 110–111
 writing, 108–115
writing functions and main game
 loop, 109–110
`treasure_x` variable, 111, 112
`tree.csv` file, 176, 177
trigonometry, 200, 201
troubleshooting LEDs, 132
`try/finally`
 flashing LEDs, 133
 running GPIO programs, 136
`try/finally/GPIO.cleanup()`, 135
tuples, 104, 248, 249
tutorials. *See also* videos
 Adafruit, 277
 Bukkit, 277
 Redstone, 276
 resources, 276–277
twitter (website), 271
2D structures
 about, 193, 213–214
 creating lines, circles, and spheres,
 194–200
 creating Minecraft clock, 200–206
 drawing polygons, 206–209
 `minecraftstuff` module, 194
 pyramids, 209–213

U

“unidentified developer” message, 24
upper case, for constants, 76
Upton, Eben (programmer), 122

V

vanishing bridges, building with
 Python lists, 101–104
`vanishingBridge.py` program,
 101–108, 113
variables
 `a`, 61, 69, 70
 `anotherGo`, 179
 boolean, 181, 182
 `bridgeBlocks`, 251
 `bridgePOS`, 251
 `data`, 167
 defined, 40, 41
 `diamondsLeft`, 260, 261
 `e.face`, 107
 `gameOver`, 245, 257, 261–262,
 264–265
 global, 83
 `line`, 163
 `lineidx`, 172
 `midx`, 77, 81
 `midy`, 77, 81
 `points`, 264–265
 `pos`, 40, 42, 62, 66, 70, 93
 `score`, 111
 `secondsLeft`, 262
 `size`, 73, 172
 `TIMEOUT`, 112
 `timer`, 112
 `treasure_x`, 111, 112
 `wallBlocks`, 244–246
 `wallPos`, 245–246
versions, 25
Video box, 10
videos
 Apple Mac setup, 20
 Bling, Seth (YouTube user), 277
 block friend tutorial, 216
 building houses tutorial, 76
 building lift, 277
 Crafty Crossing game tutorial, 238
 creating Minecraft clock, 200
 detonator tutorial, 145
 maze game tutorial, 160
 Raspberry Pi setup, 15
 resources, 277–278

vanishing bridge game tutorial, 101
Welcome Home game tutorial, 46
Windows PC setup, 20
virtual world, 2
voltage, 123

W

wallBlocks variable, 244–246
wallPos variable, 245–246
walls, creating, 243–246
WallZ constant, 245
Walmsley, Ryan (programmer), 277
Warning box, 10
WATER block type, 65, 250–252
WATER_FLOWING block type, 167
websites
 adafruit, 120
 advanced controls of Minecraft, 37
 Apple Mac, 9
 Arduino, 120–121
 Asdjke and Bro, 278
 block id numbers, 89
 book companion, 8, 11, 20, 38, 194,
 212, 266, 273
 Bresenham line algorithm
 (website), 197
 Bukkit, 275
 Bukkit server, 19
 circuits, 124
 classicgamesarcade, 271
 codeacademy, 275
 command blocks, 276
 coordinates, 36
 Craft-bukkit server, 9
 design tools, 274
 electronic accelerometer, 153
 Fire Tech Camp, 275
 geo-fencing, 53
 Harris, Nicholas, 274, 277
 help, 19
 IDLE, 275
 IDLE programming IDE, 9
 instructables, 277
 inventwithpython, 275
 Kids Math Games, 213
 Mac OS X, 9
 Maplin Electronics, 120, 276

maps, 274
Maths is Fun, 213
MCreator, 274
Met Office, 271
Microsoft Windows, 9
Minecraft, 9
Minecraft BMP builder, 156
Minecraft extra data values, 89
Minecraft Forum, 274
Minecraft Pi edition, 9
Minecraft Pi Edition, 16
Minecraft Wiki, 274
Minecraft worlds, 274
MumboJumbo, 278
Named Binary Tags (NBT), 276
Nottingham City Council, 192
O'Hanlon, Martin, 274, 278
open exchange hosting, 274
Pang, S.K., 275
ProMicro, 127
Python, 9, 14, 20, 275
Python Wiki, 20
random numbers, 86
Raspberry Juice bukkit plug in,
 9, 275
Raspberry Pi, 9, 16, 122
RaspberryJuice plugin, 19
Rastrack, 277
Reddit, 274
Redstone, 276
resistor colour code, 123
resistor values, 124
as resources, 273–275
Richardson, Craig, 274
scarabcoder, 274
The Shard, 88, 153
SimplySarc, 277
skpang, 120, 121
Sparkfun, 126, 127, 276
Stampy, 277
starter kits, 14
stuffaboutcode, 206, 273, 277
TARDIS, 278
3D mazes, 192
twitter, 271
Whale, David, 275
Wiley, 128

Welcome Home game
building, 45–52
writing, 49–50
`welcomeHome.py` program, 53
`welcomeLED.py` program, 137
Whale, David (author), 12, 275
What Happens? box, 10
`whereAmI.py` program, 41, 43, 66
while loop
alien invasion, 229–234
in block interactions, 94
colon with, 80
Crafty Crossing game, 245–246,
253–256, 255–256, 260–261
indenting with, 44, 68
while True loop
about, 67
building vanishing bridges, 103
indentation with, 50, 57
moving players, 59
running GPIO programs, 136
white space, stripping from strings, 191
wildcard, 190
Windows PC
building magic bridges, 96
configuring drivers, 126
displaying coordinates, 37
installing starter kit on, 20–22
requirements for electronic circuits
adventure, 120–121
running GPIO programs, 134
setting up to control electronic
circuits, 125–128
starting Minecraft on, 24–27

wiring
buttons, 146–148
7-segment displays, 140–142
WOOD block type, 88
`WoodenHorse.py` program,
226–228
WOOL block type, 78–79, 86–87, 88, 89
write() function, 177
writing
characters to 7-segment display, 152
detonator program, 148–151
framework of duplicating machine
program, 178–181
functions for treasure hunt game,
109–110
geo-fence program, 56–58
GPIO plans, 152
lines to files, 191
magic doormat LED program,
137–138
main game loop for treasure hunt
game, 109–110
Python to drive 7-segment displays,
142–144
3D printer, 171–173
treasure hunt game, 108–115
Welcome Home game, 49–50

X
`x` windows, 15

Y
yellow LED, 134

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.