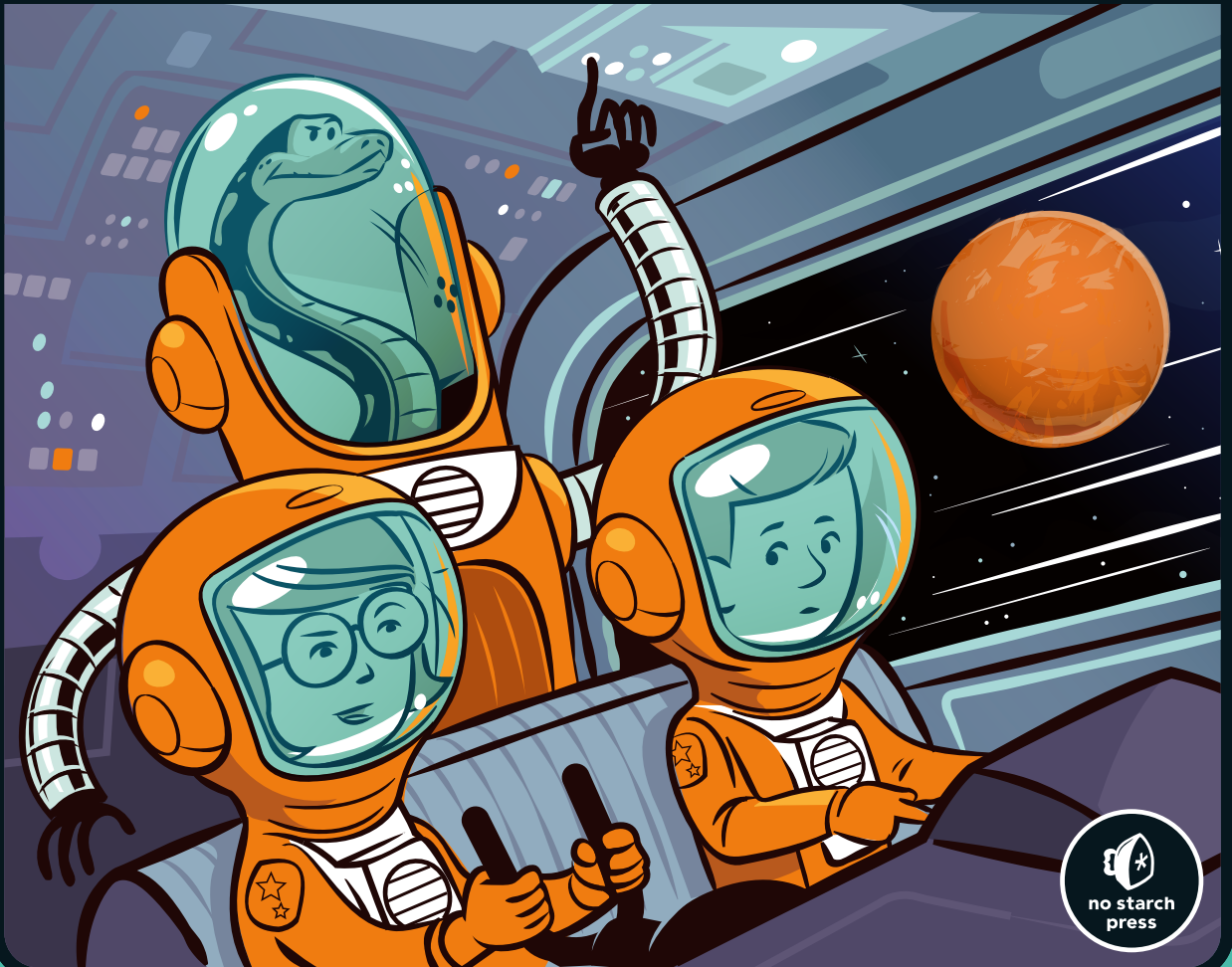


MISSION PYTHON

CODE A SPACE ADVENTURE GAME!

SEAN MCMANUS



MISSION PYTHON

CODE A SPACE ADVENTURE GAME!

BY SEAN MCMANUS



**no starch
press**

San Francisco

MISSION PYTHON. Copyright © 2018 by Sean McManus.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-10: 1-59327-857-8

ISBN-13: 978-1-59327-857-1

Publisher: William Pollock

Production Editor: Riley Hoffman

Cover Illustration: Josh Ellingson

Game Illustrations: Rafael Pimenta

Developmental Editor: Liz Chadwick

Technical Reviewer: Daniel Aldred

Copyeditor: Anne Marie Walker

Compositor: Riley Hoffman

Proofreader: Emelie Burnette

The following images are reproduced with permission:

Figure 1-1 courtesy of Johnson Space Center, NASA

Figure 1-6 courtesy of NASA/JPL-Caltech/UCLA

Figure 1-7 image of Mars courtesy of NASA

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

245 8th Street, San Francisco, CA 94103

phone: 1.415.863.9900; info@nostarch.com

www.nostarch.com

Library of Congress Control Number: 2018950581

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

To my wife, Karen, with thanks for all
her support throughout this mission;
and to Leo, our wonderful son, who is
taking us on the most amazing journey.

ABOUT THE AUTHOR

Sean McManus is an expert technology and business writer. His other books include *Cool Scratch Projects in Easy Steps*, *Scratch Programming in Easy Steps*, *Coder Academy*, and *Raspberry Pi For Dummies* (co-authored with Mike Cook). As a freelance copywriter, he writes for many of the world's largest technology companies. His novel for adults, *Earworm*, goes undercover in the music industry, exposing a conspiracy to replace bands with computer-generated music. He has been a Code Club volunteer, helping children at a local school to learn computer programming. Visit his website at www.sean.co.uk for sample chapters and bonus content from his books.

ABOUT THE TECHNICAL REVIEWER

Daniel Aldred is a passionate and experienced teacher of computer science. He leads the computing department at a CAS hub school that supports and develops other schools and organizations in the local area. He frequently writes for *Linux User & Developer* and has created resources and projects for Raspberry Pi, Pimoroni, micro:bit, and Cambridge International Assessment. In his spare time he curates two websites, www.canyoucompute.co.uk for the UK Computing examination course and www.tecoed.co.uk for his own personal hacks. Daniel also led and managed a team of eight students to win the first Astro Pi competition, where the astronaut Major Tim Peake ran their program aboard the ISS.

BRIEF CONTENTS

Acknowledgments	xv
Introduction	1
Chapter 1: Your First Spacewalk	13
Chapter 2: Lists Can Save Your Life	33
Chapter 3: Repeat After Me	47
Chapter 4: Creating the Space Station	59
Chapter 5: Preparing the Space Station Equipment	79
Chapter 6: Installing the Space Station Equipment	97
Chapter 7: Moving into the Space Station.	111
Chapter 8: Repairing the Space Station	127
Chapter 9: Unpacking Your Personal Items	151
Chapter 10: Make Yourself Useful	171
Chapter 11: Activating Safety Doors	183
Chapter 12: Danger! Danger! Adding Hazards.	197
Appendix A: Escape: The Complete Game Listing	217
Appendix B: Table of Variables, Lists, and Dictionaries.	245
Appendix C: Debugging Your Listings	249
Index	253

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

XV

INTRODUCTION 1

How to Use This Book.	1
What's in This Book?	2
Installing the Software.	3
Installing the Software on Raspberry Pi	4
Installing Python on Windows.	4
Installing Pygame Zero on Windows	5
Installing the Software on Other Machines	6
Downloading the Game Files.	7
Downloading and Unzipping the Files on a Raspberry Pi.	7
Unzipping the File on a Windows PC	8
What's in the ZIP File.	8
Running the Game	9
Running Pygame Zero Programs on the Raspberry Pi	9
Running Pygame Zero Programs in Windows	10
Playing the Game	11

1 YOUR FIRST SPACEWALK 13

Starting the Python Editor	14
Starting IDLE in Windows 10	14
Starting IDLE in Windows 8	15
Starting IDLE on the Raspberry Pi	15
Introducing the Python Shell.	15
Displaying Text	16
<i>Training Mission #1</i>	17
Outputting and Using Numbers.	17
Introducing Script Mode	18
Creating the Starfield	18
Understanding the Program So Far	21
Stopping Your Pygame Zero Program.	23
Adding the Planet and Spaceship	23
Changing Perspective: Flying Behind the Planet	24
<i>Training Mission #2</i>	25
Spacewalking!	26

<i>Training Mission #3</i>	27
Understanding the Spacewalk Listing	27
<i>Training Mission #4</i>	29
Are You Fit to Fly?	29
<i>Mission Debrief.</i>	31

2 **LISTS CAN SAVE YOUR LIFE** **33**

Making Your First List: The Take-Off Checklist	34
Seeing Your List	35
Adding and Removing Items	35
Using Index Numbers	36
Inserting an Item	36
Accessing an Individual Item	36
Replacing an Item	37
Deleting an Item	37
<i>Training Mission #1</i>	37
Creating the Spacewalk Checklist	38
<i>Training Mission #2</i>	38
A List of Lists: The Flight Manual	38
Making a List of Lists	38
<i>Training Mission #3</i>	39
Finding an Item in the Flight Manual	39
Combining Lists	40
Making Maps from Lists: The Emergency Room	41
Making the Map	42
Finding an Emergency Item	43
<i>Training Mission #4</i>	44
Swapping Items in the Room	44
<i>Training Mission #5</i>	45
Are You Fit to Fly?	45
<i>Mission Debrief.</i>	46

3 **REPEAT AFTER ME** **47**

Displaying Maps with Loops	48
Making the Room Map	48
Displaying the Map with a Loop	49
<i>Training Mission #1</i>	50
Loop the Loop	50
Nesting Loops to Get Room Coordinates	50
Cleaning Up the Map	52
<i>Training Mission #2</i>	53
Displaying a 3D Room Image	53
Understanding How the Room Is Drawn	55
Working Out Where to Draw Each Item	56
<i>Training Mission #3</i>	57
Are You Fit to Fly?	57
<i>Mission Debrief.</i>	58

4	CREATING THE SPACE STATION	59
	Automating the Map Making Process	59
	How the Automatic Map Maker Works	60
	Creating the Map Data.	60
	Writing the GAME_MAP Code	62
	Testing and Debugging the Code.	65
	Generating Rooms from the Data	66
	How the Room Generating Code Works	68
	Creating the Basic Room Shape	69
	Adding Exits.	71
	Testing the Program.	72
	<i>Training Mission #1</i>	72
	Exploring the Space Station in 3D	72
	<i>Training Mission #2</i>	75
	Making Your Own Maps	76
	Are You Fit to Fly?	76
	<i>Mission Debrief.</i>	77

5	PREPARING THE SPACE STATION EQUIPMENT	79
	Creating a Simple Planets Dictionary	80
	Understanding the Difference Between a List and a Dictionary	80
	Making an Astronomy Cheat Sheet Dictionary	80
	Error-Proofing the Dictionary.	82
	<i>Training Mission #1</i>	82
	Putting Lists Inside Dictionaries	83
	Extracting Information from a List Inside a Dictionary	84
	<i>Training Mission #2</i>	85
	Making the Space Station Objects Dictionary	85
	Adding the First Objects in Escape	87
	Viewing Objects with the Space Station Explorer	89
	Designing a Room.	89
	<i>Training Mission #3</i>	91
	Adding the Rest of the Objects	91
	<i>Training Mission #4</i>	95
	Are You Fit to Fly?	95
	<i>Mission Debrief.</i>	96

6	INSTALLING THE SPACE STATION EQUIPMENT	97
	Understanding the Dictionary for the Scenery Data.	97
	Adding the Scenery Data	99
	Adding the Perimeter Fence for the Planet Surface	102
	Loading the Scenery into Each Room	104
	Updating the Explorer to Tour the Space Station	107
	<i>Training Mission #1</i>	109
	Are You Fit to Fly?	109
	<i>Mission Debrief.</i>	110

7	MOVING INTO THE SPACE STATION	111
Arriving on the Space Station	112	
Disabling the Room Navigation Controls in the EXPLORER Section	112	
Adding New Variables	112	
Teleporting onto the Space Station	115	
Adding the Movement Code	116	
Understanding the Movement Code	119	
<i>Training Mission #1</i>	122	
Moving Between Rooms	122	
Are You Fit to Fly?	126	
<i>Mission Debrief.</i>	126	
8	REPAIRING THE SPACE STATION	127
Sending Information to a Function	128	
Creating a Function that Receives Information	128	
How It Works	129	
<i>Training Mission #1</i>	129	
Adding Variables for Shadows, Wall Transparency, and Colors	130	
Deleting the EXPLORER Section	132	
Adding the DISPLAY Section	133	
Adding the Functions for Drawing Objects	134	
Drawing the Room	136	
Understanding the New draw() Function	138	
Positioning the Room on Your Screen	141	
Making the Front Wall Fade In and Out	142	
Displaying Hints, Tips, and Warnings	145	
Showing the Room Name When You Enter the Room	146	
Are You Fit to Fly?	148	
<i>Mission Debrief.</i>	149	
9	UNPACKING YOUR PERSONAL ITEMS	151
Adding the Props Information	151	
Adding Props to the Room Map	154	
Finding an Object Number from the Room Map	157	
Picking Up Objects	159	
Picking Up Props	159	
Adding the Keyboard Controls	160	
Adding the Inventory Functionality	161	
Displaying the Inventory	162	
Adding the Tab Keyboard Control	164	
Testing the Inventory	165	
Dropping Objects	166	
<i>Training Mission #1</i>	167	
Examining Objects	168	
<i>Training Mission #2</i>	169	
Are You Fit to Fly?	169	
<i>Mission Debrief.</i>	170	

10	MAKE YOURSELF USEFUL	171
Adding the Keyboard Control for Using Objects		172
Adding Standard Messages for Using Objects		172
Adding the Game Progress Variables		174
Adding the Actions for Specific Objects		174
Combining Objects		177
<i>Training Mission #1</i>		179
Adding the Game Completion Sequence		180
Exploring the Objects		180
Are You Fit to Fly?		181
11	ACTIVATING SAFETY DOORS	183
Planning Where to Put Safety Doors		184
Positioning the Doors		185
Adding Access Controls		185
Making the Doors Open and Close		187
Adding the Door Animation		189
<i>Training Mission #1</i>		190
Shutting the Timed Door		190
Adding a Teleporter		192
<i>Training Mission #2</i>		193
Activating the Airlock Security Door		193
Removing Exits for Your Own Game Designs		195
Mission Accomplished?		196
Are You Fit to Fly?		196
12	DANGER! DANGER! ADDING HAZARDS	197
Adding the Air Countdown		198
Displaying the Air and Energy Bars		198
Adding the Air Countdown Functions		199
Starting the Air Countdown and Sounding the Alarm		202
<i>Training Mission #1</i>		202
Adding the Moving Hazards		203
Adding the Hazard Data		204
Sapping the Player's Energy		205
Starting and Stopping Hazards		205
Setting Up the Hazard Map		208
Making the Hazards Move		208
Displaying Hazards in the Room		210
<i>Training Mission #2</i>		211
Stopping the Player from Walking Through Hazards		212
Adding the Toxic Spills		212
Making the Finishing Touches		213
Disabling the Teleporter		213
Cleaning Up the Data		213
Your Adventure Begins		214

Your Next Mission: Customizing the Game	215
Are You Fit to Fly?	216
<i>Mission Debrief.</i>	216

A	
ESCAPE: THE COMPLETE GAME LISTING	217

B	
TABLE OF VARIABLES, LISTS, AND DICTIONARIES	245

C	
DEBUGGING YOUR LISTINGS	249

Indentation	250
Case Sensitivity	251
Parentheses and Brackets	251
Colons	251
Commas	252
Images and Sounds	252
Spelling	252

INDEX	253
--------------	------------

ACKNOWLEDGMENTS

Many thanks to everyone at No Starch Press who worked hard to bring you this book, including developmental editor Liz Chadwick, production editor Riley Hoffman, copyeditor Anne Marie Walker, proofreaders Emelie Burnette and Meg Sneeringer, and production manager Serena Yang. Thank you to Tyler Ortman, who commissioned the book, and Bill Pollock, for his support on this project. Josh Ellingson created the stunning cover artwork. Thank you to Amanda Hariri, Anna Morrow, and Rachel Barry for their support with marketing.

Rafael Pimenta designed the awesome graphics for the game. Daniel Aldred was the technical editor, testing the code and providing feedback on the text. Thanks to them both.

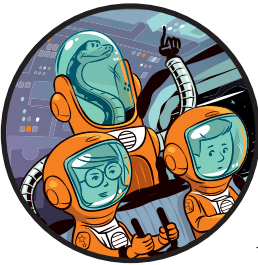
We wouldn't have been able to create this book without the dedicated work of the open source community. Daniel Pope created Pygame Zero and helped with research queries. You can learn about some more cool features of Pygame Zero that weren't required for our mission at <http://pygame-zero.readthedocs.io/en/latest/>. Pygame Zero extends Pygame, so thanks also to the Pygame development team and to the wider Python community who contribute to its success.

NASA allows us to use many of its images to tell our story, for which we are grateful. Its work is hugely inspiring.

Thank you to Russell Barnes, Sam Alder, Eben Upton, and Carrie Anne Philbin at the Raspberry Pi Foundation who helped to get this project off the ground.

Finally, thank you for reading the book! If you enjoy it, please consider sharing a review, tweet, or blog post to help others to discover it. In any event, I hope you enjoy it.

INTRODUCTION



Air is running out. There's a leak in the space station, so you've got to act fast. Can you find your way to safety? You'll need to navigate your way around the space station, find access cards to unlock doors, and fix your damaged space suit. The adventure has begun!

And it starts here: on Earth, at mission command, also known as your computer. This book shows you how to use Python to build a space station on Mars, explore the station, and escape danger in an adventure game complete with graphics. Can you think like an astronaut to make it to safety?

HOW TO USE THIS BOOK

By following the instructions in this book, you can build a game called *Escape* with a map to explore and puzzles to solve. It's written in Python, a popular programming language that is easy to read. It also uses Pygame Zero, which adds some instructions for managing images and sounds, among other things. Bit by bit, I'll show you how to make the game and how the main

parts of the code work, so you can customize it or build your own games based on my game code. You can also download all the code you need. If you get stuck or just want to jump straight into playing the game and seeing it work, you can do so. All the software you need is free, and I've provided instructions for Windows PCs and the Raspberry Pi. I recommend you use the Raspberry Pi 3 or Raspberry Pi 2. The game may run too slowly to enjoy on the Pi Zero, original Model B+, and older models.

There are several different ways you can use the book and the game:

- **Download the game, play it first, and then use the book to understand how it works.** This way, you eliminate the risk of seeing any spoilers in the book before you play the game! Although I've kept them to a minimum, you might notice a few clues in the code as you read the book. If you get really stuck on a problem in the game, you can try reading the code to work out the solution. In any case, I recommend you run the game at least once to see what you'll be building and learn how to run your programs.
- **Build the game, and then play it.** This book guides you through creating the game from start to finish. As you work your way through the chapters, you'll add new sections to the game and see how they work. If you can't get the code working at any point, you can just use my version of the code listing and continue building from there. If you choose this route, avoid making any custom changes to the game until you've built it, played it, and finished it. Otherwise, you might accidentally make the game impossible to complete. (It's okay to make any changes I suggest in the exercises.)
- **Customize the game.** When you understand how the program works, you can change it by using your own maps, graphics, objects, and puzzles. The *Escape* game is set on a space station, but yours could be in the jungle, under the sea, or almost anywhere. You could use the book to build your own version of *Escape* first, or use my version of the final game and customize that. I'd love to see what you make using the program as a starting point! You can find me on Twitter at @musicandwords or visit my website at www.sean.co.uk.

WHAT'S IN THIS BOOK?

Here's a briefing on what's in store for you as you embark on your mission.

- **Chapter 1** shows you how to go on a spacewalk. You'll learn how to use graphics in your Python programs using Pygame Zero and discover some of the basics of making Python programs.
- **Chapter 2** introduces *lists*, which store much of the information in the *Escape* game. You'll see how to use lists to make a map.
- **Chapter 3** shows you how to get parts of a program to repeat and how to use that knowledge to display a map. You'll also design a room layout for the space station, using wall pillars and floor tiles.

- In **Chapter 4**, you'll start to build the *Escape* game, laying down the blueprints for the station. You'll see how the program understands the station layout and uses it to create the fabric for the rooms, putting the walls and floor in place.
- In **Chapter 5**, you'll learn how to use *dictionaries* in Python, which are another important way of storing information. You'll add information for all the objects the game uses, and you'll see how to create a preview of your own room design. When you extend the program in **Chapter 6**, you'll see all the scenery in place and will be able to look at all the rooms.
- After building the space station, you can move in. In **Chapter 7**, you'll add your astronaut character and discover how to move around the rooms and animate movements.
- **Chapter 8** shows you how to polish the game's graphics with shadows, fading walls, and a new function to draw the rooms that fixes the remaining graphical glitches.
- When the space station is operational, you can unpack your personal effects. In **Chapter 9**, you'll position items the player can examine, pick up, and drop. In **Chapter 10**, you'll see how to use and combine items, so you can solve puzzles in the game.
- The space station is nearly complete. **Chapter 11** adds safety doors that restrict access to certain zones. Just as you're putting your feet up and celebrating a job well done, there's danger around the corner, as you'll add moving hazards in **Chapter 12**.

As you work through the book, you'll complete training missions that give you an opportunity to test your programs and your coding skills. The answers, if you need them, are at the end of each chapter.

The appendixes at the back of the book will help you, too. **Appendix A** contains the listing for the whole game. If you're not sure where to add a new chunk of code, you can check here. **Appendix B** contains a table of the most important variables, lists, and dictionaries if you can't remember what's stored where, and **Appendix C** has some debugging tips if a program doesn't work for you.

For more information and supporting resources for the book, visit the book's website at www.sean.co.uk/books/mission-python/. You can also find information and resources at <https://nostarch.com/missionpython/>.

INSTALLING THE SOFTWARE

The game uses the Python programming language and Pygame Zero, which is software that makes it easier to handle graphics and sound. You need to install both of these before you begin.

NOTE For updated installation instructions, visit the book's web page at <https://nostarch.com/missionpython/>.

INSTALLING THE SOFTWARE ON RASPBERRY PI

If you're using a Raspberry Pi, Python and Pygame Zero are already installed. You can skip ahead to “Downloading the Game Files” on page 7.

INSTALLING PYTHON ON WINDOWS

To install the software on a Windows PC, follow these steps:

1. Open your web browser and visit <https://www.python.org/downloads/>.
2. At the time of this writing, 3.7 is the latest version of Python, but Pygame isn't available for easy installation on it yet. I recommend you use the latest version of Python 3.6 instead (3.6.6 at the time of writing). You can find old versions of Python farther down the screen on the downloads page (see Figure 1). Save the file on your desktop or somewhere else you can easily find it. (Pygame Zero works only with Python 3, so if you usually use Python 2, you'll need to switch to Python 3 for this book.)

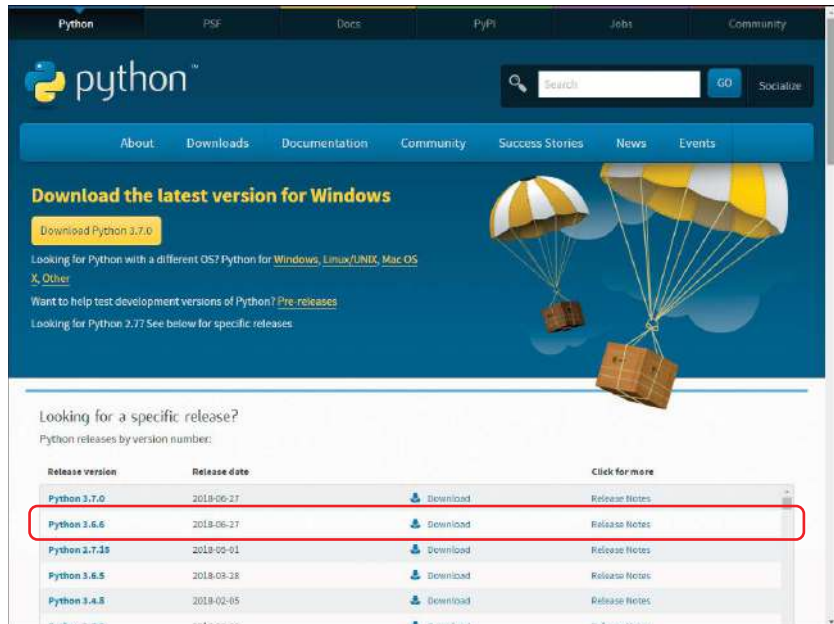


Figure 1: The Python downloads page

3. When the file has downloaded, double-click it to run it.
4. In the window that opens, select the checkbox to Add Python 3.6 to PATH (see Figure 2).
5. Click **Install Now**.



Figure 2: The Python installer

6. If you're asked whether you want to allow this application to make changes to your device, click **Yes**.
7. Python will take a few minutes to install. When it finishes, click **Close** to complete the installation.

INSTALLING PYGAME ZERO ON WINDOWS

Now that you have Python installed on your computer, you can install Pygame Zero. Follow these steps:

1. Hold down the **Windows Start key** and press **R**. The Run window should open (see Figure 3).
2. Enter `cmd` (see Figure 3). Press ENTER or click **OK**.

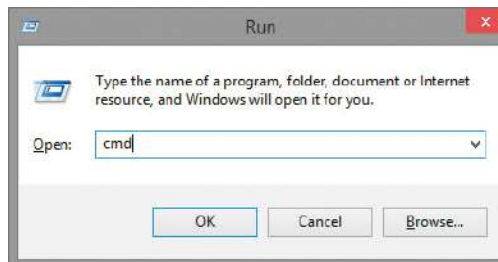


Figure 3: The Windows Run dialog box

3. The command line window should open, as shown in Figure 4. Here you can enter instructions for managing files or starting programs. Enter `pip install pgzero` and press ENTER at the end of the line.

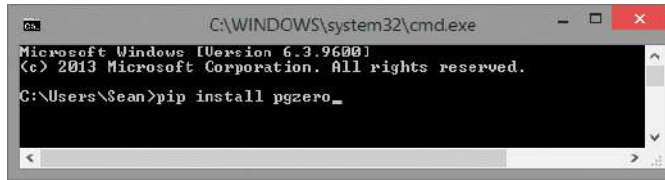


Figure 4: The command line window

4. Pygame Zero should start to install. It will take a few moments, and you'll know it's finished when your `>` prompt appears again.
5. If you get an error message saying that `pip` is not recognized, try installing Python again. You can uninstall Python first by running the installation program again or using the Windows Control panel. Make sure you select the box for the `PATH` when installing Python (see Figure 2). After you have reinstalled Python, try installing Pygame Zero again.
6. When Pygame Zero has finished downloading and you can type again, enter the following:

```
echo print("Hello!") > test.py
```

7. This line creates a new file called *test.py* that contains the instruction `print("Hello!")`. I'll explain the `print()` instruction in Chapter 1, but for now, this is just a quick way to make a test file. Be careful when you enter the parentheses (curved brackets) and quotation marks: if you miss one, the file won't work properly.
8. Open the test file by entering the following:

```
pgzrun test.py
```

9. After a short delay, a blank window should open with the title *Pygame Zero Game*. Click the command line window again to bring it to the front: you should see the text `Hello!` Press `CTRL-C` in the command line window to stop the program.
10. If you want to delete your test program, enter `del test.py`.

INSTALLING THE SOFTWARE ON OTHER MACHINES

Python and Pygame Zero are available for other computer systems. Pygame Zero has been designed in part to enable games to work across different computers, so the *Escape* code should run wherever Pygame Zero runs. This book only provides guidance for users of Windows and Raspberry Pi computers. But if you have a different computer, you can download Python at <https://www.python.org/downloads/> and can find advice on installing Pygame Zero at <http://pygame-zero.readthedocs.io/en/latest/installation.html>.

DOWNLOADING THE GAME FILES

I've provided all the program files, sounds, and images you need for the *Escape* game. You can also download all the listings in the book, so if you can't get one to work, you can use mine instead. All the book's content downloads as a single ZIP file called *escape.zip*.

DOWNLOADING AND UNZIPPING THE FILES ON A RASPBERRY PI

To download the game files on a Raspberry Pi, follow these steps, and refer to Figure 5. The numbers in Figure 5 show you where to do each step.

- ❶ Open your web browser and visit <https://nostarch.com/missionpython/>. Click the link to download the files.
- ❷ From your desktop, click the File Manager icon on the taskbar at the top of the screen.
- ❸ Double-click your Downloads folder to open it
- ❹ Double-click the *escape.zip* file.
- ❺ Click the **Extract Files** button to open the Extract Files dialog box.
- ❻ Change the folder that you'll extract to so it reads */home/pi/escape*.
- ❼ Ensure that the option is selected to Extract files with full path.
- ❽ Click **Extract**.

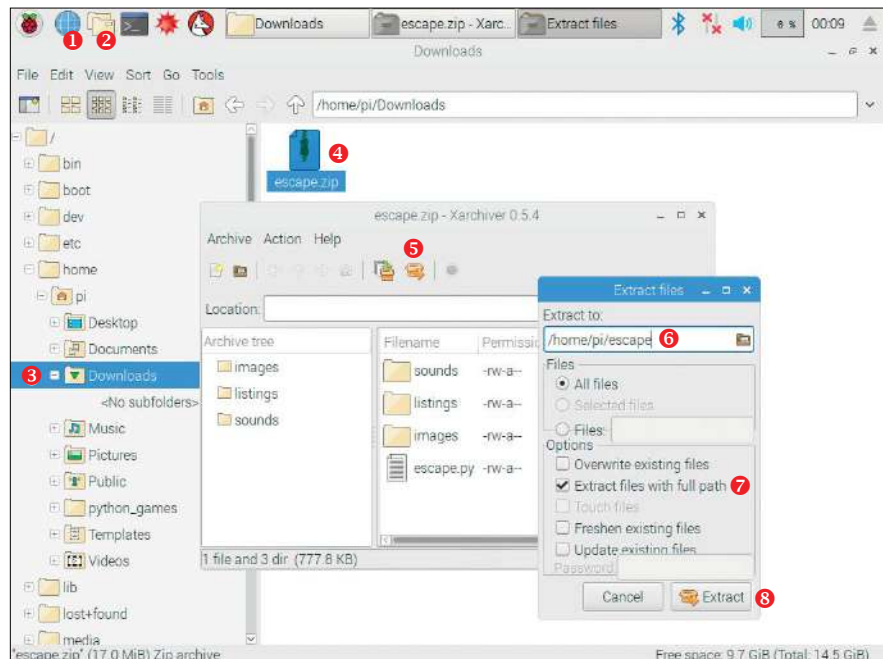


Figure 5: The steps you should take to unzip your files

UNZIPPING THE FILE ON A WINDOWS PC

To unzip the files on a Windows PC, follow these steps.

1. Open your web browser and visit <https://nostarch.com/missionpython/>. Click the link to download the files. Save the ZIP file on your desktop, in your *Documents* folder, or somewhere else you can easily find it.
2. Depending on the browser you're using, the ZIP file might open automatically, or there might be an option to open it at the bottom of the screen. If not, hold down the **Windows Start key** and press **E**. The Windows Explorer window should open. Go to the folder where you saved the ZIP file. Double-click the ZIP file.
3. Click **Extract All** at the top of the window.
4. I recommend that you create a folder called *escape* in your *Documents* folder and extract the files there. My documents folder is *C:\Users\Sean\Documents*, so I just typed *\escape* at the end of the folder name to create a new folder in that folder (see Figure 6). You can use the **Browse** button to get to your *Documents* folder first if necessary.
5. Click **Extract**.

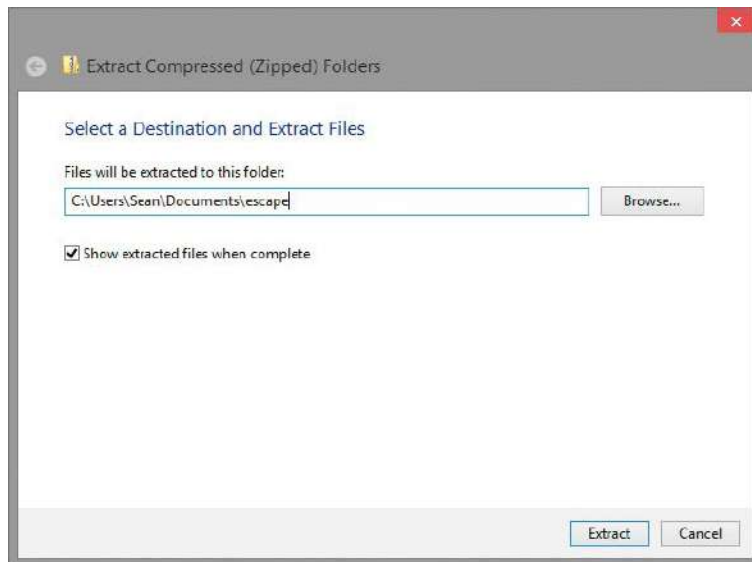


Figure 6: Setting the folder to unzip the game files into

WHAT'S IN THE ZIP FILE

The ZIP file you've just downloaded contains three folders and a Python program, *escape.py* (see Figure 7). The Python program is the final version of the *Escape* game, so you can start playing it right away. The *images* folder contains all the images you'll need for the game and other projects in this book. The *sounds* folder contains the sound effects.

In the *listings* folder, you'll find all the numbered listings in this book. If you can't get a program to work, try my version from this folder. You'll need to copy it from the listings folder first, and then paste it in the *escape* folder where the *escape.py* program is now. The reason you do this is because the program needs to be alongside the *images* and *sounds* folders to work correctly.



Figure 7: The contents of the ZIP file as they might appear in Windows

RUNNING THE GAME

When you downloaded Python, another program called IDLE will have been downloaded with it. IDLE is an integrated development environment (IDE), which is software you can use to write programs in Python. You can run some of the listings in this book from the IDLE Python editor using the instructions provided. Most of the programs, though, use Pygame Zero, and you have to run those programs from the command line. Follow the instructions here to run the *Escape* game and any other Pygame Zero programs.

RUNNING PYGAME ZERO PROGRAMS ON THE RASPBERRY PI

If you're using a Raspberry Pi, follow these steps to run the *Escape* game:

1. Using the File Manager, go to your *escape* folder in your *pi* folder.
2. Click **Tools** on the menu and select **Open Current Folder in Terminal**, or you can press F4. The command line window (also known as the *shell*) should open, as shown in Figure 8. You can enter instructions here for managing files or starting programs.

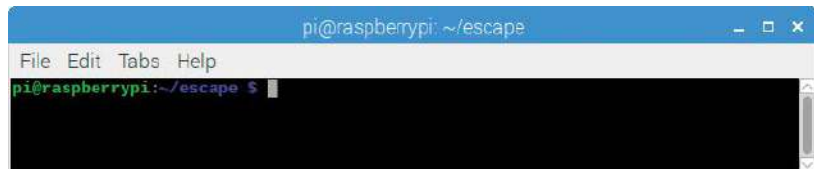


Figure 8: The command line window on the Raspberry Pi

3. Type in the following command and press ENTER. The game begins!

```
pgzrun escape.py
```

This is how you run a Pygame Zero program on the Pi. To run the same program again, repeat the last step. To run a different program that's saved in the same folder, repeat the last step but change the name of the filename after `pgzrun`. To run a Pygame Zero program in a different folder, follow the steps starting from step 1, but open the command line from the folder with the program you want to run.

RUNNING PYGAME ZERO PROGRAMS IN WINDOWS

If you're using Windows, follow these steps to run the program:

1. Go to your *escape* folder. (Hold down the **Windows Start** key and press **E** to open the Windows Explorer again.)
2. Click the long bar above your files, as shown in Figure 9. Type `cmd` into this bar and press ENTER.

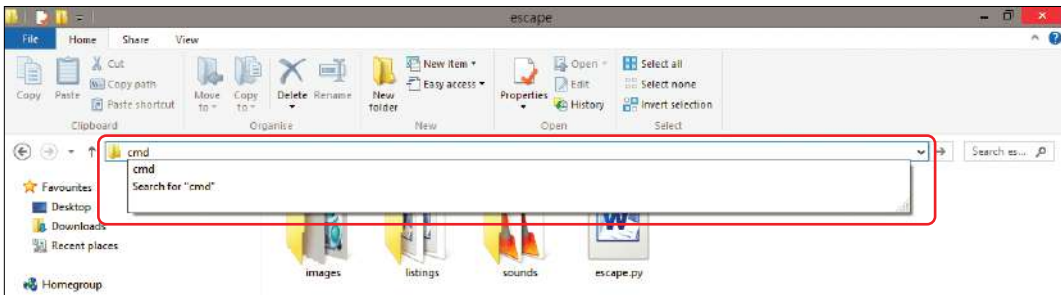


Figure 9: Finding the path to your Pygame files

3. The command line window will open. Your folder named *escape* will appear just before the `>` on the last line, as shown in Figure 10.

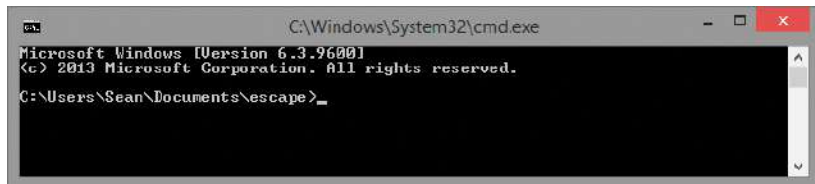


Figure 10: The command line window in Windows

4. Type `pgzrun escape.py` in the command line window. Press ENTER, and the *Escape* game begins.

This is how you run a Pygame Zero program on a Windows computer. You can run the program again by repeating the last step. To run a different program that's saved in the same folder, repeat the last step but change the name of the filename after `pgzrun`. To run a Pygame Zero program in a different folder, follow the steps starting from step 1, but open the command line from the folder with the program you want to run.

PLAYING THE GAME

You're working alone on the space station on Mars, many millions of kilometers from home. The rest of the crew is on a long-distance mission, exploring a canyon for signs of life, and won't be back for days. The murmuring hum of the life support systems surrounds you.

You're startled when the alarm sounds! There's a breach in the space station wall, and your air is slowly venting into the Martian atmosphere. You climb quickly but carefully into your space suit, but the computer tells you the suit is damaged. Your life is at risk.

Your first priority is to repair your suit and ensure a reliable air supply. Your second priority is to radio for help, but the space station's radio systems are malfunctioning. Last night the Poodle lander, sent from Earth, crash-landed in the Martian dust. If you can find it, perhaps you can use its radio to issue a distress signal.

Use the arrow keys to move around the space station. To examine an object, stand on it and press the spacebar. Alternatively, if the object is something you can't walk on, press the spacebar while walking into it.

To pick up an object, walk onto it and press the G key (for *get*).

To select an object in your inventory, shown at the top of the screen (see Figure 11), press the TAB key to move through the items. To drop the selected object, press D.

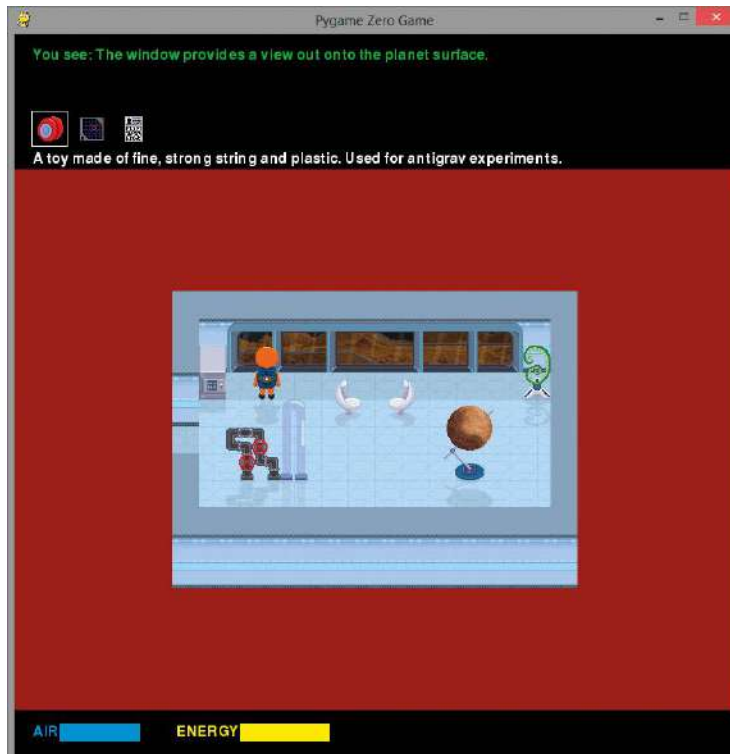


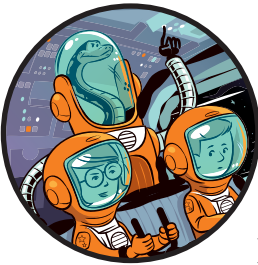
Figure 11: Your adventure begins!

To use an object, either select it in your inventory or walk onto or into it and press U. You can combine objects or use them together when you press U while you carry one object and stand on the other or while you carry one and walk into the other.

You'll need to work out how to use your limited resources creatively to overcome obstacles and get to safety. Good luck!



YOUR FIRST SPACEWALK



Welcome to the space corps. Your mission is to build the first human outpost on Mars. For years, the world's greatest scientists have been sending robots to study it up close. Soon you too will set foot on its dusty surface.

Travel to Mars takes between six and eight months, depending on how Earth and Mars are aligned. During the journey, the spaceship risks hitting meteoroids and other space debris. If any damage occurs, you'll need to put on your spacesuit, go to the airlock, and then step into the void of space to make repairs, similar to the astronaut in Figure 1-1.

In this chapter, you'll go on a spacewalk by using Python to move a character around the screen. You'll launch your first Python program and learn some of the essential Python instructions you'll need to build the space station later in the book. You'll also learn how to create a sense of depth by overlapping images, which will prove essential when we create the *Escape* game in 3D later (starting with our first room mock-up in Chapter 3).



Figure 1-1: NASA astronaut Rick Mastracchio on a 26-minute spacewalk in 2010, as photographed by astronaut Clayton Anderson. The spacewalk outside the International Space Station was one of a series to replace coolant tanks.

If you haven't already installed Python and Pygame Zero (Windows users), see "Installing the Software" on page 3. You'll also need the *Escape* game files in this chapter. "Downloading the Game Files" on page 7 tells you how to download and unzip those files.

STARTING THE PYTHON EDITOR

As I mentioned in the Introduction, in this book we'll use the Python programming language. A programming language provides a way to write instructions for a computer. Our instructions will tell the computer how to do things like react to a keypress or display an image. We'll also be using Pygame Zero, which gives Python some additional instructions for handling sound and images.

Python comes with the IDLE editor, and we'll use the editor to create our Python programs. Because you've already installed Python, IDLE should now be on your computer as well. The following sections explain how to start IDLE, depending on the type of computer you're using.

STARTING IDLE IN WINDOWS 10

To start IDLE in Windows 10, follow these steps:

1. Click the Cortana search box at the bottom of the screen, and enter **Python** in the box.
2. Click **IDLE** to open it.

3. With IDLE running, right-click its icon in the taskbar at the bottom of the screen and pin it. Then you can run it from there in the future using a single click.

STARTING IDLE IN WINDOWS 8

To start IDLE in Windows 8, follow these steps:

1. Move your mouse to the top right of the screen to show the Charms bar.
2. Click the Search icon, and enter **Python** in the box.
3. Click **IDLE** to open it.
4. With IDLE running, right-click its icon in the taskbar at the bottom of the screen and pin it. Then you can run it from there in the future using a single click.

STARTING IDLE ON THE RASPBERRY PI

To start IDLE on the Raspberry Pi, follow these steps:

1. Click the Programs menu at the top left of the screen.
2. Find the Programming category.
3. Click the Python 3 (IDLE) icon. The Raspberry Pi has both Python 2 and Python 3 installed, but most of the programs in this book will work only in Python 3.

INTRODUCING THE PYTHON SHELL

When you start IDLE, you should see the Python *shell*, as shown in Figure 1-2. This window is where you can give Python instructions and immediately see the computer respond. The three arrows (`>>>`) are called a *prompt*. They tell you that Python is ready for you to enter an instruction.

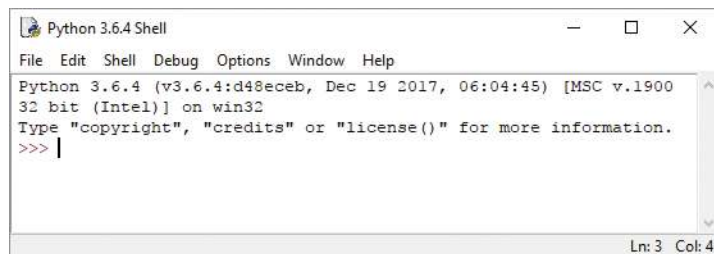


Figure 1-2: The Python shell

So let's give Python something to do!

DISPLAYING TEXT

For our first instruction, let's tell Python to display text on the screen. Type the following line and press ENTER:

```
>>> print("Prepare for launch!")
```

As you type, the color of your text will change. It starts off black, but as soon as Python recognizes a command, like `print`, the text changes color.

Figure 1-3 shows the names of the different parts of the instruction you just entered. The purple word `print` is the name of a *built-in function*, which is one of many instructions that are always available in Python. The `print()` function displays onscreen the information you place between the *parentheses* (curved brackets). The information between a function's parentheses is the function's *argument*.

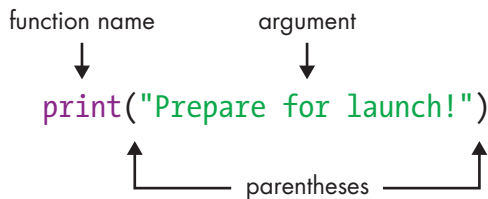


Figure 1-3: The different parts of your first instruction

In our first instruction, the `print()` function's argument is a *string*, which is what programmers call a piece of text. (A string can include numbers, but they're treated as letters, so you can't do calculations with numbers in a string.) The double quotation marks (" ") show the start and end of the string. Anything you type between double quotation marks will be green, and so will the quotation marks.

The colors do more than brighten up the screen: they highlight the different parts of the instruction to help you find mistakes. For example, if your final parenthesis is green, it means you forgot the closing double quote on the string.

If you entered the instruction correctly, your computer will display this text:

Prepare for launch!

The string that was shown in green is now displayed onscreen in blue. All *output* (information the computer gives to you) appears in blue. If your command didn't work, check that you did the following:

1. Spelled `print` correctly. If you did, it will be purple (see Figure 1-3).
2. Used two parentheses. Other bracket shapes won't work.

- Used two double quotes. Don't use two apostrophes (') instead of a double quote ("). Although the double quote includes two marks, it's just one symbol on the keyboard. On a US keyboard, the double quote is in the middle row of letters, on the right, and must be used with the SHIFT key. On a UK keyboard, the double quote is on the 2 key.

If you make a mistake typing the text between the double quotes, the instruction will still work, but the computer will display exactly what you typed. For example, try this:

```
>>> print("Prepare for lunch!")
```

It doesn't matter if you mistype the string now, but be careful when you type a string or an instruction later in the book. Mistakes often prevent a program from working correctly, and it can be hard to track down a mistake in a longer program, even with the color coding.

TRAINING MISSION #1

Can you enter a new instruction to output your name? (You'll find the answers to the Training Missions in the "Mission Debrief" section at the end of each chapter.)

OUTPUTTING AND USING NUMBERS

So far you've used the `print()` function to output a string, but it can also do calculations and output a number. Enter the following line:

```
>>> print(4 + 1)
```

The computer should output the number 5, the solution to `4 + 1`. Unlike with a string, you don't use quotes around numbers and calculations. But you still use the parentheses to mark the start and end of the information you want to give the `print()` function.

What happens if you do put quotes around `4 + 1`? Try it! The result is that the computer outputs `"4 + 1"` because it doesn't treat 4 and 1 as numbers. Instead, it treats the argument as a string. You ask it to output `"4 + 1"`, and it does exactly that!

```
>>> print(4 + 1)
5
>>> print("4 + 1")
4 + 1
```

Python does the calculation only when you don't include the quotes. You'll use the `print()` function a lot in your programs.

INTRODUCING SCRIPT MODE

The shell is great for quick calculations and for short instructions. But for longer sets of instructions, like games, it's much easier to create programs instead. *Programs* are repeatable sets of instructions that we save so we can run them whenever we want and change them whenever we need to without retyping them. We'll build programs using IDLE's *script mode*. When you enter instructions in script mode, they don't run immediately as they do in the shell.

Using the menu at the top of the shell window, select **File** and then select **New File** to open a blank new window, as shown in Figure 1-4. The title bar at the top of the window displays *Untitled* until you save your file and name it. Once you've saved your file, the title bar will display the file's name. From now on, we'll use script mode nearly all the time when we're creating Python code.

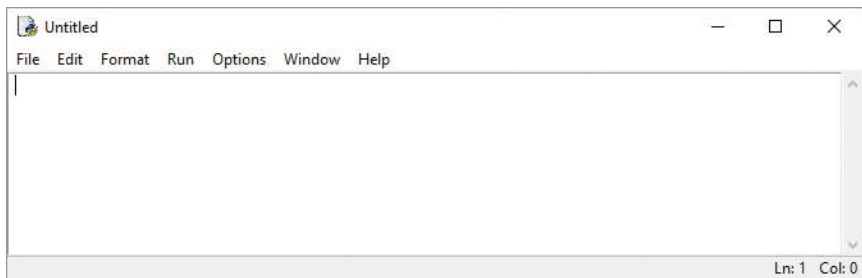


Figure 1-4: Python script mode

When you enter instructions in script mode, you can change, add, and delete instructions using the mouse or the arrow keys, so it's much easier to fix mistakes and build your programs. Starting from Chapter 4, we'll build the *Escape* game by adding to it piece-by-piece in script mode and testing each new section as we go.

TIP

If you're not sure whether you're in the shell or the script mode window, look at the title bar at the top. The shell displays *Python Shell*. The script mode window displays either *Untitled* or the name of your program.

CREATING THE STARFIELD

The first program we'll write will display the starfield image that we'll use as the space background for our *Spacewalk* program. This image is in the *images* folder within the *escape* folder. Start by entering Listing 1-1 into the new blank window in IDLE.

NOTE In this book, I'll use numbers in circles (like this: ❶) to refer to different bits of code in the explanations so it's easier for you to follow along. Don't type these numbers in your program. When you see a number in a circle in the text, refer back to the program listing to see which part of the program I'm talking about.

Listing 1-1 is a short program, but there are a couple of details that you should pay attention to while you're typing: the `def` statement ❷ needs a colon at the end of its line, and the next line ❸ needs to start with four spaces. When you add the colon to the end of the `def` line and press ENTER, IDLE automatically adds the four spaces at the beginning of the next line for you.

```
❶ # Spacewalk
❷ # by Sean McManus
❸ # www.sean.co.uk / www.nostarch.com

❹ WIDTH = 800
❺ HEIGHT = 600
❻ player_x = 600
   player_y = 350

❼ def draw():
❺     screen.blit(images.backdrop, (0, 0))
```

Listing 1-1: See the starfield in Pygame Zero.

Select the **File** menu at the top of the screen and then select **Save** (from now on, we'll use a shorthand for menu selections that looks like this: **File ▶ Save**). In the Save dialog, name your program *listing1-1.py*. You need to save your file in the *escape* folder you set up in the Introduction. This way, it's in the same folder as the book's *images* folder, and Pygame Zero can find the images when you run the program. After you save the file, your *escape* folder should now contain your *listing1-1.py* file and the *images* folder, as shown in Figure 1-5 (along with the *listings* and *sounds* folders).

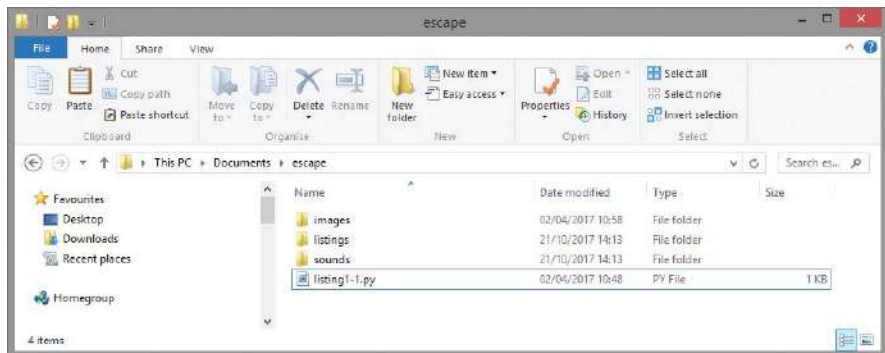


Figure 1-5: Your new Python program and the images folder should be stored in the same place.

I'll explain how the *listing1-1.py* program works shortly, but first let's run the program so we can admire the starfield. The program needs some instructions from Pygame Zero to manage the images, so to use those instructions, we need to run the program using a `pgzrun` instruction. Whenever we use any instructions from Pygame Zero in a Python program, we need to run it using `pgzrun`.

We'll type this on the computer's command line, just like we did in the Introduction to run the *Escape* game. First, look back at "Running the Game" on page 9, and follow the directions there to open your computer's command line terminal from your *escape* folder. Then run the following instruction from the command line:

```
pgzrun listing1-1.py
```

RED ALERT

Don't type this instruction in IDLE: be sure to type it in your Windows or Raspberry Pi command line. The Introduction shows you how.

If all went according to plan, you should be looking at the majesty of space, as shown in Figure 1-6.

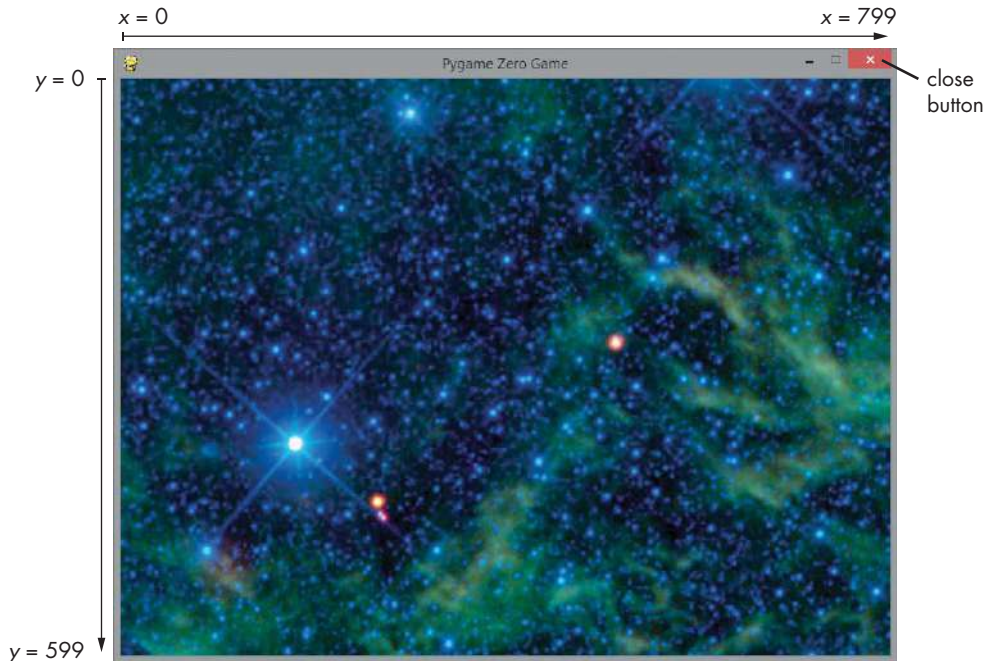


Figure 1-6: The starfield. The starfield image is courtesy of NASA/JPL-Caltech/UCLA and shows star cluster NGC 2259.

USING MY EXAMPLE LISTINGS

If you can't get a program in this book to work, you can use my example program instead. For instance, you can use my *listing1-1.py* example and modify it to make your own *listing1-2.py* shortly so you can continue following along.

You'll find my programs in the *listings* folder, which is in the *escape* folder. Simply open the *listings* folder in Windows or the Raspberry Pi desktop, find the listing you need, copy it, and then paste it into the *escape* folder. Then open the copied listing in IDLE and follow along with the next step in the book. When you look at the folder, you should be able to see your Python file and the *images* folder are in the same place (see Figure 1-5).

UNDERSTANDING THE PROGRAM SO FAR

Most of the instructions you'll see in this book will work in any Python program. The `print()` function, for example, is always available. To make the programs in this book, we're also using Pygame Zero. This adds some new functions and capabilities to Python for creating games, especially for the screen display and sound. Listing 1-1 introduces our first instructions from Pygame Zero, used to set up the game window and draw the starfield.

Let's take a closer look at how the *listing1-1.py* program works.

The first few program lines are *comments* ❶. When you use a `#` symbol, Python ignores everything after it on the same line, and the line appears in red. The comments help you and other people reading the program understand what a program does and how it works.

Next, the program needs to store some information. Programs almost always need to store information that the program uses or needs to refer back to at a later time. For example, in many games, the computer needs to keep track of the score and the player's position on the screen. Because these details can change (or *vary*) as the program runs, they're stored in something called a *variable*. A variable is a name you give to a piece of information, either a number or some text.

To create a variable, you use an instruction like this:

```
variable_name = value
```

NOTE Code terms shown in italics are placeholders that would be filled in. Instead of *variable_name*, you would enter your own variable name.

For example, the following instruction puts the number 500 into the variable *score*:

```
score = 500
```

You can name your variables almost anything you want. However, to make your program easy to write and understand, you should choose

variable names that describe the information inside each variable. Note that you can't use names for your variables that Python uses for its language, such as `print`.

RED ALERT

Python is case-sensitive, which means it is strict about whether variables use uppercase or lowercase letters. In fact, it treats `score`, `SCORE`, and `Score` as three completely different variables. Make sure you copy my example programs exactly, or they might not work properly.

Listing 1-1 begins by creating some variables. Pygame Zero uses the `WIDTH` and `HEIGHT` variables ❷ to set the size of the game window on the screen. Our window is wider than it is tall because the `WIDTH` value (800) is bigger than the `HEIGHT` value (600).

Notice that we've spelled these variables with capital letters. The capital letters in variable names tell us that they're *constants*. A constant is a particular kind of variable with values that aren't supposed to change after they've been set up. The capital letters help other programmers who are looking at the program understand that they shouldn't let anything else in the program change these variables.

The `player_x` and `player_y` variables ❸ will store your position on the screen as you carry out your spacewalk. Later in the chapter, we'll use these variables to draw you on the screen.

We then define a function using the `def()` statement ❹. A *function* is a group of instructions you can run whenever you need them in your program. You've already seen one built-in function called `print()`. We'll make our own function in this program called `draw()`. Pygame Zero will use it to draw the screen display whenever the screen changes.

We define a function using the keyword `def` ❺, followed by the function name we choose, empty parentheses, and a colon. Sometimes you'll use a function's parentheses to contain information for that function, as you'll see later in this book.

We then need to give the function instructions for what it should do. To tell Python which instructions belong to the function, we indent them by four spaces. The `screen.blit()` instruction ❻ from Pygame Zero draws an image on the screen. In the parentheses, we tell it which image to draw and where to draw it, like this:

```
screen.blit(images.image_name, (x, y) )
```

From the *images* folder, we'll use the *backdrop.jpg* file, which is the starfield. In our *listing1-1.py* program, we refer to it as `images.backdrop`. We don't have to use the file's *.jpg* extension, because we're using Pygame Zero to handle the images, and Pygame Zero doesn't require the extension. Also, the program knows where the image is because all the images must be in the *images* folder so Pygame Zero can find them.

We put the image on the screen at position (0, 0) ❼, which is the top-left corner of the screen. The first number, known as the *x position*, tells the `screen.blit()` instruction how far from the left edge we want our image to

be; the second number, known as the *y position*, describes how far down we want it to be. The *x* positions go from 0 on the left edge of the window to 799 on the right edge because our window is 800 pixels wide. Similarly, the *y* positions run from 0 at the top of the window to 599 at the bottom (see Figure 1-6).

For positions onscreen, we use a *tuple*, which is just a group of numbers or strings in parentheses, such as (0, 0). In a tuple, the numbers are separated with a comma, plus an optional space for readability.

The most important thing you need to know about tuples is that you have to take care with the punctuation. Because the tuple uses parentheses, and we put this tuple inside the parentheses for `screen.blit()`, there are two sets of parentheses here. So you need parentheses around the tuple values, but you also need to close the parentheses for `screen.blit()` after the tuple.

STOPPING YOUR PYGAME ZERO PROGRAM

Similar to space, your Pygame Zero program will go on forever. To stop it, click the game window's close button at the top right (see Figure 1-6). You can also close the program from the command line window where you entered the `pgzrun` instruction by pressing CTRL-C.

RED ALERT

Don't close the command line window itself. Otherwise, you'll have to open it again to run another Pygame Zero program. If you do close it by mistake, refer back to "Running the Game" on page 9 to open it again.

ADDING THE PLANET AND SPACESHIP

Let's bring Mars and the spaceship into view. In IDLE, add the last two lines in Listing 1-2 to your existing *listing 1-1.py* program.

NOTE *I'll use `--snip--` in code listings to show you where I've left out some code, usually because the code is repeated from before. I'll also show any repeated code in gray so you can see the new code you need to add more clearly. Don't add in the repeated code again!*

In the following code, I've excluded the comments and variable setup to save space and make it easier for you to see the new code. But make sure you keep those instructions in your program. Just add the two new lines at the end.

listing1-2.py

```
--snip--
def draw():
    screen.blit(images.backdrop, (0, 0))
    screen.blit(images.mars, (50, 50))
    screen.blit(images.ship, (130, 150))
```

Listing 1-2: Adding Mars and the ship

Save your updated program as *listing1-2.py* by selecting **File ▶ Save As**. Run your program by switching back to the command line window and entering the command `pgzrun listing1-2.py`. Figure 1-7 shows how the screen should now look, with the red planet and the spaceship above it.

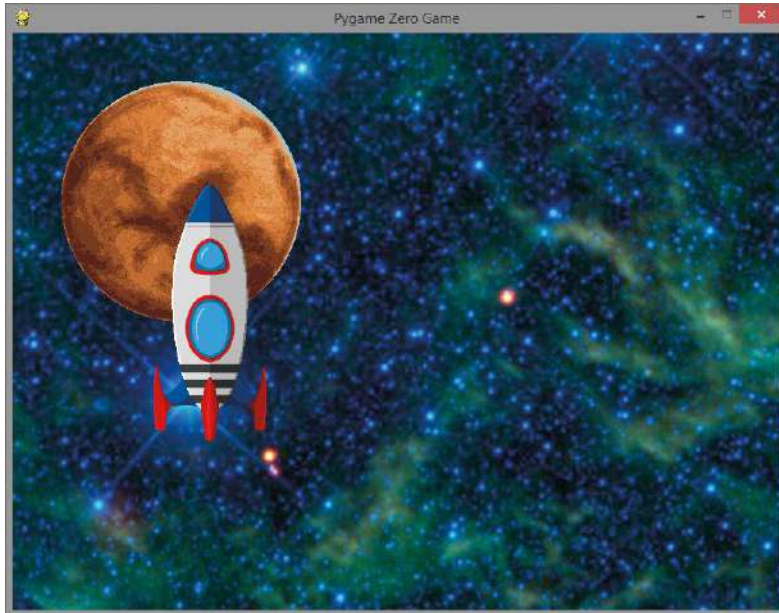


Figure 1-7: Mars and the spaceship. The Mars image was taken by the Hubble Space Telescope in 1991.

NOTE *If your program doesn't work as expected, check that all your `screen.blit()` instructions have exactly four spaces before them and are lined up with each other.*

The first of the new instructions places the image *mars.jpg* at the position (50, 50), which is near the top-left corner of the screen. The second new instruction positions the ship at (130, 150). In each case, the coordinates used are for the top-left corner of the image.

CHANGING PERSPECTIVE: FLYING BEHIND THE PLANET

Now let's look at how we can make the ship fly behind the planet. Swap the order of the last two instructions in IDLE, as shown in Listing 1-3. To do this, highlight one of the lines, press CTRL-X to cut it, click on a new line, and press CTRL-V to paste it in place. You can also use the cut and paste options in the Edit menu at the top of the screen.

listing1-3.py

```
--snip--
def draw():
    screen.blit(images.backdrop, (0, 0))
```



```
screen.blit(images.ship, (130, 150))
screen.blit(images.mars, (50, 50))
```

Listing 1-3: Swapping the order of the planet and ship instructions

If the previous version of your program is still running, close it now. Save your new program as *listing1-3.py* and run it from the command line by entering **pgzrun listing1-3.py**. You should see that the spaceship is now behind the planet, as shown in Figure 1-8. If not, make sure you ran the right file (*listing1-3.py*), and then check that the instructions in the program are correct.

The ship goes behind the planet because the images are added to the screen in the order they are drawn in the program. In our updated program, we draw the starfield, draw the ship, and then draw Mars. Each new image appears on top of the previous one. If two images overlap, the image that was drawn last appears in front of the one drawn earlier.

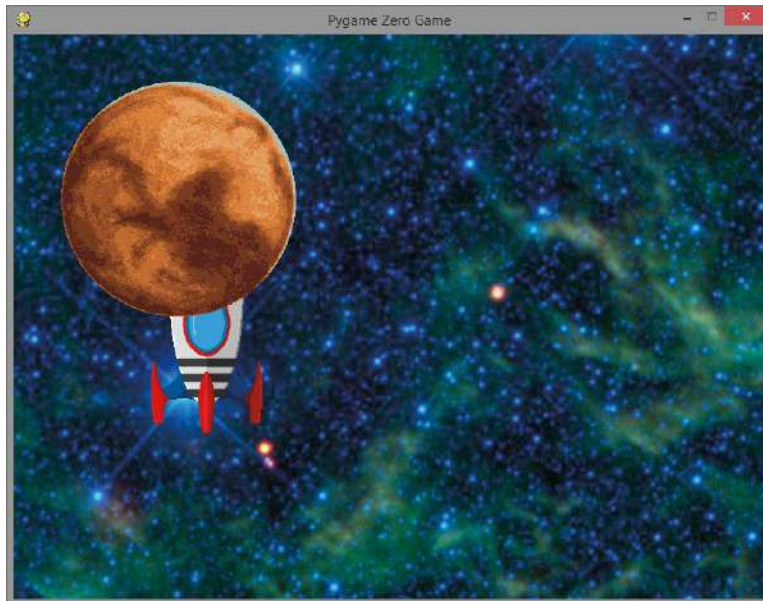


Figure 1-8: The spaceship is now behind the planet.

TRAINING MISSION #2

Can you move just one drawing instruction in your program to make the planet and the spaceship disappear? If you're not sure what to do, experiment by moving the drawing instructions to see what effect it has when you save the program and run it again.

Make sure you keep the drawing instructions aligned and indented with four spaces inside the `draw()` function. When you're done experimenting, match the instructions in Listing 1-3 again to bring the ship and Mars back into view.

SPACEWALKING!

It's time to climb out of the underside of the spaceship and begin your spacewalk. Edit your program so it matches Listing 1-4. But be sure to keep the variable instructions that aren't shown here the same as they were before. Save the updated program as *listing1-4.py*.

listing1-4.py

```
--snip--
def draw():
    screen.blit(images.backdrop, (0, 0))
    screen.blit(images.mars, (50, 50))
❶ screen.blit(images.astronaut, (player_x, player_y))
❷ screen.blit(images.ship, (550, 300))

❸ def game_loop():
❹     global player_x, player_y
❺     if keyboard.right:
❻         player_x += 5
❼         elif keyboard.left:
❺             player_x -= 5
❸         elif keyboard.up:
❺             player_y -= 5
❺         elif keyboard.down:
❺             player_y += 5

❹ clock.schedule_interval(game_loop, 0.03)
```

Listing 1-4: Adding the spacewalk instructions

In this listing, we add a new instruction ❶ to draw the astronaut image at the position in the `player_x` and `player_y` variables, which were set up at the start of the program in Listing 1-1. As you can see, we can use these variable names in place of numbers for the astronaut's position. The program will use the current numbers stored in these variables to figure out where to put the astronaut every time it is drawn.

Note that the order of drawing the images has changed in the program and is now backdrop, Mars, astronaut, and ship. Make sure you change the order of your `screen.blit()` instructions to match this listing.

The astronaut starts off overlapping the ship. Because the astronaut is drawn before the ship, the astronaut will appear to emerge from underneath (behind) the spaceship. We also changed the position of the ship ❷ to the bottom-right area of the screen. This gives the astronaut space to fly toward the planet.

Run the program by entering **pgzrun listing1-4.py**. You should now be able to use the arrow keys to move freely through space, protected by your spacesuit, as shown in Figure 1-9. You'll see that you fly behind the spaceship but in front of Mars and the starfield. The order in which we draw the images creates a simple illusion of depth. When we draw the space station beginning in Chapter 3, we'll use this drawing technique to create a 3D perspective of each room. We'll draw the rooms from back to front to create a sense of depth.

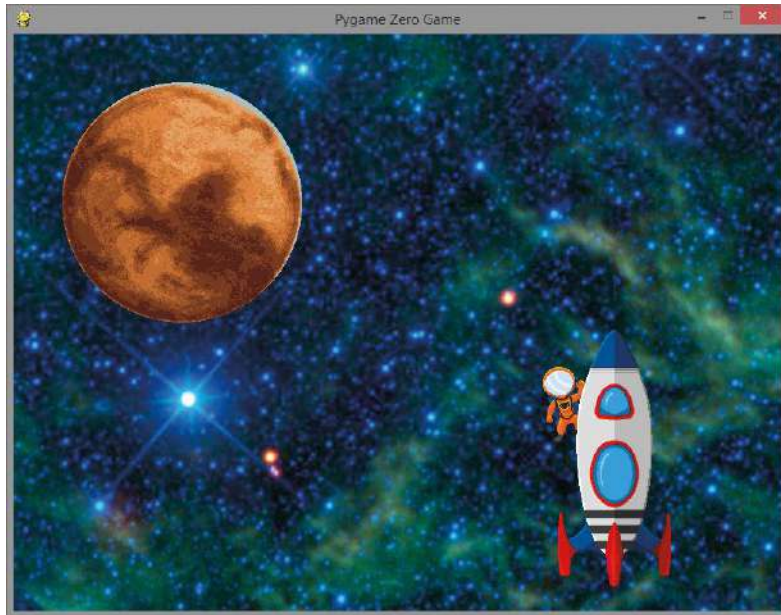


Figure 1-9: You emerge from the ship for your spacewalk.

TRAINING MISSION #3

Can you edit the code to move the spaceship and the astronaut to the top-right corner of the screen? You'll need to change the starting values for `player_x` and `player_y`, as well as where the spaceship is drawn. Make sure the player is "inside" (actually underneath) the ship at the start of the program. Experiment with other positions, too. This is a great way to get familiar with screen positions. Refer back to Figure 1-6 if you need to.

UNDERSTANDING THE SPACEWALK LISTING

The spacewalk listing, Listing 1-4, is interesting because it lets you control part of the program from the keyboard, which will be crucial in the *Escape* game. Let's look at how our final spacewalk program works.

We build on our earlier listings and add a new function called `game_loop()` ❸. This function's job is to change the values of the `player_x` and `player_y` variables when you press the arrow keys. Changing the variables enables you to move the astronaut character because those variables position the astronaut when it's drawn.

Before we go on, we need to look at two different types of variables. Variables that are changed inside a function usually belong to that function and can't be used by other functions. They're called *local variables*, and they make it harder for bits of the program to interfere with other bits accidentally and cause errors.

But in the spacewalk listing, we need both the `draw()` and `game_loop()` functions to use the same `player_x` and `player_y` variables, so they need to be *global variables*, which any part of the program can use. We set up global variables at the start of the program, outside of any functions.

To tell Python that the `game_loop()` function needs to use and change the global variables we set up outside of this function, we use the `global` command ❹. We put it at the beginning of the function and list the variables we want to use as global variables. Doing this is like overriding the safety feature that stops you from changing variables that weren't created inside the function. We don't need to use `global` in the `draw()` function, because the `draw()` function doesn't need to change those variables. It only needs to look at what those variables contain.

We tell the program to use keyboard controls using the `if` command. With this instruction, we tell Python to do something *only if* certain conditions are met. We use four spaces to indent the instructions that belong to the `if` command. That means these instructions are indented by eight spaces in total in Listing 1-4 because they are also inside the `game_loop()` function. These instructions run only if the statement after the `if` command is true. If not, the instructions that belong to the `if` command are skipped over.

It might seem odd to use spaces like this to show which instructions belong together, especially if you've used other programming languages, but it makes the programs easy to read. Other languages often need brackets around sets of instructions like this. Python keeps it simple.

We use the `if` command to check whether the right arrow key is pressed ❺. If it is, we change the value of `player_x` by adding 5 ❻, moving the astronaut image to the right. The symbols `+=` mean *increase by*, so the following line increases the number in the `player_x` variable by 5:

```
player_x += 5
```

Similarly, `-=` means *decrease by*, so the following instruction reduces the number in `player_x` by 5:

```
player_x -= 5
```

If the right arrow key is not pressed, we check whether the left key is pressed. If it is, the program subtracts 5 from the `player_x` value, moving the astronaut's position left. To do that, we use an `elif` command ❼, which is short for "else if." You can think of *else* as meaning *otherwise* here. In plain English, this part of our program means, "If the right arrow key is pressed, add 5 to the *x* position. Otherwise, if the left key is pressed, subtract 5 from the *x* position." We then use `elif` to check for up and down keypresses in the same way, and change the *y* position to move the astronaut up or down. The `draw()` function uses the `player_x` and `player_y` variables for the astronaut's position, so changing the numbers in these variables makes the astronaut move on the screen.

TIP

If you change the `elif` command at ⑧ to an `if` command, the program allows you to move up or down at the same time as moving left or right, letting you walk diagonally. That's fun in the spacewalk program, but we'll use code similar to this to move around the space station later, and it doesn't look natural there.

The final instruction ⑨ sets the `game_loop()` function to run every 0.03 seconds using the `clock` in Pygame Zero, so the program keeps checking for your keypresses and changing your position variables frequently. Note that you don't put any parentheses after `game_loop` here. This instruction isn't indented, because it doesn't belong to any function. When the program starts, it runs the instructions that aren't in any function in the order they are in the listing, from top to bottom. Therefore, the last line of the program is one of the first to run after the variables are set up. This last line starts the `game_loop()` function running.

The `draw()` function runs automatically whenever the screen needs updating. This is a feature of Pygame Zero.

TRAINING MISSION #4

Let's fit some new thrusters to the spacesuit. Can you work out how to make the astronaut move faster in the up and down directions than it does in the left and right directions? Each keypress in the up or down direction should make the space suit move more than a keypress in the left or right direction.

Enjoy the breathtaking views as you take your spacewalk and conduct any essential repairs to your ship. We'll reconvene in Chapter 2, where you'll learn some procedures that will help you stay safe in space.

ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter. If you're not sure about something, flip back through the chapter and give the topic another look.

- ☐ You use IDLE's script mode to create a program that you can save, edit, and run again. Enter script mode by selecting **File ▶ New File** or edit an existing file by selecting **File ▶ Open**.
- ☐ Strings are pieces of text in code. Double quotes mark the start and end of a string. A string can include numbers, but they're treated as letters.
- ☐ Variables store information, either numbers or strings.
- ☐ The `print()` function outputs information on the screen. You can use it for strings, numbers, calculations, or the values of variables.

- ☐ The # symbol in a program marks a comment. Python ignores anything on the same line after a #, and comments can be a handy reminder for you and anyone you share your code with.
- ☐ Use the WIDTH and HEIGHT variables to set the size of your game window.
- ☐ To run a Pygame Zero program, open the command line from the folder your Python program is in, and then enter `pgzrun filename.py` in the command line to run it.
- ☐ A function is a group of instructions you can run whenever you want your program to use the instructions. Pygame Zero uses the `draw()` function to draw or update the game screen.
- ☐ Use `screen.blit(images.image_name, (x, y))` to draw an image at position (x, y) on the screen. The x- and y-axes are numbered starting at 0 in the top-left corner.
- ☐ A *tuple* is a group of numbers or strings in parentheses, separated by a comma. The contents of a tuple can't be changed by the program after they've been set up.
- ☐ To end your Pygame Zero program, click the window's close button or press CTRL-C in the command line window.
- ☐ If images overlap, the image you drew last in the program appears at the front.
- ☐ The `elif` command is short for "else if." Use it to combine `if` conditions so that only one set of instructions can run. In our program, we use it to stop the player from moving in two directions at the same time.
- ☐ If we want to change a variable inside a function and use it in a different function, we need to use a *global variable*. We set it up outside of the functions and use the `global` keyword inside a function when we plan to change the variable there.
- ☐ We can set a function to run at regular intervals using the clock feature in Pygame Zero.

MISSION DEBRIEF

Here are the answers for the training missions in this chapter.

TRAINING MISSION #1

This answer will vary, depending on your name, but it should look something like this:

```
>>> print("Neil Armstrong")
```

TRAINING MISSION #2

If you draw the starfield last, it will hide the planet and the spaceship. Cunning!
Place the images in this order:

```
--snip--
def draw():
    screen.blit(images.mars, (50, 50))
    screen.blit(images.ship, (130, 150))
    screen.blit(images.backdrop, (0, 0))
```

TRAINING MISSION #3

Change the value of `player_y` at the start of the program from 350 to a lower number, such as 150. Change the second number in the tuple for the `screen.blit()` instruction for the ship image to a lower number, such as 50. Other numbers will also work as long as the ship is in the top right and the astronaut starts behind the ship.

TRAINING MISSION #4

To make the player move faster up and down than left and right, change how much the `player_y` variable changes by each time the key is pressed. If you change the fives to a higher number, the player will move a greater distance up or down the screen for each up or down keypress. As a result, the astronaut will appear to move faster. But if you make the value too high, the illusion of animation will be lost, and the suit will seem to just teleport through space. Experiment with a few values to see what works.

```
--snip--
    elif keyboard.up:
        player_y -= 15
    elif keyboard.down:
        player_y += 15
--snip--
```

INDEX

SYMBOLS AND NUMBERS

- ' (apostrophe), 17
- \ (backslash), 94, 177
- #\ in *Escape*, 212
- : (colon), 19, 251
- , (comma), 84, 252
- # (for comments), 21, 63–64
 - case sensitivity, of comments, 251
 - commenting to turn code off, 101, 107, 112, 212, 213
 - uncommenting to turn code on, 147, 207, 212
- { } (curly brackets), 80, 251
- " (double quotation mark), 17, 29, 251
- // (floor division operator), 163, 170
- % (modulo operator), 71, 163, 170, 206
- * (multiplication), 56, 70
- != (not-equal-to operator), 71, 103
- = operator, 28
- + operator
 - for adding numbers, 17
 - for combining lists, 40
 - for combining strings, 65
- += operator
 - for adding numbers, 28
 - for combining or extending lists, 41, 64, 103
- () (parentheses), 16, 22, 23, 37, 65, 135, 251
- ; (semicolon), 251
- [] (square brackets), 34, 36, 37, 65, 251
- 1 (as index number), 104, 190
- 3D effect, 25–26, 30
- 3D room display, 53, 72
- 255, in `room_map`, 106–107, 108, 109, 156, 157, 169

A

- access cards, 86, 184, 185–187, 190, 196
- ACCESS_DICTIONARY, 187, 196
- add(), 128–129
- adding numbers, 17, 128–129
- add_object(), 159–160
- adjust_wall_transparency(), 143–145
- air, 174, 198–202
- air_countdown(), 199–200
- air_fixed, 174, 235, 236, 237
- airlock_door_frame, 194
- AIR section, 198–200
- alarm(), 199, 201, 202
- Anderson, Clayton, 14
- animation
 - airlock door, 194
 - astronaut, 113–114, 116, 119–122
 - doors, 184–196
 - front wall, 142–145
 - game completion, 180
- apostrophe ('), 17
- append(), 35, 45, 65, 160, 208
- arguments, 16, 128, 129, 148, 156
- arrow keys
 - in *Escape*, 116, 119–120
 - in *Explorer*, 74, 91, 108
 - in *Spacewalk*, 26, 27, 28
- assert, 65, 101, 109, 110, 154, 246
- astronaut names, changing, 64, 215

B

- backslash (\), 94, 177
- black space, under objects, 91, 109, 127, 139
- Boolean values, 61, 71
- bottom_edge, 70
- bottom edge type, 69

- box Rect, 138
- brackets, differences between, 65, 251
- bugs. *See* errors
- built-in functions, 16

C

- cabinets, 156, 169
- calculations, 17
- calling a function, 157
- case sensitivity, 22, 82, 251
- centering the room display, 141
- checksum, 101, 109–110
 - turning off for props, 154
- clearing
 - game arena, 138
 - text area, 146
- clipping area, 138–139, 141
- clock, 29, 30, 74, 119, 163
- clock.schedule_interval(), 119, 163
- clock.schedule_unique(), 163
- close button, 23
- close_door(), 188
- clues, 2, 86, 173, 181
- cmd, 5, 10
- collision detection, 120, 212
- colon (:), 19, 251
- color coding, 16, 21, 88, 252
- colors, in Pygame Zero, 131–132, 138, 149
- combining lists, 40–41
- combining objects, 177–179
- command line window, 23, 75
 - on Raspberry Pi, 9
 - on Windows, 5–6, 10
- comma (,), 84, 252
- comments. *See* # (for comments)
- constants, 22
- continuing code on next line, 94, 177
- controls. *See* keyboard controls
- converting
 - decimal numbers to integers, 71
 - numbers to strings, 74, 94
- coordinates, 43, 45, 52, 56
- corridors, 75, 77

- cupboards, 169
- curly brackets, {}, 80, 251
- current_room, 72, 74, 109, 213
- current_room_hazards_list, 206
- curved brackets (parentheses), 16, 22, 23, 37, 65, 135, 251
- customizing the game, 2, 215–216
 - difficulty, 187, 202, 210
 - doors, 184, 187, 195
 - game map, 76, 104, 139
 - images, 215
 - props, 154, 169
 - room designs, 89, 96, 101, 107, 109
 - sharing your customizations, 216

D

- debugging, 65, 88, 249
- decimal numbers, 71, 107
- def statement, 19, 22, 250, 251
- delays, 160, 163
- deleting a list item, 37
- del statement, 37
- DEMO_OBJECTS, 55, 64
- deplete_energy(), 205, 210, 216
- diagonal movement, 29, 120
- dice example, 156
- dictionaries, 80, 95
 - as arguments, 129
 - checking keys, 82
 - compared to lists, 80
 - containing lists, 83–84, 95
 - creating, 80
 - errors, 82–83, 95, 251
 - keys, 80, 95
 - order of items, 82
 - using a variable as a key, 81
 - values, 80
- difficulty of game, adjusting, 187, 202, 210
- displaying numbers, 17
- displaying text. *See* print()
- display_inventory(), 160, 161, 162, 165, 167
- DISPLAY section, 133, 136, 143, 145, 148, 210

division

- // operator, 163, 170
- calculating remainder, 71, 163, 170, 206
- do_door_animation(), 188, 189
- door_in_room_26(), 194, 195
- door_object_number, 188
- doors, 86, 98, 180, 184–196.
 - See also* exits
 - airlock (room 26), 193, 194
 - animation, 188, 189, 193, 196
 - closing, 187, 193
 - in customized map, 76
 - data, 185
 - opening, 185–190, 193–194
 - positioning, 184, 185
 - removing from game, 195–196
 - setting up in props
 - dictionary, 153
 - testing, 190, 193
 - timed, 185, 186, 190, 193, 196
- DOORS section, 187, 189, 191, 193
- double quotation mark ("), 17, 29, 251
- downloading game files, 7, 21
- draw()
 - 3D room, 55
 - final code for *Escape* game, 132, 136–139, 142
 - hazards, 210
 - in *Spacewalk*, 22, 25
- draw_energy_air(), 199, 200
- draw_image(), 135
- drawing
 - filled rectangles, 138, 149
 - images, 135
 - player, 135
 - room, 55, 136–139
 - scenery, 139
 - shadows, 135, 139, 140
 - text, 146, 201
- draw_player(), 135
- draw_shadow(), 135
- drones. *See* hazards
- drop_object(), 165, 166, 167
- dropping objects, 11, 166
- drop shadow (text effect), 201

E

- edge type, 69
- elif command, 28, 30, 120
- else command, 250, 251
- end_the_game(), 199, 200, 201
- energy, 174
 - drawing indicator bar, 199
 - reducing, 205
 - restoring, 198
 - variable, 199
- energy balls. *See* hazards
- engineering bay, 185, 186, 190, 193
- errors, 249. *See also* debugging
 - error message, 173
 - not defined, 251
 - without error message, 250
- escape folder, 7, 8
- Escape* game, 1, 8
 - building, 2
 - compatibility, 6
 - complete code listing, 217
 - customizing. *See* customizing the game
 - downloading files, 7, 21
 - playing, 2, 11
 - running, 9
 - sections in program listing, 63
- escape.zip, 7, 8
- examine_object(), 165, 168
- examining objects in the game, 11, 156, 165, 168
- example listings, 21
- exits, 61, 62, 68, 71. *See also* movement: between rooms
 - adding to room_map, 71
 - in customized map, 76
 - in game map, 60
 - testing from both sides, 75
- Explorer*, 72–74, 76, 91, 97, 107–108
- EXPLORER section, 72–74, 89, 115
 - deleting, 132
 - disabling keyboard controls, 112
 - drawbacks, 127
 - modifying to show room design, 89

F

- False, 61, 83, 251
- fanfare, adding to game, 214
- fences, 102
- File ▶ Save, 19
- find and replace, 147, 207
- find_object_start_x(), 158
- floating-point numbers, 71, 107
- floor, 68, 70, 74
- floor division operator (`//`), 163, 170
- floor pad, 139
- floor type, 69
- floor_type, 70
- for command, 49, 50, 58, 103, 250, 251
- frames list, 194
- FRIEND1_NAME, 64, 65
- FRIEND2_NAME, 64, 65
- from_player_x, 120
- from_player_y, 120
- functions, 16, 22, 30, 251. *See also*
 - arguments
 - built-in, 16
 - calling, 157
 - defining, 19
 - receiving information in, 128, 129, 148
 - returning information from, 156, 170
 - sending information to, 128–129, 148

G

- game. *See Escape game*
- game_completion_sequence(), 180
- game design, 184
- game_loop()
 - in *Escape*, 116, 119, 122, 126, 161, 164, 172, 207, 212
 - in *Spacewalk*, 26, 27
- GAME LOOP section, 116, 119, 147, 161
- GAME_MAP, 61, 62, 64, 66, 75, 76
- game_over, 113, 119, 201
- GAME OVER message, 199, 201
- generate_map()
 - adding props, 154–156
 - centering the map, 141–142

- generating rooms, 66, 68, 76
- hazards, 208
- scenery, 104–105, 109
- starting, 132
- get, 11, 159, 160
- get_floor_type(), 69, 160
- get_item_under_player(), 158, 159
- get_width(), 107
- global, 28, 30
- global variables, 28, 30
- GPS system, 177, 179
- gray in code listings, 23

H

- hazard_data, 204, 206
- hazard_map, 206, 208, 210
- hazard_move(), 206, 208, 210
- hazards, 197, 203
 - choosing, for each room, 206
 - data for, 204
 - direction numbers, 203, 210, 216
 - drawing, 210
 - movement patterns, 203, 204, 210, 216
 - object numbers, 206
 - positioning, 184
 - room map for, 206, 208, 210
 - starting, 205, 206
 - stopping, 205, 207
 - stopping player from walking through, 212
 - testing, 211
 - toxic spills, 212
- HAZARDS section, 204, 205, 208
- hazard_start(), 205, 206, 208
- HEIGHT, 22, 30, 55, 142
- hidden props, 156, 168, 169

I

- IDE (integrated development environment), 9
- IDLE, 9, 14
 - color coding, 16, 21, 88, 252
 - cut and paste, 24
 - find and replace, 147, 207
 - opening a new window, 18

- Replace All, 147, 207
- script mode, 18, 29, 76
- searching within code, 87
- starting, 14
- title bar, 18
- if command, 28, 30, 108, 250, 251
 - using a list instead of, 140
- image_here, 107
- images
 - as arguments, 129
 - customizing, 215
 - filenames in Pygame Zero, 22
 - getting width, 107
- images folder, 8, 9, 18, 19, 22, 54, 55, 252
- image_to_draw, 116
- image_width, 107
- image_width_in_tiles, 107
- indentation, 22, 28, 49, 51, 66, 81, 108, 250
- index numbers, 36, 40, 45, 68, 104
 - 1 (final item in list), 104, 190
 - equivalent for dictionary, 81
- in keyword, 120, 140
- in_my_pockets, 154, 163, 164, 165
 - adding items, 160
 - removing items, 167
- input(), 192, 196
- insert(), 36
- int(), 71, 192, 196
- integer, 71
- integrated development
 - environment (IDE), 9
- interactive mode. *See* shell
- International Space Station, 14
- inventory, 154, 159, 177
 - adding items, 160
 - displaying, 160, 161, 162, 165
 - keyboard control, 164
 - removing items, 166, 167
 - testing, 165
- item_carrying, 154, 160, 164, 165, 167
 - False, 167
- item_counter, 163
- item_player_is_on, 160, 168
- items_player_may_carry, 94, 95, 165
- items_player_may_stand_on, 95, 120, 212

K

- keyboard controls
 - drop, 165, 166
 - in *Escape*, 116–119
 - examine, 165, 168
 - get, 160
 - playing *Escape*, 11
 - sensitivity, 74
 - spacebar, 168
 - in *Spacewalk*, 26–28
 - TAB, 164
 - use, 172
- keys, in dictionaries, 80, 95

L

- LANDER_SECTOR, 87
- LANDER_X, 87, 153
- LANDER_Y, 87, 153
- launch, 180
- left_tile_of_item, 168
- legs of astronaut, disappearing, 118, 127, 140
- line_number, 146
- listings folder, 9, 21
- lists, 34, 251, 252
 - 1 (as index number), 104, 190
 - accessing an item, 36, 39, 45
 - across multiple lines, 94
 - adding items to, 35, 45, 103
 - append(), 35, 45
 - as arguments, 129
 - checking whether an item is
 - in a list, 120, 140
 - combining two lists, 40–41
 - compared to dictionaries, 80
 - creating a list of 0s, 208
 - creating with list(), 94
 - deleting an item, 37
 - in keyword, 120
 - insert(), 36
 - inserting an item, 36
 - inside another list, 38, 39
 - inside dictionaries, 83, 84, 95
 - last item in, 104, 190
 - looping through items, 103
 - for maps, 42, 45
 - multiplying, 70

- lists, *continued*
 - nested, 38, 39
 - printing, 35
 - remove(), 35, 45
 - removing items from, 35, 45
 - replacing an item, 37, 45
 - slicing, 163
- list_to_show, 163
- local variables, 27, 129, 148, 157
- loops, 47, 49–50. *See also*
 - for command;
 - while command
- inside another loop, 50–52
- looping through a list, 103
- lowercase, 251

M

- MAKE MAP section, 105, 208
- map, 42, 45, 184. *See also* room_map
 - accessing an item, 43
 - coordinates, 43, 45
 - data format, 60, 61
 - designing your own, 60–61, 76–77, 139
 - doors, 184
 - extending, 75, 77
 - fixing errors, 65
 - moving between rooms, 122, 126
 - planet surface rooms, 64
 - printing an item number, 44
 - removing planet surface scenery, 104
 - replacing an item, 44
 - for space station, 60
 - using printed numbers, 49
- MAP_HEIGHT, 75
- map maker, 60
- Mars, 13
- Mastracchio, Rick, 14
- math, 17
- maze, 60
- messages, 145
- methods, 82
- modulo operator (%), 71, 163, 170, 206

- movement
 - between rooms, 122, 126
 - of player, 116–122, 158
- movement(), 74
- MP3 player, 169, 176
- multiline code, 94, 177
- multiplication (*), 56, 70

N

- NASA, 14, 20
- nested lists, 38–39
- nested loops, 50, 58
- None, 86, 251
- not, 120
- not defined error, 251
- not-equal-to operator (!=), 71, 103
- numbers in circles, 19

O

- object number, 98, 99
- objects
 - adding your own, 215
 - combining, 177
 - destroying, 152
 - dictionary. *See* objects dictionary
 - display errors, 91
 - displaying in Explorer, 89
 - drawing, 135
 - dropping, 11, 166
 - examining, 11, 168
 - hidden, 156, 168, 169
 - image file, 86
 - long description, 86
 - not currently in the game, 152
 - picking up, 11, 159, 160
 - selecting, 11
 - shadow image, 86
 - short description, 86
 - standard use messages, 172
 - using, 12, 171–181
- objects dictionary, 85, 88, 91–95, 106, 109, 151, 171, 177
 - changing images, 190, 194
 - doors, 194
 - doors animation, 190
- offset numbers, in astronaut animation, 121

- `.ogg` files, 201
- `old_hazard_x`, 210
- `old_hazard_y`, 210
- `old_player_x`, 120
- `old_player_y`, 120
- `open_door()`, 186, 188
- `outdoor_rooms`, 64, 70
- `output`, 16

P

- parentheses, `()`, 16, 22, 23, 37, 65, 135, 251
- `pgzrun`, 9–10, 20
- picking up objects, 11, 159, 160
- `pick_up_object()`, 159–161, 166
- PILLARS, 142
- pixels, 56
- planets, 80–85
- planet surface rooms, 64, 70, 76, 102
- player
 - drawing in room, 115, 135
 - movement, 116, 119, 158
 - movement between rooms, 122, 126
- PLAYER dictionary, 114, 126
- `player_direction`, 120
- `player_frame`, 119, 120
- `player_image`, 115, 131, 136
- `player_image_shadow`, 131
- PLAYER_NAME, 64
- `player_offset_x`, 116, 119, 121, 126, 136
- `player_offset_y`, 116, 119, 121, 126, 136
- PLAYER_SHADOW dictionary, 131, 136
- `player_x`
 - for *Escape*, 113, 136
 - for *Spacewalk*, 22, 26, 27
- `player_y`
 - for *Escape*, 113, 136
 - for *Spacewalk*, 22, 26, 27
- Pluto, 82, 84
- Poodle lander, 11, 87, 94, 153
- Portable Network Graphics (PNG), 55

- pressure pad, 139, 193, 194, 196
- `print()`, 16–17, 29, 53, 128
 - item number from map, 44
 - lists, 35
 - numbers, 17
- programming languages, 14
- programs, 18, 29
- prompt, 15
- prop 71 (Poodle lander), 153
- `prop_info`, 155
- PROP INTERACTIONS section 158, 159, 166
- `prop_number`, 155
- `prop_room`, 155
- props, 71, 98, 151–170
 - adding to `room_map`, 154
 - creating your own, 215
 - doors, 185
 - hidden, 156, 168, 169
 - interactions, 158
 - picking up, 11, 159, 160
 - positioning, 184
 - using, 12, 171–181
 - wide, 156
- PROPS section, 152, 153, 160, 165, 167, 169, 178, 179
- `prop_x`, 155
- `prop_y`, 155
- puzzles, 171, 177
 - creating your own, 215
 - design, 184
- Pygame Zero, 1, 14, 20, 21, 22, 54
 - drawing images, 22, 25
 - installing, 3, 5–6
 - on other computers, 6
 - running programs, 8, 9, 30
 - saving files, 54
 - testing installation, 6
- Python, 1, 14, 21
 - editor. *See* IDLE
 - installing, 3–5

Q

- quotation mark `"`, 17, 29, 251

R

- random, 156
- random.choice(), 103
- random.randint(), 87
- range(), 49, 57, 64, 65, 94, 107, 140
- Raspberry Pi
 - compatibility, 2, 6, 217
 - downloading game files, 7
 - running Pygame Zero programs, 9
 - software installation, 4
 - speed, 187, 202
 - starting IDLE, 15
- reason variable, 201
- recipes, 177–179, 181
- RECIPES, 178
- Rect, 138, 149, 163
- remove(), 35, 45
- remove_object(), 166, 167
- repeating
 - using clock, 119, 163
 - using loops. *See* loops
- Replace All, 147, 207
- replacing a list item, 37
- rescue ship, 180
- return, 69, 125, 157, 177
- robots. *See* hazards
- room
 - centering in the window, 141
 - designing your own, 89
 - drawing, 136, 139
 - drawing in 3D, 53, 55, 56
 - height, 69
 - maximum size, 61
 - name, 69
 - showing name on entry, 146
 - sizes, 77
 - width, 69
- room 0 (for storing extra items), 64, 152, 179
- room 26 (contains pressure pad), 139, 193, 215
- room 27 (engineering bay), 185
- room 32 (outside engineering bay), 185
- room_coordinate, 103
- room_data, 68, 71
- room_height, 55, 69

- room map, 167, 206
- room_map, 76
 - adding props, 154
 - adding scenery, 104–107
 - designing a room in the *Explorer*, 89–91
 - displaying with loops, 48–53
 - drawing the room, 55
 - emergency room example, 42–45
 - generating, 59–60, 62, 66–72
 - in player movement, 120
 - printing, 72
 - wide objects, 157
- room_name, 69, 147
- room_number, 103
- room_pixel_width, 142
- rooms
 - drawing, 72
 - designing, 96, 101, 107, 109
 - moving between, 122, 126
- room_width, 55, 69
- Run Module, 65

S

- saving, 18, 19, 54, 62
- scenery, 97, 108
 - adding to room_map, 105
 - changing, 109
 - changing the data, 101
 - combining with props, 177–179
 - creating your own, 215
 - dictionary, 98–100, 109
 - drawing, 139
 - error in data, 101
 - on planet surface, 102
 - randomly chosen, 103
 - randomly positioned, 103
 - removing for planet surface rooms, 104
 - shadows, 136
 - using, 12, 171–181
- scenery dictionary, 151
- scenery_number, 106
- SCENERY section, 99, 102
- scenery_x, 106
- scenery_y, 106

- scheduling, 74
- score, 21
- screen.blit(), 22, 30, 55, 135, 163
- screen.draw.filled_rect(), 138, 149
- screen.draw.text(), 146
- script mode, 18, 29, 76
- searching in your code, 87
- selected_item, 154, 160, 164
- selected_marker, 163
- selecting objects, 11, 164
- semicolon (;), 251
- sensitivity of keyboard controls, 74
- shadows, 57, 128, 130, 135, 139
 - drawing, 140
 - scenery, 136
 - spilling out of the
 - game area, 138
 - standard, 140
- shell
 - Python, 15, 18, 72, 76
 - Raspberry Pi, 9
- short description, 177
- show_text(), 146, 148
- shut_engineering_door(), 190
- side_edge, 70
- slicing, lists, 163
- slow programs, 2, 250
- snip--, 23
- software installation, 3–6
- soil, 70
- sound effects
 - alarm, 199
 - doors open, 187
 - fanfare, 214
 - playing, 201, 216
- sounds folder, 8, 9, 201, 252
- space station
 - inhabitants, 64
 - map, 60
 - rooms, 76
- Spacewalk, 14, 18–31
- spelling errors, 251–252
- spoilers, 2, 86, 171
- square brackets, [], 34, 36, 37, 65, 251
- standard_responses, 173, 181
- starfield, 18, 22
- start_display, 163

- start_room(), 125, 146, 206
- START section, 118, 144, 162, 202
- stopping programs, 23, 30
- storytelling, 184
- str(), 74, 94
- strings, 16, 29, 65
 - combining, 65
 - converting to numbers, 192, 196
 - drawing, 145, 201, 216
 - typing into a program, 192
- subtracting numbers, 149
- suit_stitched, 174
- switching off instructions, 101, 107, 112, 212, 213

T

- TAB key, 11, 161, 163, 164
- teleporter
 - adding, 192
 - disabling, 213
 - using, 192, 195
- testing, 65, 72, 197, 212
- text. *See* strings
- text_lines, 146
- text_to_show, 146
- this_scenery, 106
- tiles, 56, 61, 113, 126
- TILE_SIZE, 105
- time limit, 202
- time.sleep(), 160
- top_left_x, 56, 64, 141
- top_left_y, 56, 64, 141
- toxic spills, 212
- training missions, 3
- True, 61, 71, 81, 83, 251
- tuple, 23, 30, 131, 252
- turning off instructions, 101, 107, 112, 212, 213

U

- uncommenting, 147, 207, 212
- unexpected indent, 250
- uppercase, 251
- use_message, 173
- use_object(), 173, 174, 181, 185
- USE OBJECTS section, 172, 180
- using objects, 12, 171–173, 174–179

V

- values, in dictionaries, 80
- variables, 21, 29, 76, 105
 - as dictionary keys, 81
 - game progress, 174
 - global, 28, 30
 - increasing and decreasing values, 28
 - local, 129, 148, 157
 - names, 21, 22, 52
 - for player movement, 112
- VARIABLES section, 105, 199, 213

W

- walls, 68, 69, 70, 74, 99. *See also* exits
 - fading in and out, 142
 - front, 139
 - transparency, 130, 139, 142
- wall_transparency_frame, 131, 143
- .wav files, 201
- weight sensor. *See* pressure pad
- while command, 81, 250, 251
- while True, 81

- whiteboard, 215
- whole number, 71
- wide objects, 91, 106, 108, 157
- wide props, 156
- WIDTH, 22, 30, 55, 142
- Windows 8, starting IDLE in, 15
- Windows 10, starting IDLE in, 14
- Windows Explorer, 8
- window size, 22
- Windows PC, 4, 6, 8, 10

X

- x position, 22, 24, 98

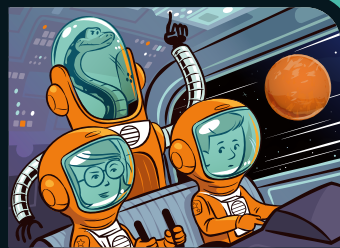
Y

- "You're out of air!", 200
- y position, 23, 24, 98

Z

- ZIP file, 7, 8

CODE YOUR OWN SPACE STATION ADVENTURE GAME!



Mission Python is a hands-on guide to building a computer game in Python—a beginner-friendly programming language used by millions of professionals as well as hobbyists who just want to have fun.

In *Mission Python*, you'll code a puzzle-based adventure game, complete with graphics, sound, and animations. Your mission: to escape the station before your air runs out. To make it to safety, you must explore the map, collect items, and solve puzzles while avoiding killer drones and toxic spills. When you've finished building your game, you can share it with your friends!

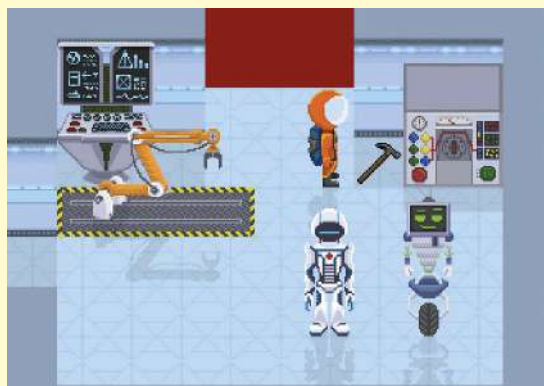
As you code, you'll learn fundamentals of Python, like how to:

- Store data in variables, lists, and dictionaries
- Add keyboard controls to your game
- Create functions to organize your instructions
- Make loops to repeat blocks of code
- Add graphics, sound, and animations to your game

The book uses Pygame Zero, a free resource that makes coding games easier. Plus, all graphics, sound, and code used in the game are available for you to download for free!

ABOUT THE AUTHOR

Sean McManus is a computer book author with extensive experience in writing coding books for children. Visit his website at www.sean.co.uk.



BUILD THIS GAME!

Requires Python 3.x on Windows or Raspberry Pi (it's free!)



THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com



\$29.95 (\$39.95 CDN)

ISBN: 978-1-59327-857-1



9 781593 278571

SHRIVE IN: PROGRAMMING
LANGUAGES / PYTHON