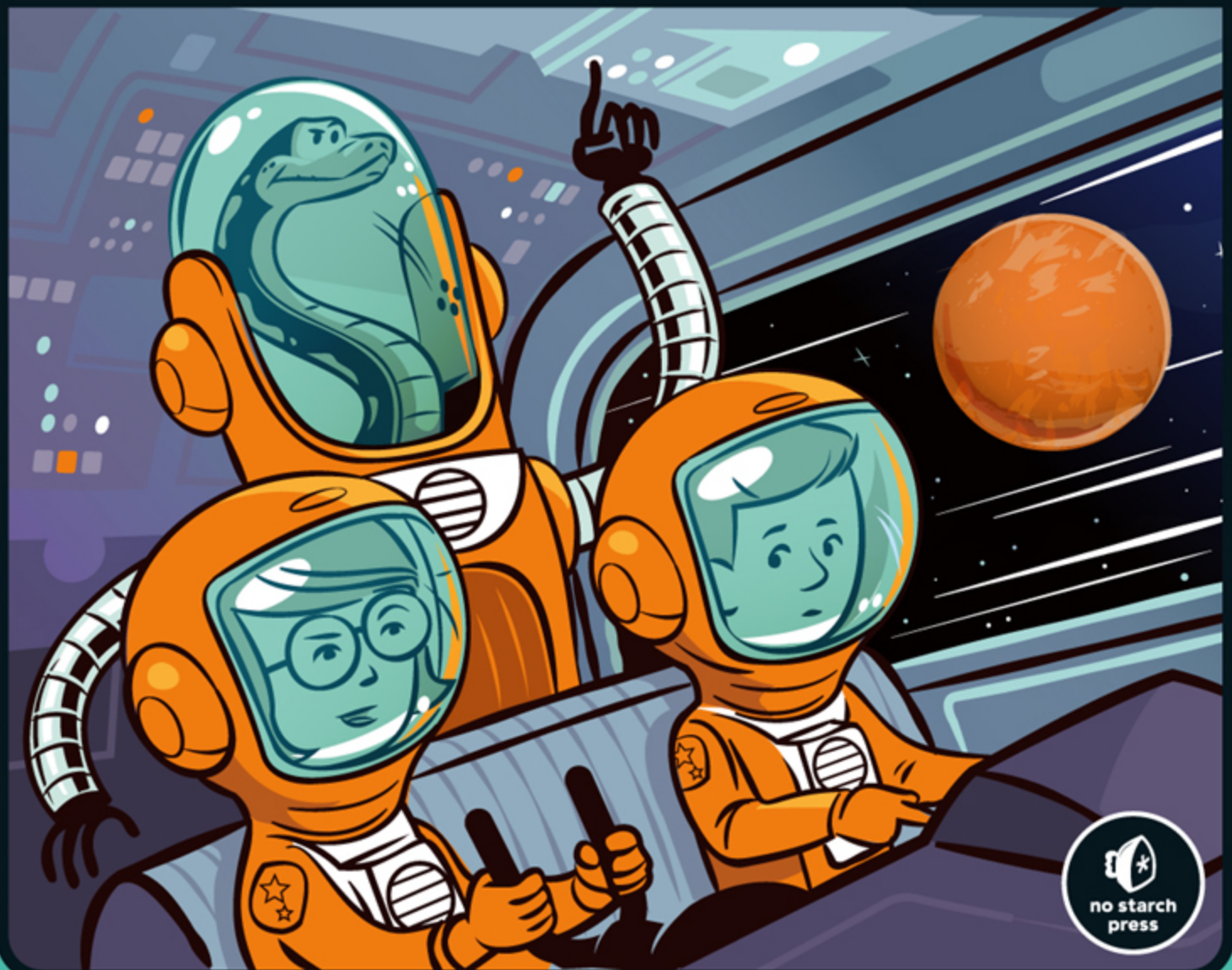


MISSION PYTHON

CODE A SPACE ADVENTURE GAME!

SEAN MCMANUS



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.

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.)

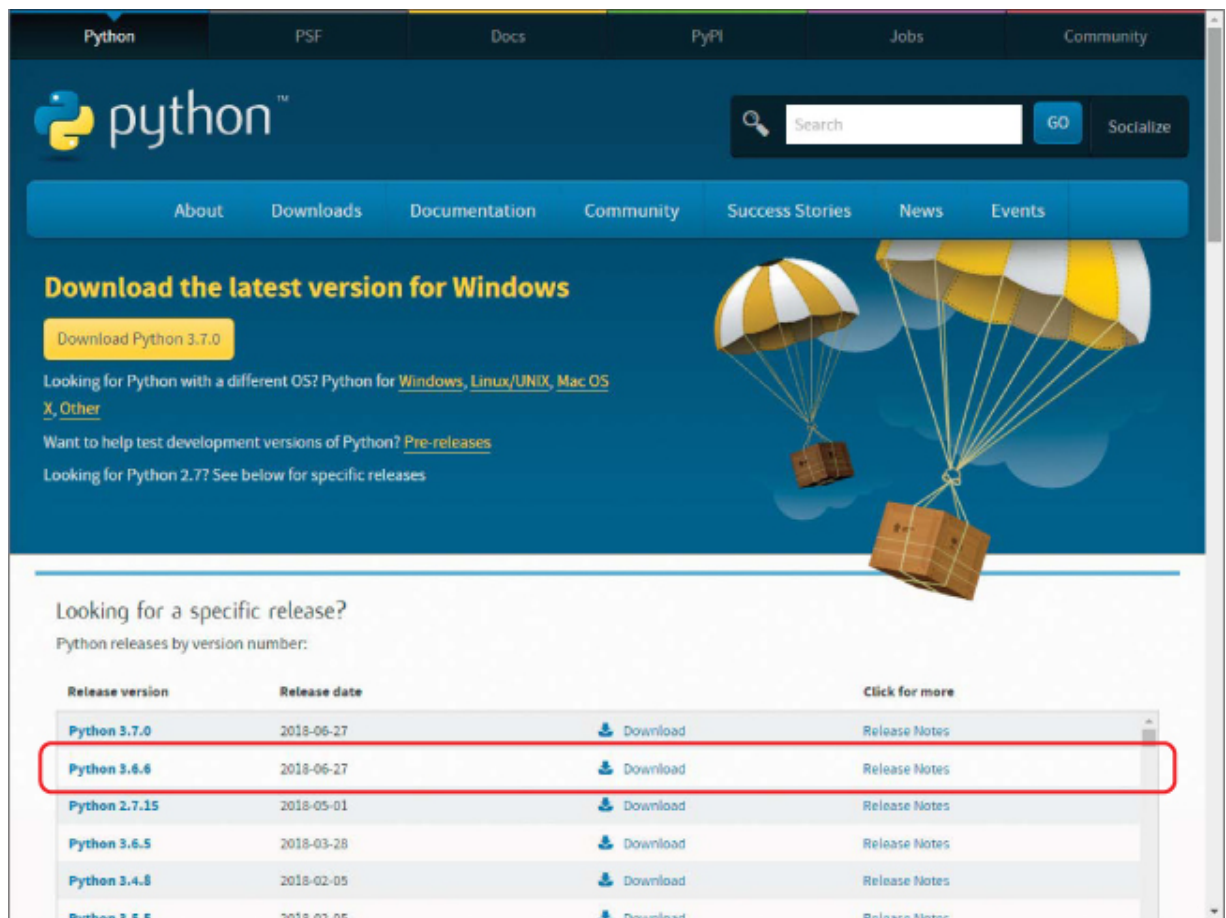


Figure 1: The Python downloads page

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**.

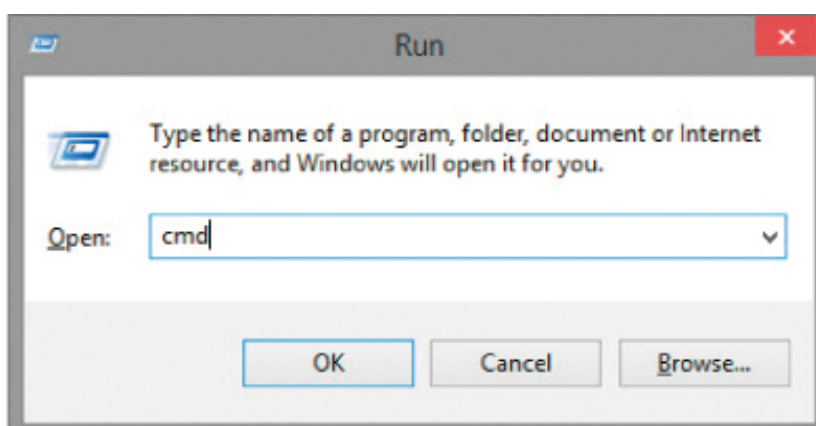


Figure 3: The Windows Run dialog box

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.

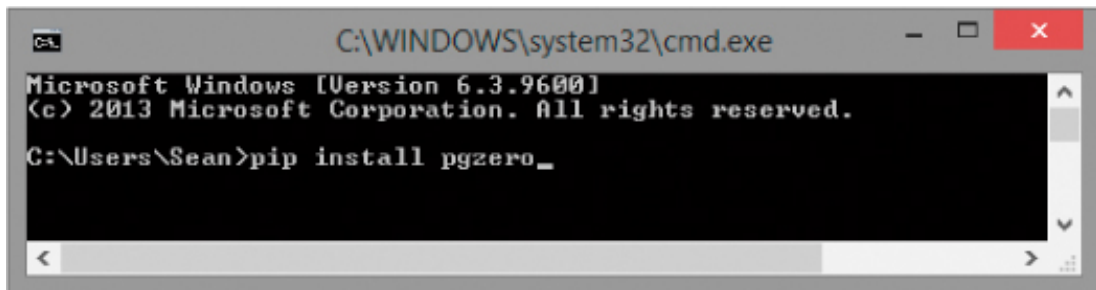


Figure 4: The command line window

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**.

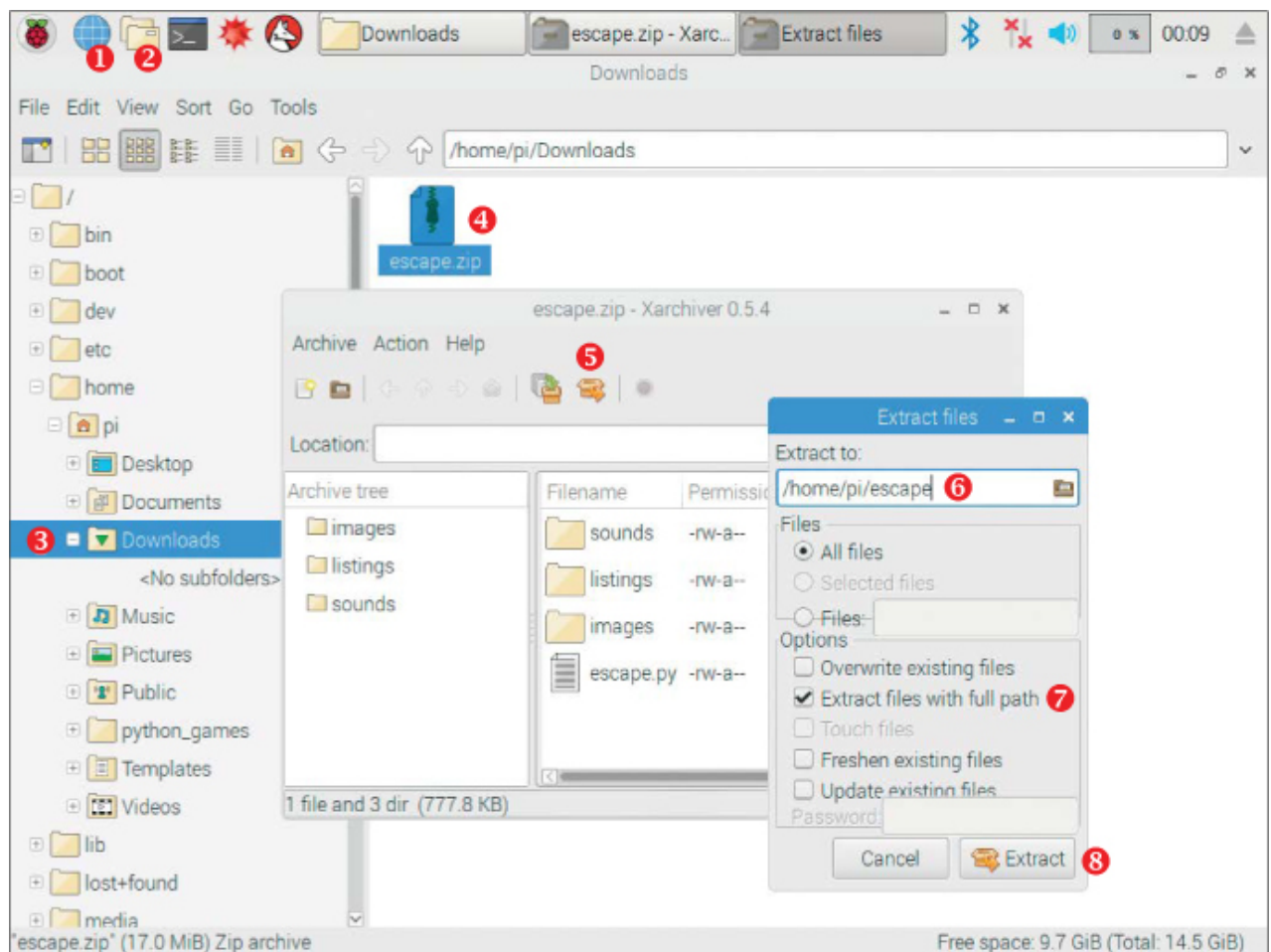


Figure 5: The steps you should take to unzip your files

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**.

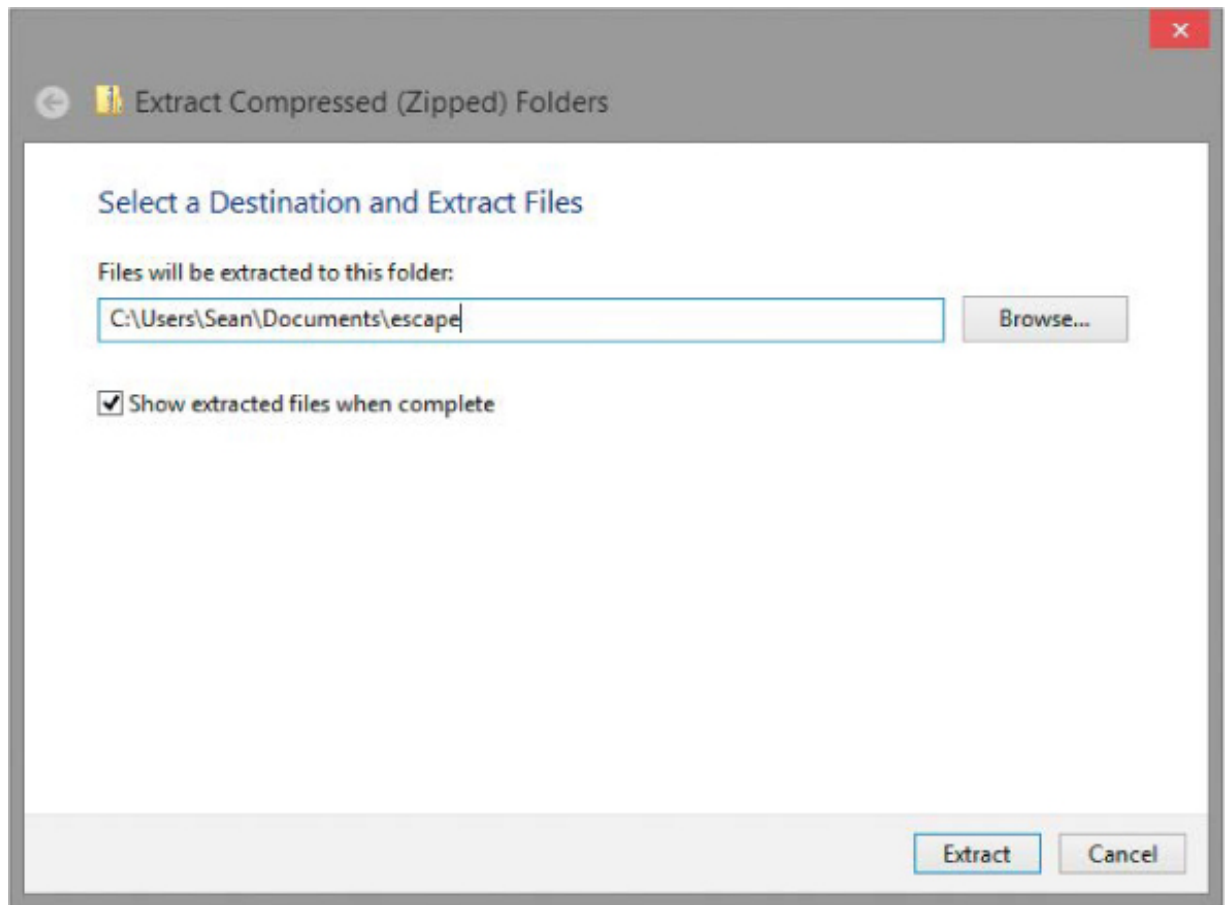


Figure 6: Setting the folder to unzip the game files into

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 be alongside the *images* and *sounds* folders to work correctly.

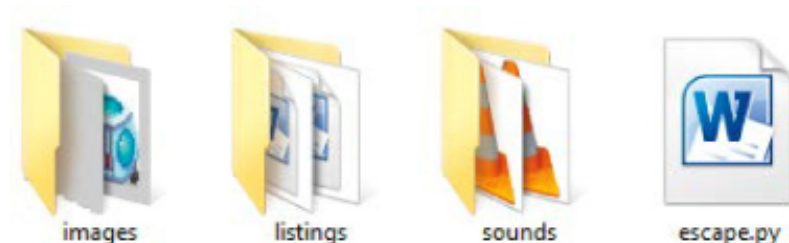


Figure 7: The contents of the ZIP file as they might appear in Windows

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.

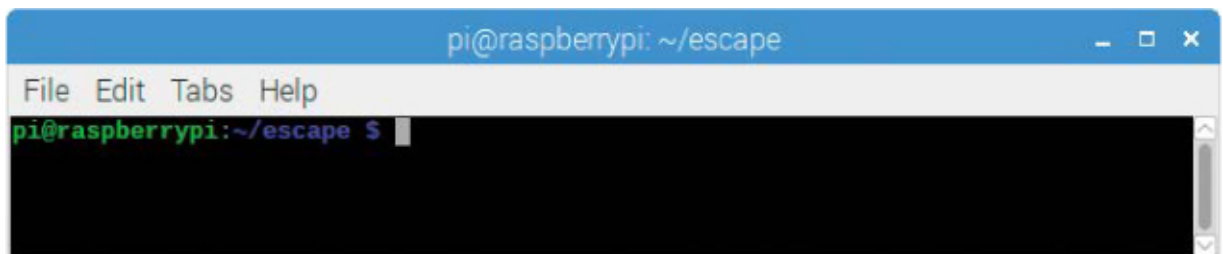


Figure 8: The command line window on the Raspberry Pi

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.

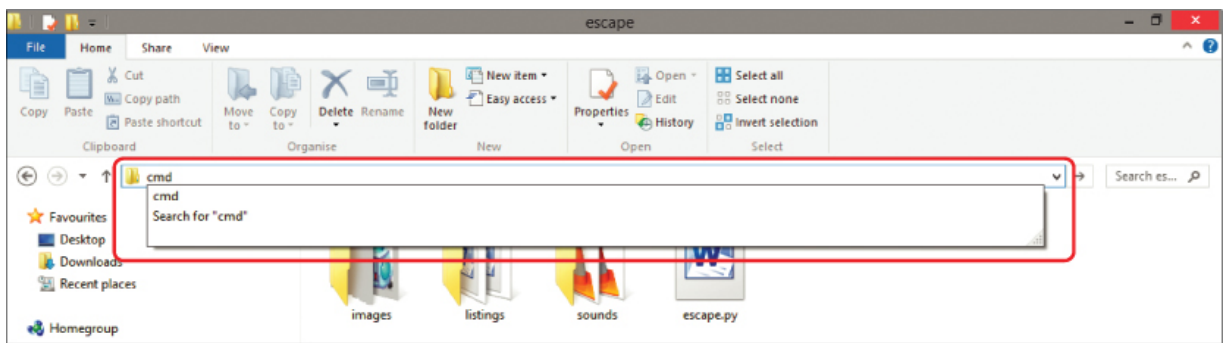


Figure 9: Finding the path to your Pygame files

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](#).

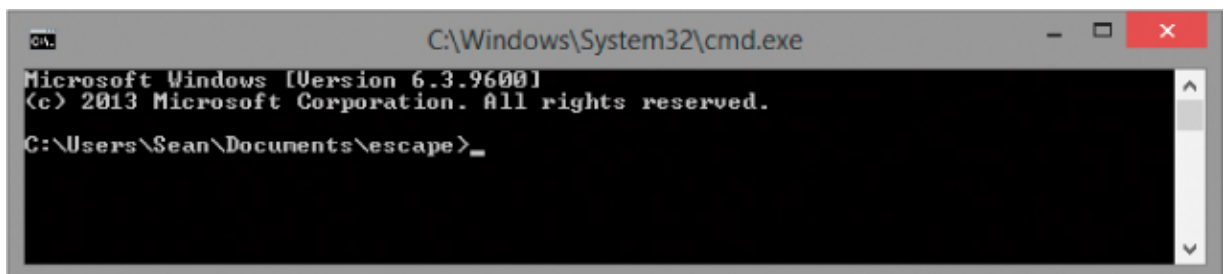


Figure 10: The command line window in Windows

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.

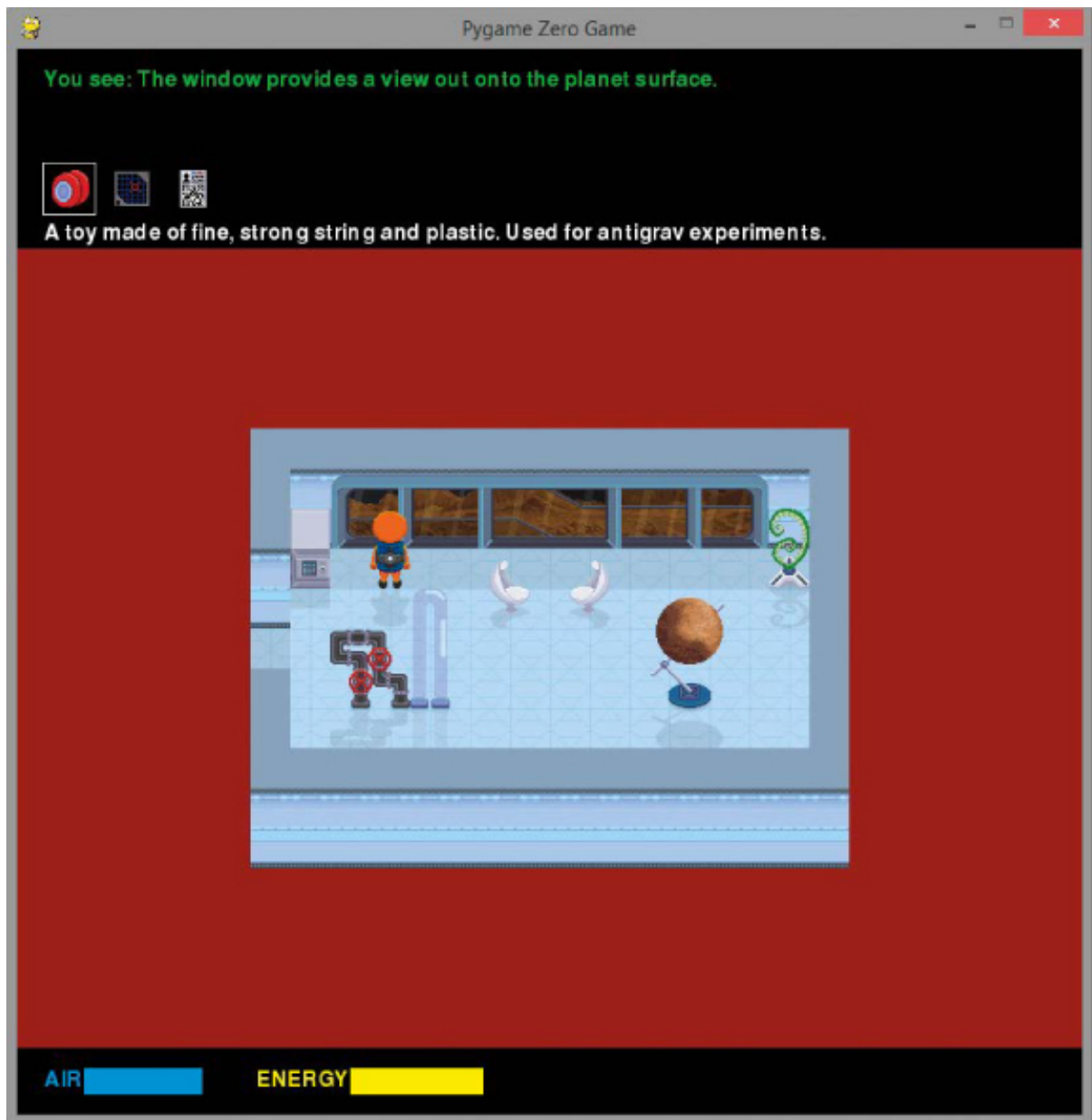


Figure 11: Your adventure begins!

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!

1

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.

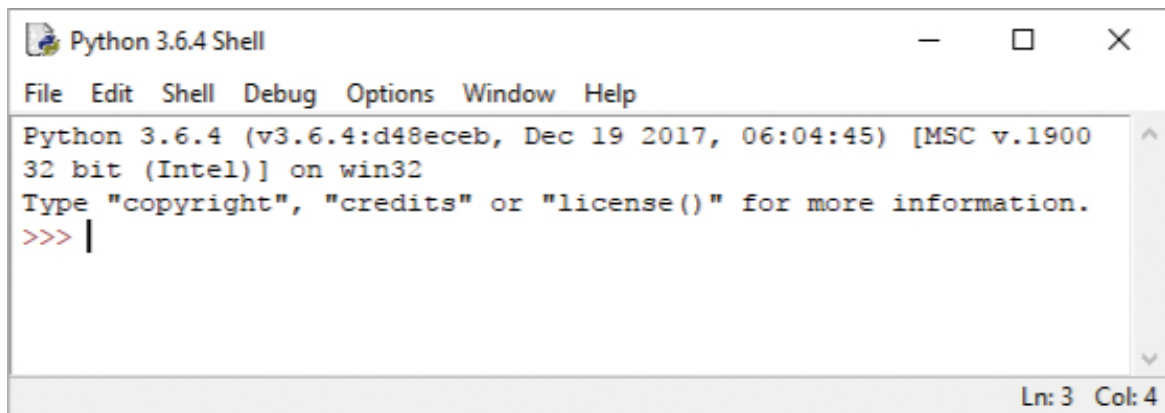


Figure 1-2: The Python shell

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*.

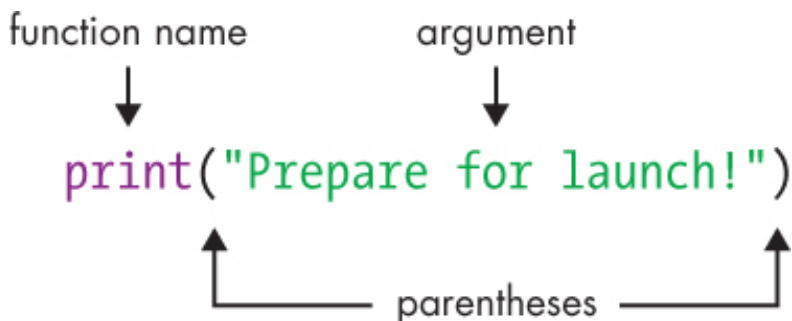


Figure 1-3: The different parts of your first instruction

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.
3. 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

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.

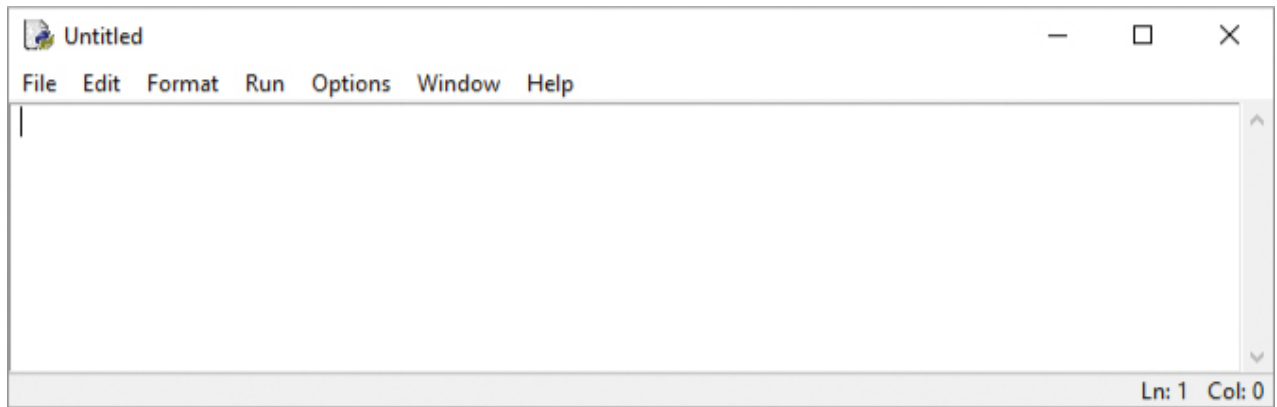


Figure 1-4: Python script mode

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 ❹ needs a colon at the end of its line, and the next line ❺ 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.

listing1-1.py

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

❷ WIDTH = 800
   HEIGHT = 600
❸ player_x = 600
   player_y = 350

❹ 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).

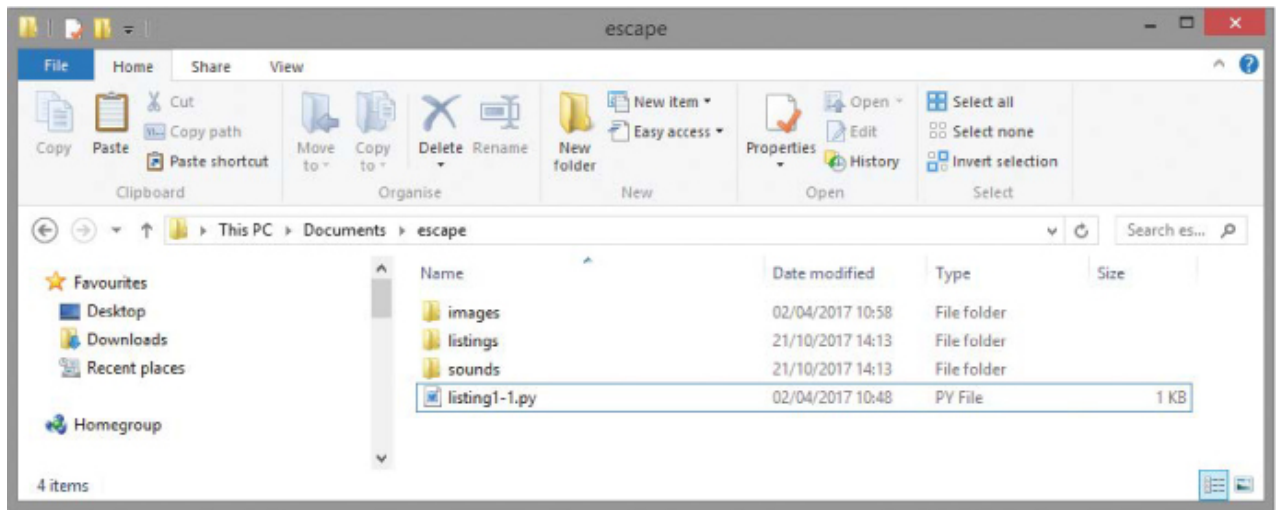


Figure 1-5: Your new Python program and the images folder should be stored in the same place.

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](#).

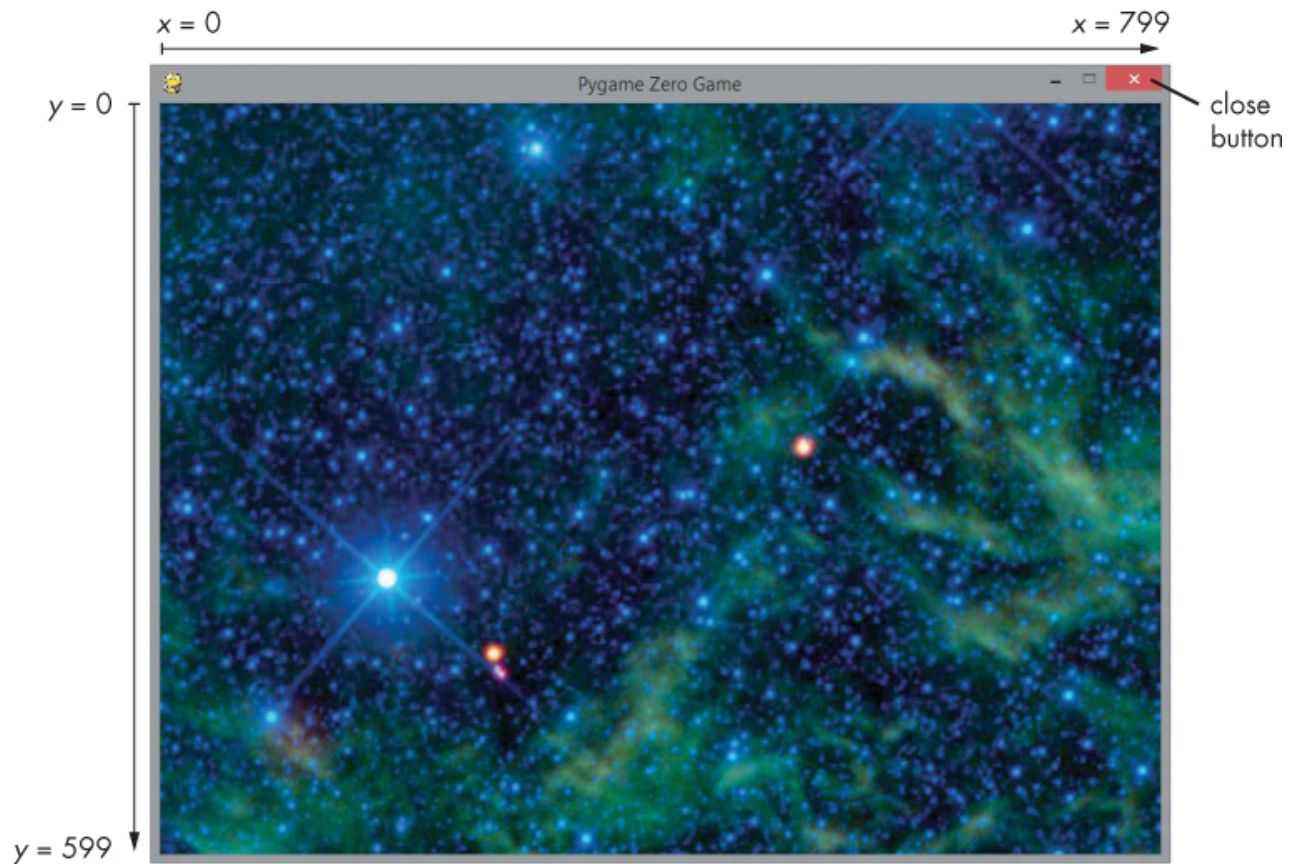


Figure 1-6: The starfield. The starfield image is courtesy of NASA/JPL-Caltech/UCLA and shows star cluster NGC 2259.

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` ❹, 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 ❺ 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)` ❸, 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.

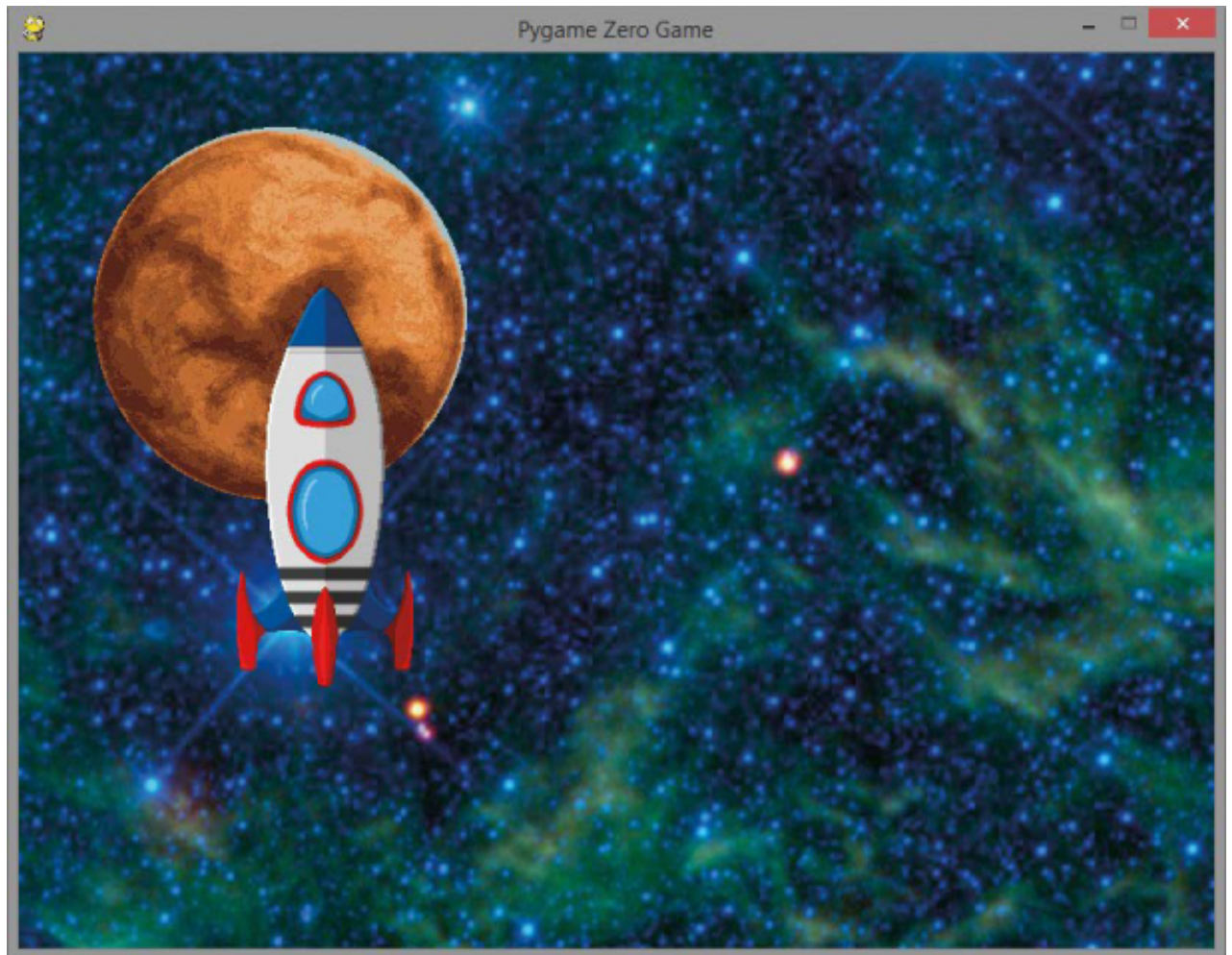


Figure 1-7: Mars and the spaceship. The Mars image was taken by the Hubble Space Telescope in 1991.

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.

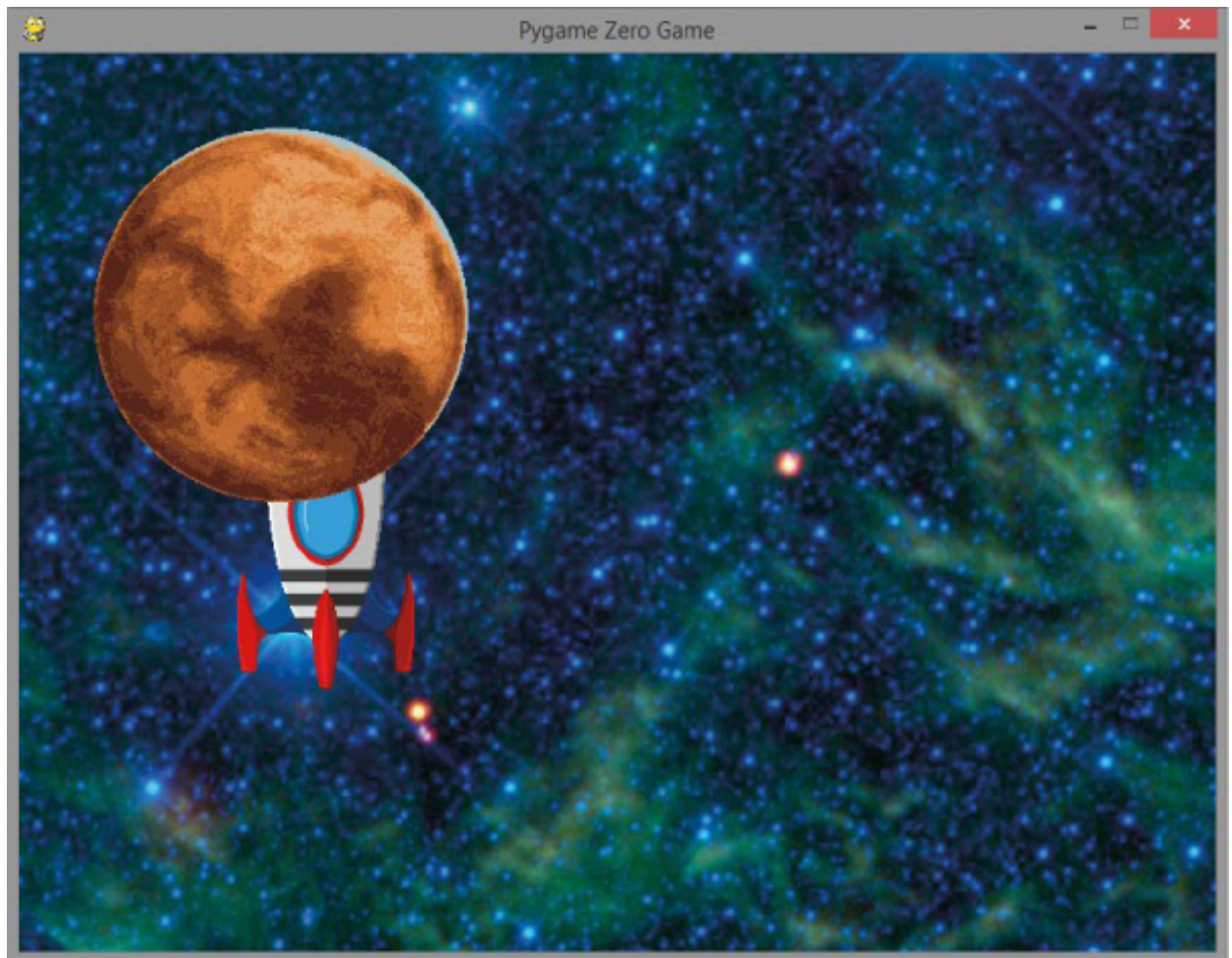


Figure 1-8: The spaceship is now behind the planet.

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*.

```
--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:
❽             player_x -= 5
❾         elif keyboard.up:
❿             player_y -= 5
⓫         elif keyboard.down:
⓬             player_y += 5

⓭ 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.

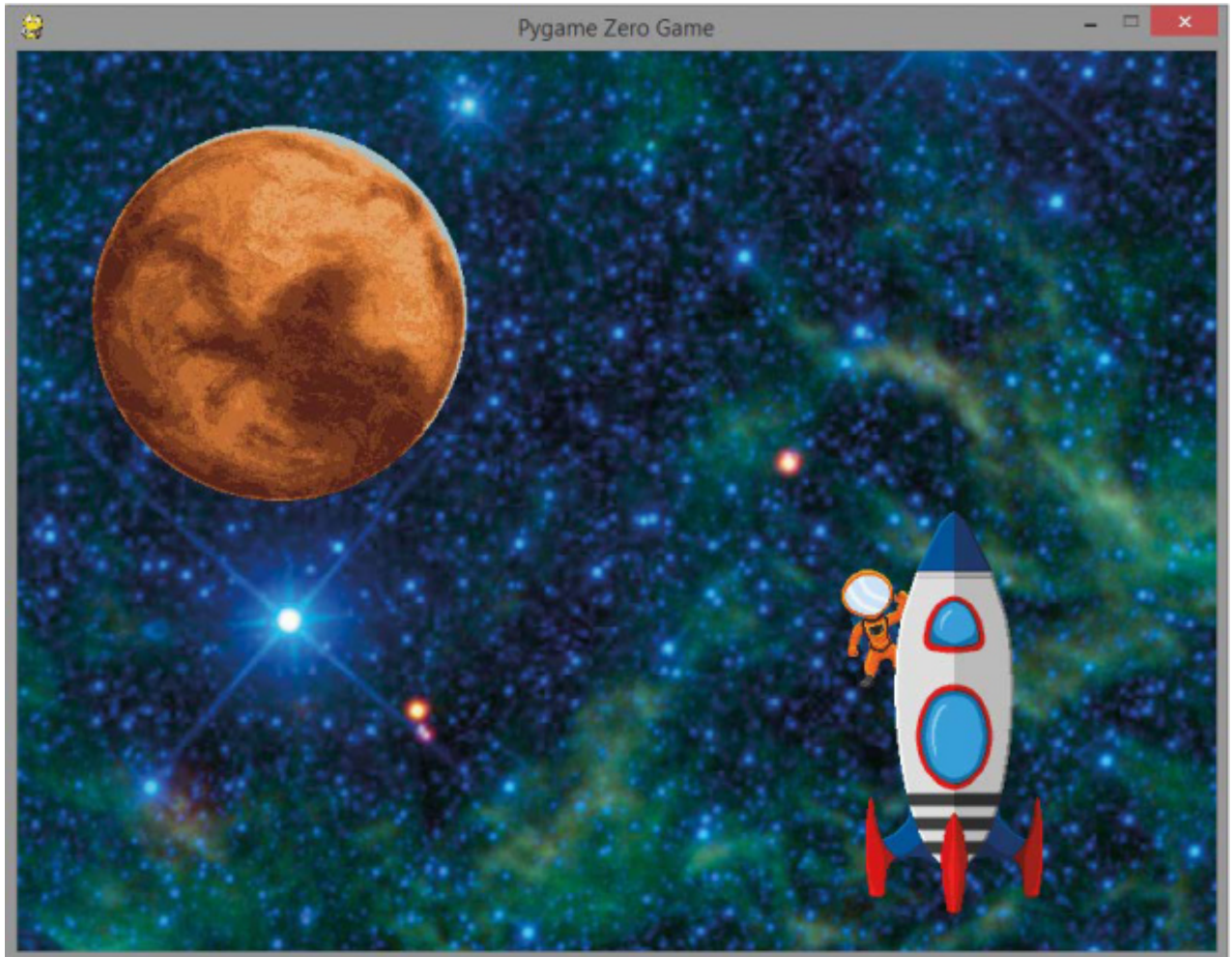


Figure 1-9: You emerge from the ship for your spacewalk.

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()` ③. 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 ④. 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 ❸. If it is, we change the value of `player_x` by adding 5 ❹, 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 ❺, 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 ❽ 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 ❾ 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--
```

2

LISTS CAN SAVE YOUR LIFE



Astronauts live by lists. The safety checklists they use help make sure all systems are working before they entrust their lives to those systems. For example, emergency checklists tell the astronauts what to do in dire situations to prevent them from panicking. Procedural checklists confirm that they're using their equipment correctly so nothing breaks and prevents them from returning home. These lists just might save their lives one day.

In this chapter, you'll learn how to manage lists in Python and how to use them for checklists, maps, and almost anything in the universe. When you build the *Escape* game, you'll use lists to store information about the space station layout.

MAKING YOUR FIRST LIST: THE TAKE-OFF CHECKLIST

Take-off is one of the most dangerous aspects of space travel. When you're strapped to a rocket, you want to double-check everything before it launches. A simple checklist for take-off might contain the following steps:

☐ Put on suit

☐ Seal hatch

☐ Check cabin pressure

☐ Fasten seatbelt

Python has the perfect way to store this information: the Python *list* is like a variable that stores multiple items. As you'll see, you can use it for numbers and text as well as a combination of both.

Let's make a list in Python called `take_off_checklist` for our astronauts to use. Because we're just practicing with a short example, we'll enter the code in the Python shell rather than creating a program. (If you need a refresher on how to use the Python shell, see [“Introducing the Python Shell”](#) on page 15.) Enter the following in the IDLE shell, pressing ENTER at the end of each line to start a new line in the list:

```
>>> take_off_checklist = ["Put on suit",  
                           "Seal hatch",  
                           "Check cabin pressure",  
                           "Fasten seatbelt"]
```

RED ALERT

*Make sure the brackets, quote marks, and commas in your code are precise. If you get any errors, enter the list code again, and double-check that the brackets, quotes, and commas are in the correct places. To avoid having to retype the code, use your mouse to highlight the text in the shell, right-click the text, select **Copy**, right-click again, and select **Paste**.*

Let's take a closer look at how the `take_off_checklist` list is made. You mark the start of the list with an opening square bracket. Python knows the list is not finished until it detects the final closing square bracket. This means you can press ENTER at the end of each line to continue typing the instruction, and Python will know you're not finished until you've given it the final bracket.

Quote marks tell Python that you're giving it some text and where each piece of text starts and ends. Each entry needs its own opening and closing quote marks. You also need to separate the different pieces of text with commas. The last entry doesn't need a comma after it, because there isn't another list item following it.

SEEING YOUR LIST

To see your checklist, you can use the `print()` function, as we did in [Chapter 1](#). Add the name of your list to the `print()` function, like this:

```
>>> print(take_off_checklist)
['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt']
```

You don't need quotes around `take_off_checklist`, because it's the name of a list, not a piece of text. If you do put quotes around it, Python will just write the text `take_off_checklist` onscreen instead of giving you back your list. Try it to see what happens.

ADDING AND REMOVING ITEMS

Even after you've created a list, you can add an item to it using the `append()` command. The word *append* means to add something at the end (think of an appendix, at the end of a book). You use the `append()` command like this:

```
>>> take_off_checklist.append("Tell Mission Control checks are complete")
```

You enter the name of the list (without quote marks) followed by a period and the `append()` command, and then put the item to add in parentheses. The item will be added to the end of the list, as you'll see when you print the list again:

```
>>> print(take_off_checklist)
['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt', 'Tell
Mission Control checks are complete']
```

You can also take items out of the list using the `remove()` command. Let's remove the `seal hatch` item:

```
>>> take_off_checklist.remove("Seal hatch")
>>> print(take_off_checklist)
['Put on suit', 'Check cabin pressure', 'Fasten seatbelt', 'Tell Mission
Control checks are complete']
```

Again, you enter the name of the list followed by a period and the `remove()` command, and then specify the item you want to remove inside the parentheses.

RED ALERT

When you're removing an item from a list, make sure what you type matches the item exactly, including capital letters and any punctuation. Otherwise, Python won't recognize it and will give you an error.

USING INDEX NUMBERS

Hmm, we should probably put the `seal hatch` check back into the list before anyone at Mission Control notices. You can insert an item in a specific position in the list by using that item's index number. The *index* is the position of the item in the list. Python starts counting at 0, not 1, so the first item in the list always has an index of 0, the second item has an index of 1, and so on.

INSERTING AN ITEM

Using the position index, we'll put `seal hatch` back where it belongs:

```
>>> take_off_checklist.insert(1, "Seal hatch")
>>> print(take_off_checklist)
['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt', 'Tell
Mission Control checks are complete']
```

Phew! I think we got away with it. Because the index starts at 0, when we inserted `seal hatch`, we placed it at position 1, the second item in the list. The rest of the list items shifted down in the list to make room, increasing their index numbers, as shown in Figure 2-1.

```
["Put on suit", "Check cabin pressure", "Fasten seatbelt", "Tell Mission Control..."]
  ↑           ↑           ↑           ↑
index 0      index 1      index 2      index 3

["Put on suit", "Seal hatch", "Check cabin pressure", "Fasten seatbelt", "Tell Mission Control..."]
  ↑           ↑           ↑           ↑           ↑
index 0      index 1      index 2      index 3      index 4
```

Figure 2-1: Inserting an item at index 1. Top row: before insertion. Bottom row: after insertion.

ACCESSING AN INDIVIDUAL ITEM

You can also access a particular item in a list using the list name with the index number of the item you want to access in square brackets. For example, to print particular items in the list, you can enter the following:

```
>>> print(take_off_checklist[0])
```

```
Put on suit
```

```
>>> print(take_off_checklist[1])
```

```
Seal hatch
```

```
>>> print(take_off_checklist[2])
```

```
Check cabin pressure
```

Now you can see individual items in the list!

RED ALERT

Don't mix up your brackets. Roughly speaking: Use square brackets when you're telling Python which list item to use. Use parentheses when you're doing something to the list or items in it, such as printing the list or appending items to it. Every opening bracket needs a closing bracket of the same type.

REPLACING AN ITEM

You can also replace an item if you know its index number. Simply enter the list name followed by the index of the item you want to replace, and then use an equal sign (=) to tell Python what you want at that index, like this:

```
>>> take_off_checklist[3] = "Take a selfie"
```

```
>>> print(take_off_checklist)
```

```
['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Take a selfie', 'Tell  
Mission Control checks are complete']
```

The old item at index 3 is removed and replaced with the new item. Be aware that when you replace an item, Python forgets the original. Recall your training to put it back, like this:

```
>>> take_off_checklist[3] = "Fasten seatbelt"
```

```
>>> print(take_off_checklist)
```

['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt', 'Tell Mission Control checks are complete']

DELETING AN ITEM

If you know where an item is in a list, you can delete it using its index number too, like this:

```
>>> del take_off_checklist[2]
>>> print(take_off_checklist)
['Put on suit', 'Seal hatch', 'Fasten seatbelt', 'Tell Mission Control checks are complete']
```

The "Check cabin pressure" item disappears from the list.

TRAINING MISSION #1

It's time to practice your skills! We just deleted item 2 in the list. Can you insert it back into the list in the correct position? Print the list to check that it worked.

CREATING THE SPACEWALK CHECKLIST

As you know from [Chapter 1](#), another dangerous activity for an astronaut is venturing out into the black vacuum of space with just a suit to protect you and provide oxygen. Here is a checklist to help keep you safe when you're spacewalking:

- ☐ Put on suit
- ☐ Check oxygen
- ☐ Seal helmet
- ☐ Test radio
- ☐ Open airlock

Let's make this checklist into a Python list. We'll call it `spacewalk_checklist`, like this:

```
>>> spacewalk_checklist = ["Put on suit",  
                             "Check oxygen",  
                             "Seal helmet",  
                             "Test radio",  
                             "Open airlock"]
```

Remember to be careful with the commas and brackets.

TRAINING MISSION #2

It's always a good idea to test your code so you know it's working as it should. Can you try printing all the list items to check that they're in the right place?

A LIST OF LISTS: THE FLIGHT MANUAL

We have two checklists now: one for take-off and one for spacewalking. We can organize them by putting them into another list to create our “flight manual.” Think of the flight manual as a folder that contains two sheets of paper, and each piece of paper has one list on it.

MAKING A LIST OF LISTS

Here is how we make the flight manual list of lists:

```
>>> flight_manual = [take_off_checklist, spacewalk_checklist]
```

We give IDLE the `flight_manual` list name, use the equal sign (`=`), and then add the two lists we want to put in the `flight_manual` list inside square brackets. As we did earlier when making lists, we separate the two items with a comma. The new `flight_manual` list has two items in it: the `take_off_checklist` and the `spacewalk_checklist`. When you print `flight_manual`, it looks like this:

```
>>> print(flight_manual)  
[['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt',  
'Tell Mission Control checks are complete'], ['Put on suit', 'Check oxygen',  
'Seal helmet', 'Test radio', 'Open airlock']]
```

TIP

Remember that you don't need to use quote marks around list names; you use them only when you're entering text into a list.

RED ALERT

If you don't see 'Check cabin pressure' in your list, it's because you skipped Training Mission #1. To make it easier to follow along, I recommend you go back and complete that mission. You can check the training mission answers at the end of the chapter if you need to.

The output looks messy! To work out what's going on, look closely at the brackets. Square brackets mark the start and end of each list. If you strip out the list items, the output looks like this:

```
[ [ first list is here ], [ second list is here ] ]
```

In the middle, you can see where the first list ends with a closed bracket followed by a comma before the next list begins with an opening bracket. So what happens when you try to print the first item in the `flight_manual` list?

```
>>> print(flight_manual[0])
```

The first item is the `take_off_checklist`, so the output looks like this:

```
['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt', 'Tell  
Mission Control checks are complete']
```

TRAINING MISSION #3

Try adding other checklists to `flight_manual` and printing them. For example, you could add a checklist for landing on a planet or docking with another

spaceship.

FINDING AN ITEM IN THE FLIGHT MANUAL

If you want to look at a particular item in one of the lists in `flight_manual`, you must give Python two pieces of information: the list the item is in, and the index of the item in the list, in that order. For each piece of information, you can use index numbers, like this:

```
>>> print(flight_manual[0][1])
```

```
Seal hatch
```

Check your result against the printout of your checklist higher up in the shell. The `seal hatch` item is in the first list (index 0), which is the `take_off_checklist`, and it's the second item in that list (index 1). Those are the two index numbers we used to find it. Let's choose an item from the second list:

```
>>> print(flight_manual[1][3])
```

```
Test radio
```

This time, we're printing from the second list (index 1), and from that list, we're printing the fourth item (index 3). Although it might seem confusing that Python starts counting at 0, soon it will become second nature to subtract one from the position number you want. Be careful that you don't end up buying one fewer of everything when you go shopping!

TIP

To print a list or variable on the screen, you can leave out the `print()` command when you're typing into the shell, like so:

```
>>> flight_manual[0][2]
```

```
'Check cabin pressure'
```

This only works in the shell, though, and not in a program. Often, you'll have many ways to do the same thing in Python. This book focuses on the techniques that will most help you make the *Escape* game. As you learn

COMBINING LISTS

You can join two lists using a plus sign (+) to combine them into a single list. Let's make a list of all the skills needed for take-off and spacewalking and call it `skills_list`:

```
>>> skills_list = take_off_checklist + spacewalk_checklist
>>> print(skills_list)
['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt', 'Tell
Mission Control checks are complete', 'Put on suit', 'Check oxygen', 'Seal
helmet', 'Test radio', 'Open airlock']
```

The output you see here is a single list containing the skills astronauts need from the two lists we already made. We can also add more skills to the list by entering the combined list's name and using `+=` to add single items or other lists to the end of it. (In [Chapter 1](#), you learned how to use `+=` to add a number to a variable's value.)

Few people get to go into space, so a big part of an astronaut's role is to share that experience. Let's add a list called `pr_list` for public relations (PR) skills that an astronaut might need. I think there might be a place for selfie skills after all!

```
>>> pr_list = ["Taking a selfie",
               "Delivering lectures",
               "Doing TV interviews",
               "Meeting the public"]
>>> skills_list += pr_list
>>> print(skills_list)
['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt',
'Tell Mission Control checks are complete', 'Put on suit', 'Check oxygen',
'Seal helmet', 'Test radio', 'Open airlock', 'Taking a selfie', 'Delivering
lectures', 'Doing TV interviews', 'Meeting the public']
```

The `skills_list` now has the items from `pr_list` added. The `skills_list` is still just a single list with individual items in it, unlike `flight_manual`, which has two separate lists inside it.

TIP

You might have noticed that this code line:

```
skills_list += pr_list
```

is just a shorter way of writing this:

```
skills_list = skills_list + pr_list
```

It's a very useful shortcut!

MAKING MAPS FROM LISTS: THE EMERGENCY ROOM

Navigation is an essential skill for an astronaut. You must always know where you are, where your nearest sanctuary is, and even where the air is so you're always ready in an emergency. The *Escape* game will keep a map of the room the player is in, so it can draw the room correctly and enable the player to interact with objects. Let's look at how we can use lists to make a map of the emergency supplies room.

MAKING THE MAP

Now that you know how to manage lists and lists inside lists, you can make maps. This time, we'll create a program rather than working in the shell. At the top of the Python window, select **File ▶ New File** to open a new window.

Enter Listing 2-1 into your new program window:

listing2-1.py

```
room_map = [ [1, 0, 0, 0, 0],
              [0, 0, 0, 2, 0],
              [0, 0, 0, 0, 0],
              [0, 3, 0, 0, 0],
              [0, 0, 0, 0, 4]
            ]
print(room_map)
```

Listing 2-1: Setting up the emergency room

Note that you don't need a comma at the end of the last line in the list. This program creates and displays a list called `room_map`. Our new emergency room is five meters by five meters. The `room_map` list contains five lists. Each of those lists contains five numbers, which represent one row of the map. I've lined up the numbers in the code so it looks like the grid shown in [Figure 2-2](#), which shows a map of the room. Compare the diagram and the program; you'll see that the first list is for the top row, the second list is for the second row, and so on. A 0 represents an empty space in the grid, and the numbers 1 to 4 are for various emergency items in the room. The numbers we'll use in this chapter represent the following items:

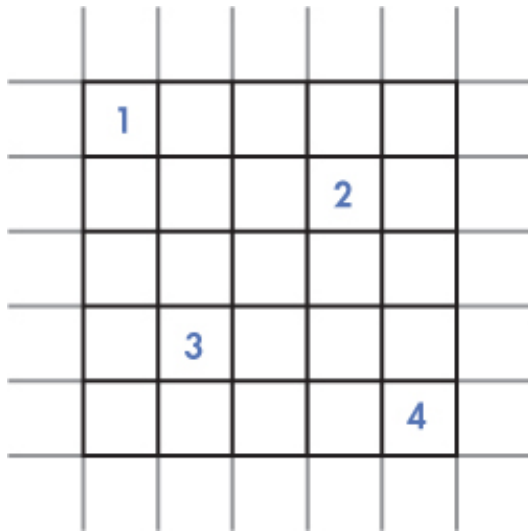


Figure 2-2: Our first simple map

1. Fertilizer
2. Spare oxygen tanks
3. Scissors
4. Toothpaste
5. Emergency blankets
6. Emergency radio

RED ALERT

Make sure your brackets and commas are in the correct places. One reason for putting [Listing 2-1](#) into a program instead of typing it into the shell is so

you can easily make corrections if you make a mistake.

Click **File** ▶ **Save** and save your program as *listing2-1.py*. This program doesn't use Pygame Zero, so we can run it from IDLE. Click **Run** in the menu bar at the top of the window, and then click **Run Module**. You should see the following output in the shell window:

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

It's hard to work out what you're looking at when the list is shown like this, which is why I lined up the numbers in a grid in the program listing. But this shell output is the same map and the same data, so everything is where it should be: it's just being presented in a different way. In [Chapter 3](#), you'll learn how to print this map data so it looks more like the listing we created.

FINDING AN EMERGENCY ITEM

To find out what item is at a particular point in the map, you need to give Python a coordinate to check. The *coordinates* are a combination of the *y* position (from top to bottom) and the *x* position (from left to right), in that order. The *y* position will be the list in `room_map` you want to check (the row in the grid). The *x* position will be the item in that list you want to look at (the column) (see [Figure 2-3](#)). As always, remember that index numbers start at 0.

x = Which list item?

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | | | | |
| 1 | | | | 2 | |
| 2 | | | | | |
| 3 | | 3 | | | |
| 4 | | | | | 4 |

y = Which list?

Figure 2-3: The y-coordinate indicates the list we want to look at. The x-coordinate

indicates the item in that list.

RED ALERT

If you've used coordinates before, you know that you usually put the x-coordinate before the y-coordinate. We're doing the opposite here because it makes the code simpler. If we put x first, we would have to make each list in `room_map` represent a column of the map, from top to bottom, instead of a row, from left to right. That would make the map look wrong in our code: the map would be on its side and a mirror image, which would be very confusing! Just remember that our map coordinates use y and then x.

Let's work through an example: we'll find out what item is at the position marked 2 on our simple map diagram. We need to know the following:

- The 2 is in the second row (from top to bottom), so it's in the second list in `room_map`. The index starts at 0, so we subtract 1 from 2 to get the index number for the y position, which is 1. Use [Figure 2-3](#) to check this index number: the index numbers for the rows are on the left of the grid in red.
- The 2 is in the fourth column (from left to right) of the list. Again, we subtract 1 to get the index number for the x position, which is 3. Use [Figure 2-3](#) to check this index number as well. The index numbers for the columns are shown across the top of the grid in red.

Go to the shell and enter the following `print()` command to view the number in that position on the map:

```
>>> print(room_map[1][3])
```

```
2
```

As expected, the result is the number 2, which happens to be spare oxygen tanks. You've successfully navigated your first map!

TRAINING MISSION #4

Try to predict the output before you enter the following command into the shell:

```
>>> print(room_map[3][1])
```

Refer to the map in [Figure 2-2](#) and your code listing to make your prediction. If you need more help, look at [Figure 2-3](#). Then check your answer by entering the instruction in the shell.

SWAPPING ITEMS IN THE ROOM

You can also change items in the room. Let's check which item is at the top-left position of the map, using the shell again:

```
>>> print(room_map[0][0])
```

```
1
```

The 1 is fertilizer. We don't need fertilizer in the emergency room, so let's change that item to emergency blankets in the map. We'll use a 5 to represent them. Remember how we used an equal sign (=) to change the value of an item in a list? We can do the same to change the number in the map, like this:

```
>>> room_map[0][0] = 5
```

We enter the coordinates and then enter a new number to replace the original number. We can check that the code worked by printing the value at that coordinate again, which was 1 a moment ago. Let's also print `room_map` and confirm that the emergency blankets appear in the correct position:

```
>>> print(room_map[0][0])
```

```
5
```

```
>>> print(room_map)
```

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

Perfect! The emergency blankets are stored in the top-left corner of the room. Item 5 is the first item in the first list.

TRAINING MISSION #5

Space is precious in the emergency room! Replace the toothpaste (4) with an emergency radio (6). You'll need to find the coordinates of the 4 first and then enter the command to change it. Refer to [Figure 2-2](#) and [Figure 2-3](#) if you need more help with the index numbers.

In the *Escape* game, the `room_map` list is used to remember the items in the room the player is currently in. The map stores the number of the object that appears at each position on the map, or a 0 if the floor space is empty. The rooms in the game will be bigger than this 5×5 grid, so the size of the `room_map` will vary depending on the width and height of the room the player is in.

ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ Python lists store words, numbers, or a mixture of both.
- ☐ To see an item in a list, use its index number in square brackets: for example, `print(take_off_checklist[2])`.
- ☐ The `append()` function adds items to the end of a list.
- ☐ The `remove()` function removes items from a list: for example, `spacewalk_checklist.remove("Seal helmet")`.
- ☐ You can use index numbers to delete or insert an item at a particular position in a list.
- ☐ Index numbers start at 0.
- ☐ You can change an item in a list using the equal sign (`=`): for example, `take_off_checklist[3] = "Test comms"`.
- ☐ You can make a list that contains other lists to build a simple map.
- ☐ You can check which item is in your map using coordinates: for example, use `room_map[y coordinate][x coordinate]`.

- ☐ Be sure to use y first and then x for your coordinates. In space, everything is upside down.
- ☐ The coordinates are index numbers, so both start at 0, not 1.
- ☐ You can use `+=` to add an item to a list, or to join two lists.

MISSION DEBRIEF

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

TRAINING MISSION #1

```
>>> take_off_checklist.insert(2, "Check cabin pressure")
```

TRAINING MISSION #2

Print the items from your list using their index numbers:

```
>>> print(spacewalk_checklist[0])
Put on suit
>>> print(spacewalk_checklist[1])
Check oxygen
>>> print(spacewalk_checklist[2])
Seal helmet
>>> print(spacewalk_checklist[3])
Test radio
>>> print(spacewalk_checklist[4])
Open airlock
```

TRAINING MISSION #3

```
>>> docking_checklist = ["Doors to manual", "Rotational lock-on", "Approach and lock"]
>>> flight_manual.append(docking_checklist)
>>> print(flight_manual)
[['Put on suit', 'Seal hatch', 'Check cabin pressure', 'Fasten seatbelt', 'Tell
Mission Control checks are complete'], ['Put on suit', 'Check oxygen', 'Seal helmet',
'Test radio', 'Open airlock'], ['Doors to manual', 'Rotational lock-on', 'Approach
and lock']]
>>> print(flight_manual[2])
['Doors to manual', 'Rotational lock-on', 'Approach and lock']
```

TRAINING MISSION #4

```
3
```

TRAINING MISSION #5

```
>>> room_map[4][4] = 6
>>> print(room_map)
[[1, 0, 0, 0, 0], [0, 0, 0, 2, 0], [0, 0, 0, 0, 0], [0, 3, 0, 0, 0],
[0, 0, 0, 0, 6]]
```


3

REPEAT AFTER ME



Everyone talks about the heroism and glamour of space travel, but some of it is routine, repetitive work. When you're cleaning, gardening in the space station greenhouse, or exercising to keep your strength up, you're following detailed plans designed to keep the team safe and the space station operating. Luckily, robots take care of some of the drudgery, and they never complain about having to repeat themselves.

Whether you're programming robots or building games, the loop is one of your basic programming building blocks. A *loop* is a section of a program that repeats: sometimes it repeats a set number of times, and sometimes it continues until a particular event takes place. Sometimes, you'll even set a loop to keep going forever. In this chapter, you'll learn how to use loops to repeat instructions a certain number of times in your programs. You'll use loops, along with your knowledge of lists, to display a map and draw a 3D room image.

DISPLAYING MAPS WITH LOOPS

In the *Escape* game, we'll use loops extensively. Often, we'll use them to pull information from a list and perform some action on it.

Let's start by using loops to display a text map.

MAKING THE ROOM MAP

We'll make a new map for the example in this chapter and use 1 to represent a wall and 0 to represent a floor space. Our room has a wall all the way around the edge and a pillar near the middle. The pillar is the same as a section of wall, so it's also marked with a 1. I've chosen its position so it looks good when we draw a 3D room later in this chapter. The room doesn't have any other objects, so we won't use any other numbers at this time.

In IDLE, open a new Python program, and enter the code in [Listing 3-1](#), saving it as *listing3-1.py*:

listing3-1.py

```
room_map = [ [1, 1, 1, 1, 1],
              [1, 0, 0, 0, 1],
              [1, 0, 1, 0, 1],
              [1, 0, 0, 0, 1],
              [1, 0, 0, 0, 1],
              [1, 0, 0, 0, 1],
              [1, 1, 1, 1, 1]
            ]
print(room_map)
```

Listing 3-1: Adding the room map data

This program creates a list called `room_map` that contains seven other lists. Each list starts and ends with square brackets and is separated from the next list with a comma. As you learned in [Chapter 2](#), the last list doesn't need a comma after it. Each list represents a row of the map. Run the program by clicking **Run ▶ Run Module** and you should see the following in the shell window:

```
[[1, 1, 1, 1, 1], [1, 0, 0, 0, 1], [1, 0, 1, 0, 1], [1, 0, 0, 0, 1], [1, 0, 0, 0, 1], [1, 0, 0, 0, 1], [1, 1, 1, 1, 1]]
```

As you saw in [Chapter 2](#), printing the map list shows you all the rows run together, which isn't a useful way to view a map. We'll use a loop to display the map in a way that is much easier to read.

DISPLAYING THE MAP WITH A LOOP

To display the map in rows and columns, delete the last line of your program and add the two new lines shown in [Listing 3-2](#). As before, don't type in the grayed-out lines—just use them to find your place in the program. Save your program as *listing3-2.py*.

listing3-2.py

```
--snip--
    [1, 0, 0, 0, 1],
    [1, 1, 1, 1, 1]
]
❶ for y in range(7):
❷     print(room_map[y])
```

Listing 3-2: Using a loop to display the room map

RED ALERT

Remember to place a colon at the end of the first new line! The program won't work without it. The second new line should be indented with four spaces to show Python which instructions you want to repeat. If you add the colon at the end of the `for` line, the spaces are added automatically for you when you press ENTER to go to the next line.

When you run the program again, you should see the following in the shell:

```
[1, 1, 1, 1, 1]
[1, 0, 0, 0, 1]
[1, 0, 1, 0, 1]
[1, 0, 0, 0, 1]
[1, 0, 0, 0, 1]
[1, 0, 0, 0, 1]
[1, 1, 1, 1, 1]
```

That's a more useful way to view a map. Now you can easily see that a wall (represented by 1s) runs all around the edge. So how does the code work? The `for` command ❶ is the engine here. It's a loop command that tells Python to repeat a piece of code a certain number of times. [Listing 3-2](#) tells Python to repeat the `print()` instruction for each item

in our `room_map` list ❷. Each item in `room_map` is a list containing one row of the map, so printing them separately displays our map one row at a time, resulting in this organized display.

Let's break down the code in more detail. We use the `range()` function to create a sequence of numbers. With `range(7)`, we tell Python to generate a sequence of numbers up to, but not including, 7. Why does it leave out the last number? That's just how the `range()` function works! If we give the `range()` function just one number, Python assumes we want to start counting at 0. So `range(7)` creates the sequence of numbers 0, 1, 2, 3, 4, 5, and 6.

Each time the code repeats, the variable in the `for` command takes the next item from the sequence. In this case, the `y` variable takes on the values 0, 1, 2, 3, 4, 5, and 6 in turn. This matches the index numbers in `room_map` perfectly.

I've chosen `y` as the variable name because we're using it to represent which map row we want to display, and the row on the map is referred to as the y-coordinate.

The `print(room_map[y])` command ❷ is indented four spaces, telling Python that this is the chunk of code we want our `for` loop ❶ to repeat.

The first time through the loop, `y` has a value of 0, so `print(room_map[y])` prints the first item in `room_map`, which is a list containing the data for the first row of the map. The second time through, `y` has a value of 1, so `print(room_map[y])` prints the second row. The code repeats until it's printed all seven lists inside `room_map`.

TRAINING MISSION #1

In an emergency situation on the space station, you might need to issue a distress signal. Write a simple program to print the word *Mayday!* three times only, using a loop.

If you're stuck, start with [Listing 3-2](#), used for printing the map. You just need to change what the program prints and how many times it loops the print code.

LOOP THE LOOP

Our map output is getting better, but it still has a couple of limitations. One is that the

commas and brackets make it look cluttered. The other limitation is that we can't do anything with the individual wall panels or spaces in the room. We'll need to be able to handle whatever is at each position in the room separately, so we can display its image correctly. To do that, we'll need to use more loops.

NESTING LOOPS TO GET ROOM COORDINATES

The *listing3-2.py* program uses a loop to extract each row of the map. Now we need to use another loop to examine each position in the row, so we can access the objects there individually. Doing so will enable us to have full control over how the items are displayed.

You just saw that we can repeat a piece of code inside a loop. We can also put a loop inside another loop, which is known as a *nested loop*. To see how this works, we'll first use this technique to print the coordinates for each space in the room. Edit your code to match [Listing 3-3](#):

listing3-3.py

```
--snip--
    [1, 0, 0, 0, 1],
    [1, 1, 1, 1, 1]
]
❶ for y in range(7):
❷   for x in range(5):
❸       print("y=", y, "x=", x)
❹   print()
```

Listing 3-3: Printing the coordinates

RED ALERT

As every astronaut knows, space can be dangerous. Spaces can, too. If the indentation in a loop is wrong, the program won't work correctly. Indent the first `print()` command ❸ with eight spaces so it's part of the inner `x` loop. Make sure the final `print()` instruction ❹ is lined up with the second `for` command ❷ (with four spaces of indentation) so it stays in the outer loop. When you start a new line, Python indents it the same as the previous one, but you can delete the indentation when you no longer need it.

Save your program as *listing3-3.py* and run the program by clicking **Run ▶ Run Module**. You'll see the following output:

```
y= 0 x= 0
y= 0 x= 1
y= 0 x= 2
y= 0 x= 3
y= 0 x= 4

y= 1 x= 0
y= 1 x= 1
y= 1 x= 2
y= 1 x= 3
y= 1 x= 4

y= 2 x= 0
y= 2 x= 1
y= 2 x= 2
--snip--
```

The output continues and ends on `y= 6 x= 4`.

We've set up the `y` loop the same as before so it repeats seven times ❶, once for each number from 0 to 6, putting that value into the `y` variable. This is what is different in our program this time: inside the `y` loop, we start a new `for` loop that uses the `x` variable and gives it a range of five values, from 0 to 4 ❷. The first time through the `y` loop, `y` is 0, and `x` then takes the values 0, 1, 2, 3, and 4 in turn while `y` is 0. The second time through the `y` loop, `y` is 1. We start a new `x` loop, and it takes the values 0, 1, 2, 3, and 4 again while `y` is 1. This looping keeps going until `y` is 6 and `x` is 4.

You can see how the loops work when you look at the program's output: inside the `x` loop, we print the values for `y` and `x` each time the `x` loop repeats ❸. When the `x` loop finishes, we print a blank line ❹ before the next repeat of the `y` loop. We do this by leaving the `print()` function's parentheses empty. The blank line breaks up where the `y` loop repeats, and the output shows you what the values of `x` and `y` are each time through the inner `x` loop. As you can see, this program outputs the `y`- and `x`-coordinates of every position in the room.

TIP

We've used the variable names `y` and `x` in our loops, but those variable names don't affect the way the program runs. You could call them `sausages` and `eggs`, and the program would work just the same. It wouldn't be as easy to understand, though. Because we're getting `x`- and `y`-coordinates, it makes sense to use `x` and `y` for our variable names.

CLEANING UP THE MAP

We'll use the coordinates in the loops to print our map without any brackets and commas. Edit your program to change the inner nested loop as shown in [Listing 3-4](#):

listing3-4.py

```
--snip--
for y in range(7):
    for x in range(5):
        print(room_map[y][x], end="")
    print()
```

Listing 3-4: Tidying up the map display

Save your program as *listing3-4.py* and run the program by clicking **Run ▶ Run Module**. You should see the following in the shell:

```
11111
10001
10101
10001
10001
10001
10001
11111
```

That map is much cleaner and easier to understand. It works by going through the coordinates in the same way the program in [Listing 3-3](#) did. It takes each row in turn using the `y` loop, and then uses the `x` loop to get each position in that row. This time,

instead of printing the coordinates, we look at what is in the `room_map` at each position, and print that. As you learned in [Chapter 2](#), you can pull any item out of the map using coordinates in the form `room_map[y coordinate][x coordinate]`.

The way we've formatted the output means the map resembles the room: we put all the numbers from one row together, and only start a new line on the screen when we start a new row of the map (a new repeat of the `y` loop).

The `print()` instruction inside the `x` loop finishes with `end=""` (with no space between the quote marks) to stop it from starting a new line after each number. Otherwise, by default, the `print()` function would end each piece of output by adding a code that starts a new line. But instead, we tell it to put nothing ("") at the end. As a result, all the items from one complete run of the `x` loop (from 0 to 4) appear on the same line.

After each row is printed, we use an empty `print()` command to start a new line. Because we indent this command with only four spaces, it belongs to the `y` loop and is not part of the code that repeats in the `x` loop. That means it runs only once each time through the `y` loop, after the `x` loop has finished printing a row of numbers.

TRAINING MISSION #2

The final `print()` command is indented using four spaces. See what happens when you indent it eight spaces, and then see what happens if you don't indent it at all. In each case, record how many times it runs and how the indentation changes the output.

DISPLAYING A 3D ROOM IMAGE

You now know enough about maps to display a 3D room image. In [Chapter 1](#), you learned how to use Pygame Zero to place images on the screen. Let's combine that knowledge with your newfound skills in getting data from the `room_map`, so we can display our map with images instead of 0s and 1s.

Click **File** ▶ **New File** to start a new file in Python, and then enter the code in [Listing 3-5](#). You can copy the `room_map` data from your most recent program for this chapter.

[listing3-5.py](#)

```
room_map = [ [1, 1, 1, 1, 1],
              [1, 0, 0, 0, 1],
              [1, 0, 1, 0, 1],
              [1, 0, 0, 0, 1],
              [1, 0, 0, 0, 1],
              [1, 0, 0, 0, 1],
              [1, 1, 1, 1, 1]
            ]
```

❶ WIDTH = 800 # window size

❷ HEIGHT = 800

```
top_left_x = 100
```

```
top_left_y = 150
```

❸ DEMO_OBJECTS = [images.floor, images.pillar]

```
room_height = 7
```

```
room_width = 5
```

❹ def draw():

```
    for y in range(room_height):
```

```
        for x in range(room_width):
```

❺ image_to_draw = DEMO_OBJECTS[room_map[y][x]]

❻ screen.blit(image_to_draw,
 (top_left_x + (x*30),
 top_left_y + (y*30) - image_to_draw.get_height()))

Listing 3-5: Code for displaying the room in 3D

Save the program as *listing3-5.py*. You need to save it in your *escape* folder, because the program will use the files inside the *images* folder stored there. Don't save your file *inside* the *images* folder: your file should be alongside it instead. If you haven't downloaded the *Escape* game files yet, see “[Downloading the Game Files](#)” on [page 7](#) for download instructions.

The *listing3-5.py* program uses Pygame Zero, so you need to go to the command line and enter the instruction `pgzrun listing3-5.py` to run the program. See “[Running the Game](#)” on [page 9](#) for advice on running programs that use Pygame Zero, including the final *Escape* game.

The *listing3-5.py* program uses the *Escape* game's image files to create an image of a room. **Figure 3-1** shows the room with its single pillar. The *Escape* game uses a simplified 3D perspective where we can see the front and top surfaces of an object. Objects at the front and back of the room are drawn at the same size.

When you created the spacewalk simulator in **Chapter 1**, you saw how the order in which objects are drawn determines which ones are in front of the others. In the *Escape* game and **Listing 3-5**, the objects are drawn from the back of the room to the front, enabling us to create the 3D effect. Objects nearer to the viewer (sitting at their computer) appear to be in front of those at the back of the room.

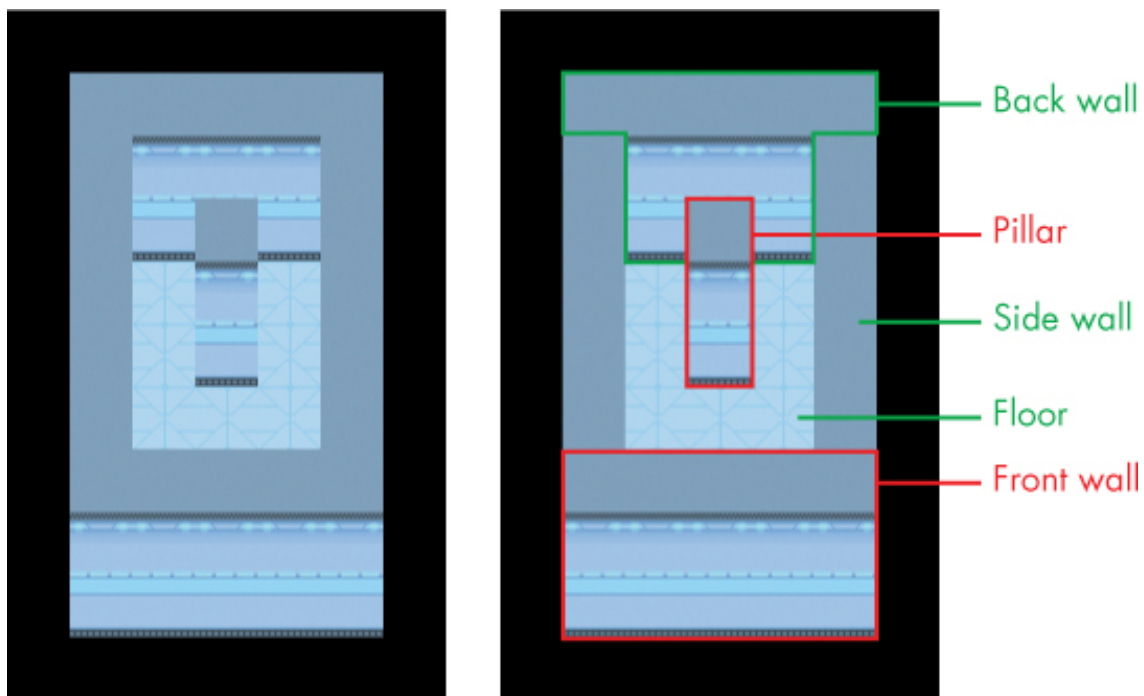


Figure 3-1: Your first 3D room (left) and the same room with the parts labeled (right)

UNDERSTANDING HOW THE ROOM IS DRAWN

Now let's look at how the *listing3-5.py* program works. Much of the program will be familiar to you from **Chapters 1** and **2**. The `WIDTH` ❶ and `HEIGHT` ❷ variables hold the size of the window, and we use the `draw()` function to tell Pygame Zero what to draw onscreen ❸. The `y` and `x` loops come from **Listing 3-4** earlier in this chapter and give us coordinates for each space in the room.

Instead of using numbers in the `range()` functions to tell Python how many times to repeat our `y` and `x` loops, we're using the new variables `room_height` and `room_width`. These variables store the size of our room map and tell Python how many times to repeat the loops. For example, if we changed the `room_height` variable to 10, the `y` loop would repeat 10 times and work through 10 rows of the map. The `room_width` variable controls how

many times the `x` loop repeats in the same way, so we can display wider rooms.

RED ALERT

If you use room widths and heights that are bigger than the actual `room_map` data, you'll cause an error.

The `listing3-5.py` program uses two images from the `images` folder: a floor tile (with the filename `floor.png`) and a wall pillar (called `pillar.png`), as shown in [Figure 3-2](#). A *PNG* (Portable Network Graphics) is a type of image file that Pygame Zero can use. PNG enables parts of the image to be see-through, which is important for our 3D game perspective. Otherwise, we wouldn't be able to see the background scenery through the gaps in a plant, for example, and the astronaut would look like they had a square halo around them.

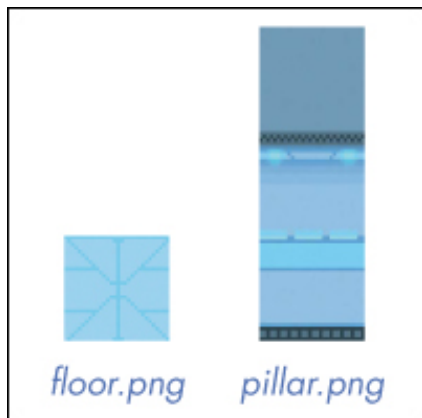


Figure 3-2: The images used to make your first 3D room

Inside the `draw()` function ④, we use `y` and `x` loops to look at each position in the room map in turn. As you saw earlier, we can find the number at each position in the map by accessing `room_map[y][x]`. In this map, that number will be either 1 for a wall pillar or 0 for an empty floor space. Instead of printing the number onscreen, as we did before, we use the number to look up an image of the item in the `DEMO_OBJECTS` list ⑤. That list contains our two images ③: the floor tile is at index position 0, and the wall pillar is at index position 1. If the `room_map` contains a 1 at the position we're looking at, for example, we'll take the item at list index 1 in the `DEMO_OBJECTS` list, which is the wall pillar image. We store that image in the variable `image_to_draw` ⑥.

We then use `screen.blit()` to draw this image onscreen, giving it the `x` and `y` coordinate of the pixel on the screen where we want to draw it ⑦. This instruction extends over three

lines to make it easier to read. The amount of indentation on the second and third lines doesn't matter, because these lines are surrounded by the `screen.blit()` parentheses.

WORKING OUT WHERE TO DRAW EACH ITEM

To figure out where to draw each image that makes up the room, we need to do a calculation at ❹. We'll look at how that calculation works, but before we do, I'll explain how the space station was designed. All the images are designed to fit a grid. The units we use for measuring images on a computer are called *pixels* and are the size of the smallest dot you can see on your screen. We'll call each square of the grid a *tile*. Each tile is 30 pixels across the screen and 30 pixels down the screen. It's the same size as one floor tile. We position objects in terms of tiles, so a chair might be 4 tiles down and 4 tiles across, measured from the top-left corner.

Figure 3-3 shows the room we've just created with a grid laid on top. Each floor tile and pillar is one tile wide. The pillar is tall, so it covers three tile spaces: the front surface of the wall pillar is two tiles tall, and the top surface of the pillar covers another tile space.

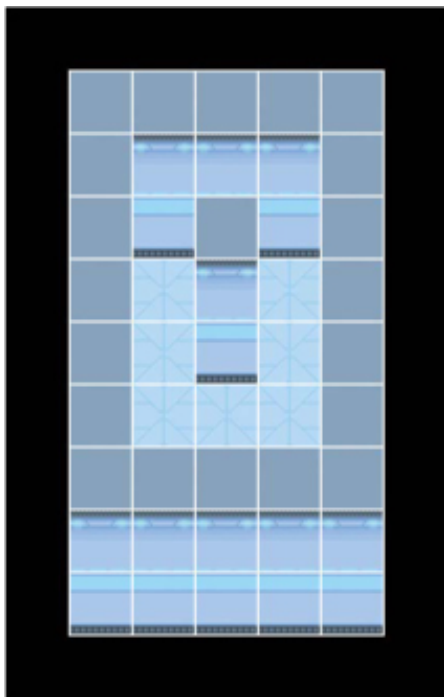


Figure 3-3: The tile grid overlaid on your first room

The `top_left_x` and `top_left_y` variables store the coordinates at which we want to start drawing the first image of the room in the window. We never change these variables in this chapter. I've chosen to start drawing where `x` is 100 and `y` is 150 so we have a bit of a border around the room image.

To work out where to draw a piece of wall or floor, we need to convert our map positions (which range from 0 to 4 in the `x` direction, for example) into pixel positions in

the window.

Each tile space is 30 pixels square, so we multiply the `x` loop number by 30 and add it to the `top_left_x` position to get the x-coordinate for the image. In Python, the `*` symbol is for multiplication. The `top_left_x` value is 100, so the first image is drawn at $100 + (0 * 30)$, which is 100. The second image is drawn at $100 + (1 * 30)$, which is 130, one tile position to the right of the first. The third image is drawn at $100 + (2 * 30)$, which is 160. These positions ensure that the images sit perfectly side by side.

The `y` position is calculated in a similar way. We use `top_left_y` as the starting position vertically and add `y * 30` to it to make the images join together precisely. The difference is that we subtract the height of the image we're drawing, so we ensure that the images align at the same point at the bottom. As a result, tall objects can rise out of a tile space and obscure any scenery or floor tiles behind them, making the room display look three-dimensional. If we didn't align the images at the bottom, they would all align at the top, which would destroy the 3D effect. The second and third rows of floor tiles would cover up the front surface of the back wall, for example.

TRAINING MISSION #3

Now that you know how to display a 3D room, try to adjust the map to change the room layout, adding new pillars or floor spaces. You can edit the `room_map` data to add new rows or columns to the map. Remember to change the `room_height` and `room_width` variables too.

Perhaps try making a room with more rows and adding a doorway by replacing the 1s used for pillars with 0s. In the final *Escape* game, each doorway will be three spaces. For best results, design rooms with an odd width and height so you can center the door in the wall.

Figure 3-4 shows a room I designed with a width and height of 9. You can try copying my design if you like. I've added a grid to make it easier to work out the data for the `room_map` list. The wall pillars rise two tiles out of the floor, so the grid shown is 11 tiles high. Look at the bottoms of the pillars, not the tops, to work out where to position them. See the end of the chapter for the code to make this room.

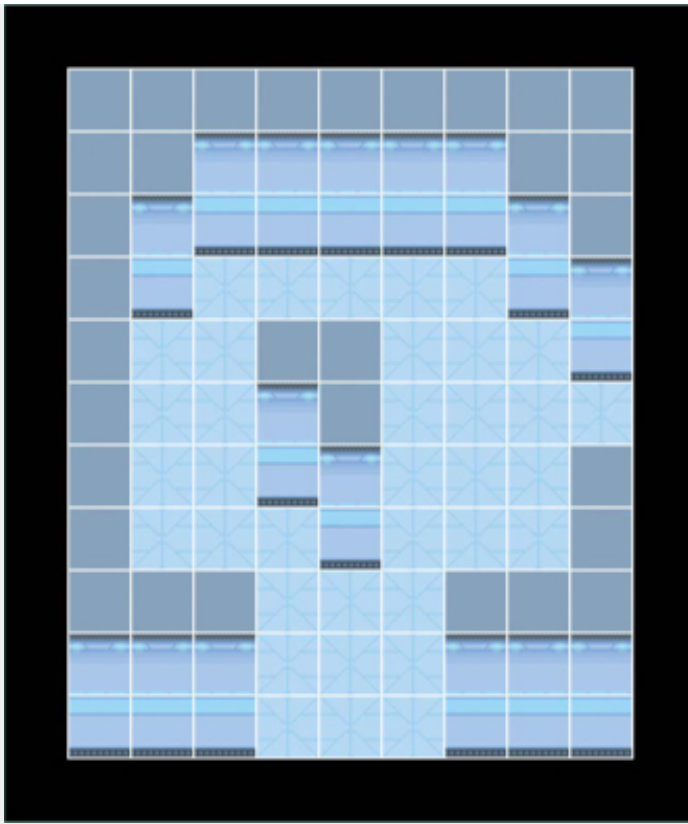


Figure 3-4: One possible new room design

In the real *Escape* game, the tall wall pillars will only be used at the edges of the rooms. They can look a bit odd in the middle of the room, especially if they touch the back wall. When we add shadows to the game later in the book, objects in the middle of the room won't look like they're floating in space, which is a risk of this way of simulating a 3D perspective.

ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ The `for` loop repeats a section of code a set number of times.
- ☐ The `range()` function creates a sequence of numbers.
- ☐ You can use `range()` to tell a `for` loop how many times to repeat.
- ☐ The colon at the end of the `for` line is essential.
- ☐ To show Python which lines to repeat in the loop, indent the lines using four spaces.
- ☐ A loop inside another loop is called a *nested loop*.

- ☐ Images are aligned at the bottom to create a 3D illusion with tall objects rising up from the floor.
- ☐ The `room_height` and `room_width` variables store the room size in *Escape* and are used to set up the loop that displays the room.

MISSION DEBRIEF

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

TRAINING MISSION #1

```
for y in range(3):  
    print("Mayday!")
```

TRAINING MISSION #2

If you don't indent the final `print()` command, it won't repeat; instead, the final `print()` command will run only once after both loops have finished. As a result, all the output will be on one line because the program doesn't start a new line between rows.

If you indent the command with eight spaces, it becomes part of the `x` loop. That means the `print()` command runs each time a number is printed, so every number is on a new line.

TRAINING MISSION #3

Here is the data for the room design in Figure 3-4. You also need to change `room_height` to 9 and `room_width` to 9.

```
room_map = [  
    [1, 1, 1, 1, 1, 1, 1, 1, 1],  
    [1, 1, 0, 0, 0, 0, 0, 1, 1],  
    [1, 0, 0, 0, 0, 0, 0, 0, 1],  
    [1, 0, 0, 0, 0, 0, 0, 0, 0],  
    [1, 0, 0, 1, 1, 0, 0, 0, 0],  
    [1, 0, 0, 0, 1, 0, 0, 0, 0],  
    [1, 0, 0, 0, 0, 0, 0, 0, 1],  
    [1, 0, 0, 0, 0, 0, 0, 0, 1],  
    [1, 1, 1, 0, 0, 0, 1, 1, 1]  
]
```

4

CREATING THE SPACE STATION



In this chapter, you'll build the map for your space station on Mars. Using the simple *Explorer* code that you'll add in this chapter, you'll be able to look at the walls of each room and start to find your bearings. We'll use lists, loops, and the techniques you learned in Chapters 1, 2, and 3 to create the map data and display the room in 3D.

AUTOMATING THE MAP MAKING PROCESS

The problem with our current `room_map` data is that there's a lot of it. The *Escape* game includes 50 locations. If you had to enter `room_map` data for every location, it would take ages and be hugely inefficient. As an example, if each room consisted of 9×9 tiles, we would have 81 data items per room, or 4,050 data items in total. Just the room data would take up 10 pages of this book.

Much of that data is repeated: 0s mark the floor and exits, and 1s mark the walls at the edges. You know from Chapter 3 that we can use loops to efficiently manage repetition. We can use that knowledge to make a program that will generate the `room_map` data automatically when we give it certain information, such as the room size and the location of the exits.

HOW THE AUTOMATIC MAP MAKER WORKS

The *Escape* program will work like this: when the player visits a room, our code will take the data for that room (its size and exit positions) and convert it into the `room_map` data. The `room_map` data will include columns and rows that represent the floor, walls around the edge, and gaps where the exits should be. Eventually, we'll use the `room_map` data to draw the room with the floor and walls in the correct place.

Figure 4-1 shows the map for the space station. I'll refer to each location as a room, although numbers 1 to 25 are sectors on the planet surface within the station compound, similar to a garden. Numbers 26 to 50 are rooms inside the space station.

The indoor layout is a simple maze with many corridors, dead-ends, and rooms to explore. When you make your own maps, try to create winding paths and corners to explore, even if the map isn't very big. Be sure to reward players for their exploration by placing a useful or appealing item at the end of each corridor. Players also often feel more comfortable travelling from left to right as they explore a game world, so the player's character will start on the left of the map in room 31.

Outside, players can walk anywhere, but a fence will stop them from leaving the station compound (or wandering off the game map). Due to the claustrophobic atmosphere inside the space station, players will experience a sense of freedom outside with space to roam.

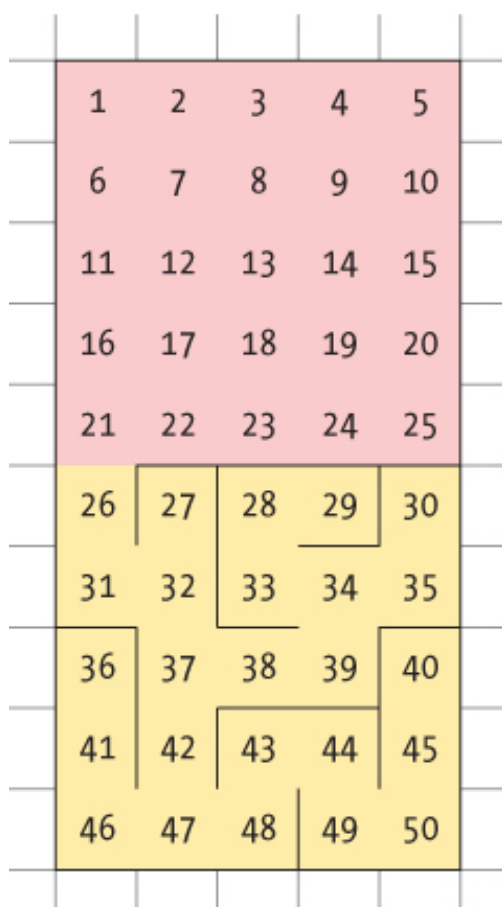


Figure 4-1: The space station map

When you're playing the final *Escape* game, you can refer to this map, but you might find it more enjoyable to explore without a map or to make your own. This map doesn't show where the doors are, which in the final game will stop players from accessing some parts of the map until they find the right key cards.

CREATING THE MAP DATA

Let's create the map data. The rooms in our space station will all join up, so we only need to store the location of an exit from one side of the wall. For instance, an exit on the right of room 31 and an exit on the left of room 32 would be the same doorway connecting the two rooms. We don't need to specify that exit for both rooms. For each room in the map, we'll store whether it has an exit at the top or on the right. The program can work out on its own whether an exit exists at the bottom or on the left (as I'll explain shortly). This approach also ensures that the map is consistent and no exits seem to vanish after you walk through them. If you can go one way through an exit, you can always go back the other way.

Each room in the map needs the following data:

- A short description of the room.
- Height in tiles, which is the size of the room from top to bottom on the screen. (This has nothing to do with the distance from floor to ceiling.)
- Width in tiles, which is the size of the room from left to right across the screen.
- Whether or not there is an exit at the top (`True` or `False`).
- Whether or not there is an exit on the right (`True` or `False`).

TIP

`True` and `False` are known as *Boolean values*. In Python, these values must start with a capital letter, and they don't need quotes around them, because they're not strings.

We call the unit we use to measure the room size a *tile* because it's the same size as a floor tile. As you learned in [Chapter 3](#), a tile will be our basic unit of measurement for all objects. For instance, a single object in the room, such as a chair or a cabinet, will often be the size of one tile. In [Chapter 3](#) (see [Figure 3-1](#) and [Listing 3-5](#)), we made a room map that had seven rows with five list items in each row, so that room would be seven tiles high and five tiles wide.

Having rooms of different sizes adds variety to the map: some rooms can be narrow like corridors, and some can be expansive like community rooms. To fit in our game window, the maximum size of a room is 15 tiles high by 25 tiles wide. Large rooms or rooms with lots of objects in them might run more slowly on older computers, though.

Here's an example of the data for room 26: it's a narrow room 13 tiles high and 5 tiles wide with an exit at the top but none to the right (see the map in [Figure 4-1](#)).

```
["The airlock", 13, 5, True, False]
```

We give the room a name (or description), numbers for the height and width respectively, and `True` and `False` values for whether the top and right edges have an exit. In this game, each wall can have only one exit, and that exit will be automatically positioned in the middle of the wall.

When the program makes the `room_map` data for room 27 next door, it will check room 26 to see whether it has an exit on the right. Because room 26 has no exit on the right, the program will know that room 27 has no left exit.

We'll store the lists of data for each room in a list called `GAME_MAP`.

WRITING THE GAME_MAP CODE

Click **File** ▶ **New File** to start a new file in Python. Enter the code from [Listing 4-1](#) to start building the space station. Save your listing as *listing4-1.py*.

TIP

Remember to save your work regularly when you're typing a long program. As in many applications, you can press CTRL-S to save in IDLE.

```
# Escape - A Python Adventure
# by Sean McManus / www.sean.co.uk
# Typed in by PUT YOUR NAME HERE

import time, random, math

#####

## VARIABLES ##

#####

WIDTH = 800 # window size
HEIGHT = 800

#PLAYER variables
❶ PLAYER_NAME = "Sean" # change this to your name!
FRIEND1_NAME = "Karen" # change this to a friend's name!
FRIEND2_NAME = "Leo" # change this to another friend's name!
current_room = 31 # start room = 31

❷ top_left_x = 100
top_left_y = 150

❸ DEMO_OBJECTS = [images.floor, images.pillar, images.soil]

#####

## MAP ##

#####

❹ MAP_WIDTH = 5
MAP_HEIGHT = 10
MAP_SIZE = MAP_WIDTH * MAP_HEIGHT

❺ GAME_MAP = [ ["Room 0 - where unused objects are kept", 0, 0, False, False] ]

outdoor_rooms = range(1, 26)
❻ for planetsectors in range(1, 26): #rooms 1 to 25 are generated here
    GAME_MAP.append( ["The dusty planet surface", 13, 13, True, True] )
```

```

7 GAME_MAP += [
    #["Room name", height, width, Top exit?, Right exit?]
    ["The airlock", 13, 5, True, False], # room 26
    ["The engineering lab", 13, 13, False, False], # room 27
    ["Poodle Mission Control", 9, 13, False, True], # room 28
    ["The viewing gallery", 9, 15, False, False], # room 29
    ["The crew's bathroom", 5, 5, False, False], # room 30
    ["The airlock entry bay", 7, 11, True, True], # room 31
    ["Left elbow room", 9, 7, True, False], # room 32
    ["Right elbow room", 7, 13, True, True], # room 33
    ["The science lab", 13, 13, False, True], # room 34
    ["The greenhouse", 13, 13, True, False], # room 35
    [PLAYER_NAME + "'s sleeping quarters", 9, 11, False, False], # room 36
    ["West corridor", 15, 5, True, True], # room 37
    ["The briefing room", 7, 13, False, True], # room 38
    ["The crew's community room", 11, 13, True, False], # room 39
    ["Main Mission Control", 14, 14, False, False], # room 40
    ["The sick bay", 12, 7, True, False], # room 41
    ["West corridor", 9, 7, True, False], # room 42
    ["Utilities control room", 9, 9, False, True], # room 43
    ["Systems engineering bay", 9, 11, False, False], # room 44
    ["Security portal to Mission Control", 7, 7, True, False], # room 45
8    [FRIEND1_NAME + "'s sleeping quarters", 9, 11, True, True], # room 46
    [FRIEND2_NAME + "'s sleeping quarters", 9, 11, True, True], # room 47
    ["The pipeworks", 13, 11, True, False], # room 48
    ["The chief scientist's office", 9, 7, True, True], # room 49
    ["The robot workshop", 9, 11, True, False] # room 50
]

# simple sanity check on map above to check data entry
9 assert len(GAME_MAP)-1 == MAP_SIZE, "Map size and GAME_MAP don't match"

```

Listing 4-1: The GAME_MAP data

Let's take a closer look at this code for setting out the room map data. Keep in mind that as we build the *Escape* game, we'll keep adding to the program. To help you find your way around the program, I'll mark the different sections with headings like this:

```
#####  
## VARIABLES ##  
#####
```

The # symbol marks a comment and tells Python to ignore anything after it on the same line, so the game will work with or without these comments. The comments will make it easier to figure out where you are in the code and where you need to add new instructions as the program gets bigger. I've drawn boxes using the comment symbols to make the headings stand out as you scroll through the program code.

Three astronauts are based on the space station, and you can personalize their names in the code ❶. Change the `PLAYER_NAME` to your own, and add the names of two friends for the `FRIEND1_NAME` and `FRIEND2_NAME` variables. Throughout the code, we'll use these variables wherever we need to use the name of one of your friends: for example, each astronaut has their own sleeping quarters. We need to set up these variables now because we'll use them to set up some of the room descriptions later in this program. Who will you take with you to Mars?

The program also sets up some variables we'll need at the end of this chapter to draw our room: the `top_left_x` and `top_left_y` variables ❷ specify where to start drawing the room; and the `DEMO_OBJECTS` list contains the images to use ❸.

First, we set up variables to contain the height, width, and overall size of the map in tiles ❹. We create the `GAME_MAP` list ❺ and give it the data for room 0: this room is for storing items that aren't in the game yet because the player hasn't discovered or created them. It's not a real room the player can visit.

We then use a loop ❻ to add the same data for each of the 25 planet surface rooms that make up the grounds of the compound. The `range(1, 26)` function is used to repeat 25 times. The first number is the one we want to start at, and the second is the number we want to finish at plus one (`range()` doesn't include the last number you give it, remember). Each time through the loop, the program adds the same data to the end of `GAME_MAP` because all the planet surface "rooms" are the same size and have exits in every direction. The data for every surface room looks like this:

```
["The dusty planet surface", 13, 13, True, True]
```

When this loop finishes, `GAME_MAP` will include room 0 and also have the same "dusty planet surface" data for rooms 1 to 25. We also set up the `outdoor_rooms` range to store the

room numbers 1 to 25. We'll use this range when we need to check whether a room is inside or outside the space station.

Finally, we add rooms 26 to 50 to `GAME_MAP` ⑦. We do this by using `+=` to add a new list to the end of `GAME_MAP`. That new list includes the data for the remaining rooms. Each of these rooms will be different, so we need to enter the data for them separately. You saw the information for room 26 earlier: the data contains the room name, its height and width, and whether it has exits at the top and the right. Each piece of room data is a list, so it has square brackets at the start and end. At the end of each piece of room data (except the last one), we must use a comma to separate it from the next one. I've also put the room number in a comment at the end of each line to help keep track of the room numbers. These comments will be helpful as you develop the game. It's good practice to annotate your code like this so you can understand it when you revisit it.

Rooms 46 and 47 add the variables `FRIEND1_NAME` and `FRIEND2_NAME` to the room description, so you have two rooms called something like "Karen's sleeping quarters," using your friends' names ⑧. As well as using the `+` symbol to add numbers and combine lists, you can also use it to combine strings.

At the end of *listing4-1.py*, we perform a simple check using `assert()` to make sure the map data makes sense ⑨. We check whether the length of the `GAME_MAP` (the number of rooms in the map data) is the same as the size of the map, which we calculated at ④ by multiplying its width by its height. If it's not, it means we're missing some data or have too much.

We have to subtract 1 from the length of `GAME_MAP` because it also includes room 0, which we didn't include when we calculated the map size. This check won't catch all errors, but it can tell you whether you missed a line of the map data when entering it. Wherever possible, I'll try to include simple tests like this to help you check for any errors as you enter the program code.

TESTING AND DEBUGGING THE CODE

Run *listing4-1.py* by clicking **Run ▶ Run Module** or press F5 (the keyboard shortcut). Nothing much should happen. The shell window should just display a message that says "RESTART:" together with your filename. The reason is that all we've asked the program to do is set up some variables and a list, so there is nothing to see. But if you made a mistake entering the listing, you might also see a red error message in the shell window. If you do get an error, double-check the following details:

- Are the quote marks in the right place? Strings are in green in the Python program window, so look for large areas of green, which suggest you didn't close your string. If room descriptions are in black, you didn't open the string. Both indicate a missing quote mark.
- Are you using the correct brackets and parentheses in the proper places? In this listing, square brackets surround list items, and parentheses (curved brackets) are used for functions, such as `range()` and `append()`. Curly brackets `{...}` are not used at all.
- Are you missing any brackets or parentheses? A simple way to check is to count the number of opening and closing brackets and parentheses. Every opening bracket or parenthesis should have a closing bracket or parenthesis of the same shape.
- You have to close brackets and parentheses in the reverse order of how you opened them. If you have an opening parenthesis and then an opening square bracket, you must close them first with a closing square bracket and then a closing parenthesis. This format is correct: `([...])`. This format is wrong: `([...)]`.
- Are your commas in the correct place? Remember that each list for a room in `GAME_MAP` must have a comma after the closing square bracket to separate it from the next room's data (except for the last room).

TIP

Why not ask a friend to help you build the game? Programmers often work in pairs to help each other with ideas and, perhaps most importantly, have two pairs of eyes checking everything. You can take turns typing too!

GENERATING ROOMS FROM THE DATA

Now the space station map is stored in our `GAME_MAP` list. The next step is to add the function that takes the data for the current room from `GAME_MAP` and expands it into the `room_map` list that the *Escape* game will use to see what's at each position in the room. The `room_map` list always stores information about the room the player is currently in. When the player enters a different room, we replace the data in `room_map` with the map of the new room. Later in the book, we'll add scenery and props to the `room_map`, so the player

has items to interact with too.

The `room_map` data is made by a function we'll create called `generate_map()`, shown in [Listing 4-2](#).

Add the code in [Listing 4-2](#) to the end of [Listing 4-1](#). The grayed out code shows you where [Listing 4-1](#) ends. Make sure all the indentation is correct. The indentation determines whether code belongs to the `get_floor_type()` OR `generate_map()` function, and some code is indented further to tell Python which `if` or `for` command it belongs to.

Save your program as *listing4-2.py* and click **Run ▶ Run Module** to run it and check for any error messages in the shell.

RED ALERT

Don't start a new program with the code in [Listing 4-2](#): make sure you add [Listing 4-2](#) to the end of [Listing 4-1](#). As you follow along in this book, you'll increasingly add to your existing program to build the Escape game.

listing4-2.py

--snip--

simple sanity check on map above to check data entry

assert len(GAME_MAP)-1 == MAP_SIZE, "Map size and GAME_MAP don't match"

#####

MAKE MAP

#####

❶ `def get_floor_type():`

`if current_room in outdoor_rooms:`

`return 2 # soil`

`else:`

`return 0 # tiled floor`

`def generate_map():`


```

# This function makes the map for the current room,
# using room data, scenery data and prop data.
global room_map, room_width, room_height, room_name, hazard_map
global top_left_x, top_left_y, wall_transparency_frame

❷ room_data = GAME_MAP[current_room]
room_name = room_data[0]
room_height = room_data[1]
room_width = room_data[2]

❸ floor_type = get_floor_type()
if current_room in range(1, 21):
    bottom_edge = 2 #soil
    side_edge = 2 #soil
if current_room in range(21, 26):
    bottom_edge = 1 #wall
    side_edge = 2 #soil
if current_room > 25:
    bottom_edge = 1 #wall
    side_edge = 1 #wall

# Create top line of room map.
❹ room_map=[[side_edge] * room_width]
# Add middle lines of room map (wall, floor to fill width, wall).
❺ for y in range(room_height - 2):
    room_map.append([side_edge]
                    + [floor_type]*(room_width - 2) + [side_edge])
# Add bottom line of room map.
❻ room_map.append([bottom_edge] * room_width)

# Add doorways.
❼ middle_row = int(room_height / 2)
middle_column = int(room_width / 2)

❽ if room_data[4]: # If exit at right of this room
    room_map[middle_row][room_width - 1] = floor_type
    room_map[middle_row+1][room_width - 1] = floor_type
    room_map[middle_row-1][room_width - 1] = floor_type

❾ if current_room % MAP_WIDTH != 1: # If room is not on left of map

```

```
room_to_left = GAME_MAP[current_room - 1]
```

```
# If room on the left has a right exit, add left exit in this room
```

```
if room_to_left[4]:
```

```
    room_map[middle_row][0] = floor_type
```

```
    room_map[middle_row + 1][0] = floor_type
```

```
    room_map[middle_row - 1][0] = floor_type
```

```
⑩ if room_data[3]: # If exit at top of this room
```

```
    room_map[0][middle_column] = floor_type
```

```
    room_map[0][middle_column + 1] = floor_type
```

```
    room_map[0][middle_column - 1] = floor_type
```

```
if current_room <= MAP_SIZE - MAP_WIDTH: # If room is not on bottom row
```

```
    room_below = GAME_MAP[current_room+MAP_WIDTH]
```

```
# If room below has a top exit, add exit at bottom of this one
```

```
if room_below[3]:
```

```
    room_map[room_height-1][middle_column] = floor_type
```

```
    room_map[room_height-1][middle_column + 1] = floor_type
```

```
    room_map[room_height-1][middle_column - 1] = floor_type
```

Listing 4-2: Generating the `room_map` data

You can build the *Escape* game and even make your own game maps without understanding how the `room_map` code works. But if you're curious, read on and I'll walk you through it.

HOW THE ROOM GENERATING CODE WORKS

Let's start with a reminder of what we want the `generate_map()` function to do. Given the height and width of a room, and the location of the exits, we want it to generate a room map, which might look something like this:

```
[  
[1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1],  
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],  
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]  
]
```

This is room number 31 on the map, the room the player starts the game in. It's 7 tiles high and 11 tiles wide, and it has an exit at the top and right. The floor spaces (and exits in the wall) are marked with a 0. The walls around the room are marked with a 1. [Figure 4-2](#) shows the same room in a grid layout, with the index numbers for the lists shown at the top and on the left.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 4-2: A grid representing room 31; the 1s are wall pillars, and the 0s are empty floor spaces.

The number of the room the player is currently in is stored in the `current_room` variable, which you set up in the `VARIABLES` section of your program (see [Listing 4-1](#)). The `generate_map()` function starts by collecting the room data for the current room from the `GAME_MAP` ❷ and putting it into a list called `room_data`.

If you cast your mind back to when we set up `GAME_MAP`, the information in the `room_data` list will now look similar to this:

```
["The airlock", 13, 5, True, False]
```

This list format allows us to set up the `room_name` by taking the first element from this list at index 0. We can find the room's height at index 1 and width at index 2 by taking the next elements. The `generate_map()` function stores the height and width information in the `room_height` and `room_width` variables.

CREATING THE BASIC ROOM SHAPE

The next step is to set the materials we'll use to build the rooms and create the basic

room shape using them. We'll add exits later. We'll use three elements for each room:

- The *floor type*, which is stored in the variable `floor_type`. Inside the space station, we use floor tiles (represented by 0 in `room_map`), and outside we use soil (represented by 2 in `room_map`).
- The *edge type*, which appears in each space at the edge of the room. For an inside room, this is a wall pillar, represented by 1. For an outside room, this is the soil.
- The *bottom edge type*, which is a wall inside the station and usually soil outside. The bottom row of the outside compound, where it meets the space station, is a special case because the station wall is visible here, so the `bottom_edge` type is a wall pillar (see Figure 4-3).

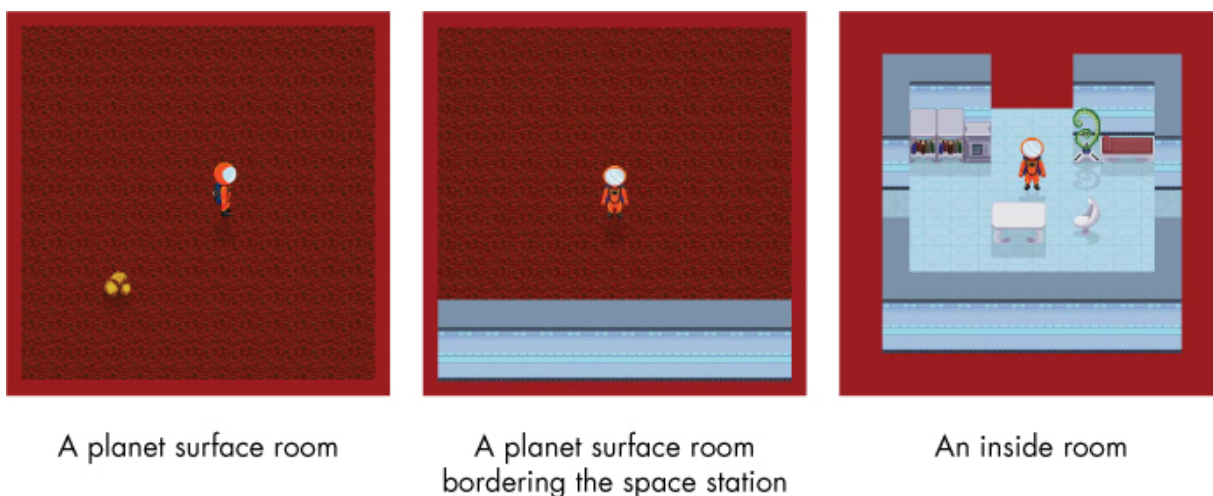


Figure 4-3: Different materials are used for the edges and bottom edge of the room, depending on where the room is in the space station compound. (Note that the astronaut and additional scenery won't be in your game yet.)

We use a function called `get_floor_type()` ❶ to find out the correct floor type for the room. Functions can send information back to other parts of the program using the `return` instruction, as you can see in this function. The `get_floor_type()` function checks whether the `current_room` value is in the `outdoor_rooms` range. If so, the function returns the number 2, which represents Martian soil. Otherwise, it returns the number 0, which represents a tiled floor. This check is in a separate function so other parts of the program can use it too. The `generate_map()` function puts the number that `get_floor_type()` returns into the `floor_type` variable. Using one instruction ❸, `generate_map()` sets up the `floor_type` variable to be equal to whatever `get_floor_type()` sends back, and it tells the `get_floor_type()` function to run now too.

The `generate_map()` function also sets up variables for the `bottom_edge` and `side_edge`. These

variables store the type of material that will be used to make the edges of the room, as shown in [Figure 4-3](#). The side edge material is used for the top, left, and right sides, and the bottom edge material is for the bottom edge. If the room number is between 1 and 20 inclusive, it's a regular planet surface room. The bottom and edge are soil in that case. If the room number is between 21 and 25, it's a planet surface room that touches the space station at the bottom. This is a special case: the side edge material is soil, but the bottom edge is made of wall pillars. If the room number is higher than 25, the side and bottom edges are made of wall pillars because it's an inside room. (You can check that these room numbers make sense in [Figure 4-1](#).)

We start making the `room_map` list by creating the top row, which will be a row of soil outside or the back wall inside. The top row is made of the same material all the way across, so we can use a shortcut. Try this in the shell:

```
>>> print([1] * 10)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

The `[1]` in the `print()` instruction is a list that contains just one item. When we multiply it by 10, we get a list that contains that item 10 times. In our program, we multiply the edge type we're using by the width of the room ❹. If the top edge has an exit in it, we'll add that shortly.

The middle rows of the room are made using a loop ❺ that adds each row in turn to the end of `room_map`. All the middle rows in a room are the same and are made up of the following:

1. An edge tile (either wall or soil) for the left side of the room.
2. The floor in the middle. We can use our shortcut again here. We multiply the `floor_type` by the size of the space in the middle of the room. That is the `room_width` minus 2 because there are two edge spaces.
3. The edge piece at the right side.

The bottom line is then added ❻ and is generated in the same way as the top line.

ADDING EXITS

Next, we add exits in the walls where required. We'll put the exits in the middle of the walls, so we start by figuring out where the middle row and middle column are ❼ by

dividing the room height and width by 2. Sometimes this calculation results in a number with a decimal. We need a whole number for our index positions, so we use the `int()` function to remove the decimal part ⑦. The `int()` function converts a decimal number into a whole number (an *integer*).

We check for a right exit first ⑧. Remember that `room_data` contains the information for this room, which was originally taken from `GAME_MAP`. The value `room_data[4]` tells us whether there is an exit on the right of this room. This instruction:

```
if room_data[4]:
```

is shorthand for this instruction:

```
if room_data[4] == True:
```

We use `==` to check whether two things are the same. One reason that Boolean values are often a great choice to use for your data is that they make the code easier to read and write, as this example shows.

When there is a right exit, we change three positions in the middle of the right wall from the edge type to the floor type, making a gap in the wall there. The value `room_width-1` finds the *x* position on the right edge: we subtract 1 because index numbers start at 0. In [Figure 4-2](#), for example, you can see that the room width is 11 tiles, but the index position for the right wall is 10. On the planet surface, this code doesn't change anything, because there's no wall there to put a gap in. But it's simpler to let the program add the floor tiles anyway so we don't have to write code for special cases.

Before we check whether we need an exit for the left wall, we make sure the room isn't on the left edge of the map where there can be no exit ⑨. The `%` operator gives us the remainder when we divide one number by another. If we divide the current room number by the map width, 5, using the `%` operator, we'll get a 1 if the room is on the left edge. The left edge room numbers are 1, 6, 11, 16, 21, 26, 31, 36, 41, and 46. So we only continue checking for a left exit if the remainder is not 1 (`!=` means "is not equal to").

To see whether we need an exit on the left in this room, we work out which room is on the other side of that wall by subtracting 1 from the current room number. Then we check whether that room has a right exit. If so, our current room needs a left exit, and we add it.

The exits at the top and bottom are added in a similar way ⑩. We check `room_data` directly to see whether there's an exit at the top of the room, and if so, we add a gap in that wall. We can check the room below as well to see whether there should be a bottom exit in the room.

TESTING THE PROGRAM

When you run the program, you can confirm that you don't see any errors in the Python shell. You can also check that the program is working by generating the map and then printing it from the shell, like this:

```
>>> generate_map()
>>> print(room_map)
[[1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], [1, 1,
1, 1, 1, 1, 1, 1,
1, 1, 1]]
```

The `current_room` variable is set by default to be room 31, the starting room in the game, so that is the `room_map` data that prints. From our `GAME_MAP` data (and [Figure 4-2](#)) we can see that this room has 7 rows and 11 columns, and our output confirms that we have 7 lists, each containing 11 numbers: perfect. What's more, we can see that the first row features four wall pillars, three empty spaces, and then four more wall pillars, so the function has put an exit here as we would expect. Three of the lists have a 0 as their last number too, indicating an exit on the right. It looks like the program is working!

TRAINING MISSION #1

You can change the value of `current_room` from the shell to print a different room. Try entering different values for the room, regenerating the map, and printing it. Check the output against the map and the `GAME_MAP` code to make sure the results match what you expect. Here is one example:

```
>>> current_room = 45
>>> generate_map()
>>> print(room_map)
[[1, 1, 0, 0, 0, 1, 1], [1, 0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 1],
[1, 0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 1],
[1, 0, 0, 0, 0, 0, 1]]
```



```
[1, 1, 0, 0, 0, 1, 1]
```

What happens when you enter a value for one of the planet surface rooms?

EXPLORING THE SPACE STATION IN 3D

Let's turn our room maps into rooms! We'll combine the code we created for turning room maps into 3D rooms in [Chapter 3](#) with our code for extracting the room map from the game map. Then we can tour the space station and start to get our bearings.

The *Explorer* feature of our program will enable us to view all the rooms on the space station. We'll give it its own `EXPLORER` section in the program. It's a temporary measure to enable us to quickly see results. We'll replace the *Explorer* with better code for viewing rooms in [Chapters 7 and 8](#).

Add the code in [Listing 4-3](#) to the end of your program for [Listing 4-2](#), after the instructions shown in gray. Then save the program as *listing4-3.py*. Remember to save it with your other programs for this book in the *escape* folder so the *images* folder is in the right place (see [“Downloading the Game Files”](#) on [page 7](#)).

listing4-3.py

```
room_map[room_height-1][middle_column] = floor_type
room_map[room_height-1][middle_column + 1] = floor_type
room_map[room_height-1][middle_column - 1] = floor_type

#####
## EXPLORER ##
#####

def draw():
    global room_height, room_width, room_map
    ❶ generate_map()
    screen.clear()

    ❷ for y in range(room_height):
        for x in range(room_width):
            image_to_draw = DEMO_OBJECTS[room_map[y][x]]
```

```

        screen.blit(image_to_draw,
            (top_left_x + (x*30),
             top_left_y + (y*30) - image_to_draw.get_height()))

❸ def movement():
    global current_room
    old_room = current_room

    if keyboard.left:
        current_room -= 1
    if keyboard.right:
        current_room += 1
    if keyboard.up:
        current_room -= MAP_WIDTH
    if keyboard.down:
        current_room += MAP_WIDTH

❹ if current_room > 50:
❺     current_room = 50
    if current_room < 1:
        current_room = 1

❻ if current_room != old_room:
❼     print("Entering room:" + str(current_room))

❽ clock.schedule_interval(movement, 0.1)

```

Listing 4-3: The Explorer code

The new additions in [Listing 4-3](#) should look familiar to you. We call the `generate_map()` function to create the `room_map` data for the current room ❶. We then display it ❷ using the code we created in [Listing 3-5](#) in [Chapter 3](#). We use keyboard controls to change the `current_room` variable ❸, similar to how we changed the `x` and `y` position of our spacewalking astronaut in [Chapter 1](#) (see [Listing 1-4](#)). To go up or down a row in the map, we change the `current_room` number by the width of the game map. For example, to go up a row from room 32, we subtract 5 to go into room 27 (see [Figure 4-1](#)). If the room number has changed, the program prints the `current_room` variable ❹. The `str()` function converts the room number to a string ❺, so it can be joined to the "Entering room:" string. Without using the `str()` function, you can't join a number to a string.

Finally, we schedule the `movement` function to run at regular intervals ❸, as we did in [Chapter 1](#). This time, we have a longer gap between each time the function runs (0.1 seconds), so the keys are less responsive.

From the command line, navigate to your *escape* folder and run the program from the command line using `pgzrun listing4-3.py`.

The screen should be similar to [Figure 4-4](#), which shows the walls and doorways for room 31.

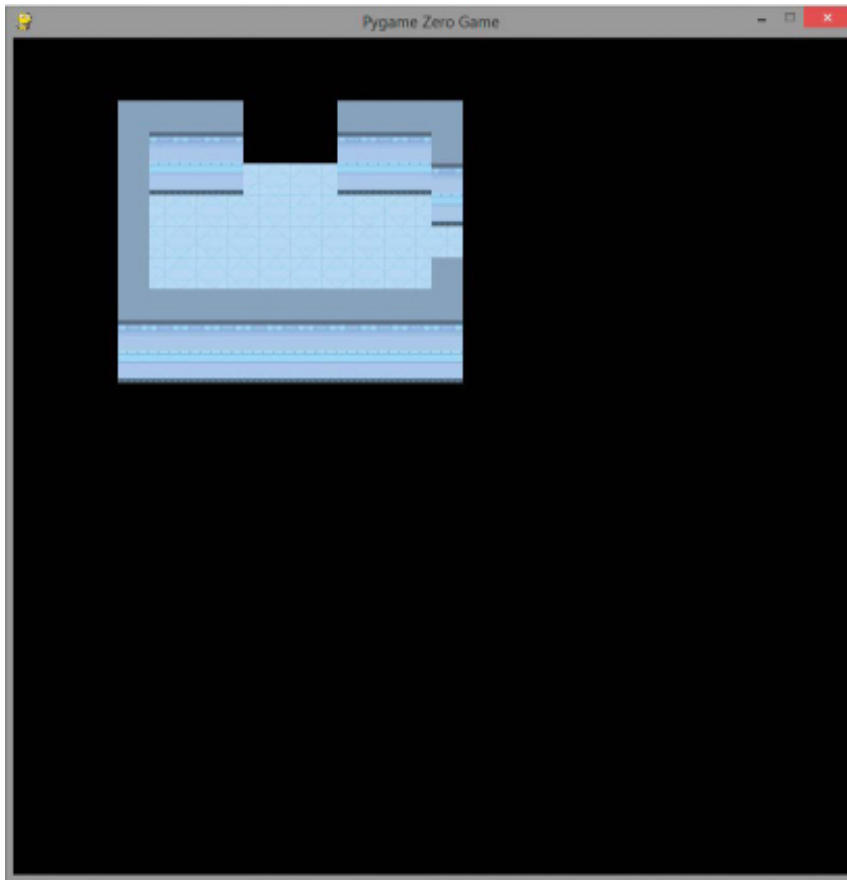


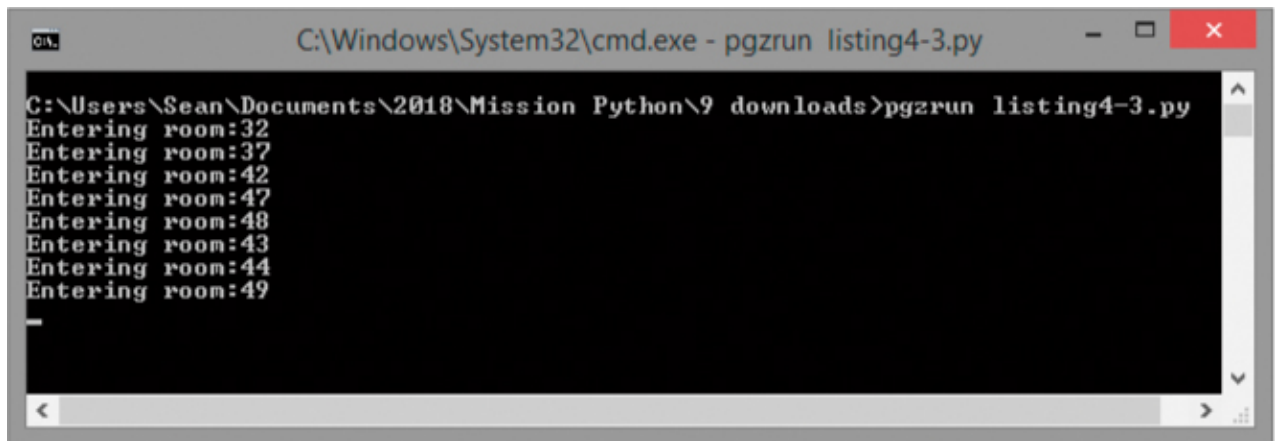
Figure 4-4: The Explorer shows your starting room in 3D.

Now you can use the arrow keys to explore the map. The program will draw a room for you and enable you to go to the neighboring rooms by pressing an arrow key. At this point, you only see the shell of the room: walls and floor. We'll add more objects in the rooms and your character later.

At the moment, you can walk in any direction, including through walls: the program doesn't check for any movement errors. If you walk off the left of the map, you'll reappear on the right, a row higher. If you walk off the right, you'll reappear on the left, a row lower. If you try to go off the top or the bottom of the map, the program will return you to room 1 (at the top) or room 50 (at the bottom). For example, if the room number is more than (>) 50 ❹ it's reset to 50 ❺. In this code, I've lowered the sensitivity

of the keys to reduce the risk of whizzing through the rooms too fast. If you find the controls unresponsive or sluggish, you might need to press the keys for slightly longer.

Explore the space station and compare what you see on screen with the map in [Figure 4-1](#). If you see any errors, go back to the `GAME_MAP` data to check the data, and then take another look at the `generate_map()` function to make sure it's been entered correctly. To help you follow the map, when you move to a new room, its number will appear in the command line window where you entered the `pgzrun` command, as shown in [Figure 4-5](#).



```
C:\Windows\System32\cmd.exe - pgzrun listing4-3.py
C:\Users\Sean\Documents\2018\Mission Python\9 downloads>pgzrun listing4-3.py
Entering room:32
Entering room:37
Entering room:42
Entering room:47
Entering room:48
Entering room:43
Entering room:44
Entering room:49
-
```

Figure 4-5: The command line window tells you which room you're entering.

Also, check that exits exist from both sides: if you go through a door and it isn't there when you look from the other side, `generate_map()` has been entered incorrectly. Follow along on the map first to make sure you're not going off the edge of the map and coming back on the other side before you start debugging. It's worth taking the time to make sure your map data and functions are all correct at this point, because broken map data can make it impossible to complete the *Escape* game!

TRAINING MISSION #2

To enjoy playing *Escape* and solving the puzzles, I recommend that you use the data I've provided for the game map. It's best not to change the data until you've completed playing the game and have decided to redesign it.

Otherwise, objects might be in locations you can't reach, making the game impossible to complete.

However, you can safely extend the map. The easiest way to do so is to add another row of rooms at the bottom of the map, making sure a door connects at least one of the new rooms to the existing bottom row of the map.

Remember to change the `MAP_HEIGHT` variable. You'll also need to change the number 50 in the *Explorer* code (*listing4-3.py*) to your highest room number

(see ❹ and ❺). Why not add a corridor now?

MAKING YOUR OWN MAPS

After you've finished building and playing *Escape*, you can customize the map or design your own game layouts using this code.

If you want to add your own map data for rooms 1 to 25, delete the code that generates their data automatically (see ❻ in [Listing 4-1](#)). You can then add your own data for these rooms.

Alternatively, if you don't want to use the planet surface locations, just block the exit to them. The exit onto the planet surface is in room 26. Change that room's entry in the `GAME_MAP` list so it doesn't have a top exit. You can use room numbers starting at room 26 and extend the map downward to make a game that is completely indoors. As a result, you won't need to make any code changes to account for the planet surface.

If you remove a doorway from the *Escape* game map (including the one in room 26), you might also need to remove a door. Some of the exits at the top and bottom of the room will have doors that seal them off. (We'll add doors to the *Escape* game in [Chapter 11](#).)

ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ The `GAME_MAP` list stores the main map data for *Escape*.
- ☐ The `GAME_MAP` only needs to store the exit at the top and right of a room.
- ☐ When the player visits a room, the `generate_map()` function makes the `room_map` list for the current room. The `room_map` list describes where the walls and objects are in the room.
- ☐ Locations 1 to 25 are on the planet surface, and a loop generates their map data. Locations 26 to 50 are the space station rooms, and you need to input their data manually.
- ☐ We use comments to help us find our way around the *Escape* program listing.

☐ When adding data using a program in script mode, you can use the shell to check the contents of lists and variables to make sure the program is working correctly.

Remember to run the program first to set up the data!

☐ The *Explorer* code enables you to look at every room in the game map using the arrow keys.

☐ It's important to make sure the game map matches Figure 4-1. Otherwise, it might not be possible for players to complete the *Escape* game. You can use the *Explorer* program to do this.

MISSION DEBRIEF

Here are the solutions to the training missions in this chapter.

TRAINING MISSION #1

If you go to one of the planet surface rooms, the entire map consists of Martian soil, so you should see only the number 2 repeated. If you go to a surface room that borders the space station, you should also see the space station wall at the bottom.

TRAINING MISSION #2

To extend my game, I added a secret passageway at the bottom of the map that connects rooms 46 and 50. To do so, in the `MAP` section of the program, change `MAP_HEIGHT` from 10 to 11:

```
MAP_HEIGHT = 11
```

In the `GAME_MAP` list, add a comma at the end of room 50's data but before the `#` comment:

```
["The south east corner", 7, 9, True, False], # room 50
```

Add a row of rooms in the `GAME_MAP` list, after room 50. Each room's list must end with a comma except for the final room list. All of the lists should be inside the final closing square bracket of `GAME_MAP`:

```
--snip--
["The robot workshop", 9, 11, True, False], # room 50
["Secret Passageway", 9, 15, True, True], # room 51
["Secret Passageway", 9, 9, False, True], # room 52
["Secret Passageway", 9, 15, False, True], # room 53
["Secret Passageway", 9, 9, False, True], # room 54
["Secret Passageway", 9, 15, True, False] # room 55
]
```

I alternated the width of the rooms in this passageway between 15 and 9, so you can easily see when you've moved to another room. If your rooms all look the same, it's hard to know when you've moved to a different room in this simple *Explorer* program. In the final *Escape* game, you will be able to clearly see when you walk between similar rooms because the character will walk out one door and enter through the opposite door.

I also changed the *Explorer* code (*listing4-3.py*) to show my new row of rooms up to room 55:

```
--snip--
if current_room > 55:
    current_room = 55
if current_room < 1:
    current_room = 1
--snip--
```

5

PREPARING THE SPACE STATION EQUIPMENT



Now that the space station walls are in place, we can start installing the equipment. We'll need detailed information about the different pieces of equipment, including the furniture, survival systems, and experimental machinery. In this chapter, you'll add information about all the items on the space station, including their images and descriptions. You'll also experiment with designing your own room and view it using the *Explorer* program you created in [Chapter 4](#).

CREATING A SIMPLE PLANETS DICTIONARY

To store the information about the space station equipment, we'll use a programming concept called dictionaries. A *dictionary* is a bit like a list but with a built-in search engine. Let's take a closer look at how it works.

UNDERSTANDING THE DIFFERENCE BETWEEN A LIST AND A DICTIONARY

As with a paper dictionary, you can use a word or phrase to look up information in a Python dictionary. That word or phrase is called the *key*, and the information linked to the key is called the *value*. Unlike in a paper dictionary, the entries in a Python dictionary can be in any order. They don't have to be alphabetical. Python can go

directly to the entry you need, wherever it is.

Imagine you have a list that contains information about previous space missions. You could get the first item from that list by using this line:

```
print(mission_info[0])
```

If `mission_info` was a dictionary instead of a list, you could use a mission name instead of an index number to get the information on that mission, like this:

```
print(mission_info["Apollo 11"])
```

The key can be a word or phrase but can also be a number. We'll start by using words because it's easier to understand the difference between a list and a dictionary that way.

MAKING AN ASTRONOMY CHEAT SHEET DICTIONARY

All astronauts need a good understanding of the solar system, so let's learn about the planets as we build our first dictionary. We'll use the planet names as the keys and connect each name to information about that planet.

Take a look at [Listing 5-1](#), which creates a dictionary called `planets`. When you make a dictionary, you use curly brackets `{}` to mark the start and end of it, instead of the square brackets you use for a list.

Each entry in the dictionary is made up of the key, followed by a colon and then the information for that entry. As with a list, we separate the entries with commas and put double quotes around pieces of text.

Open a new file in IDLE (**File** ▶ **New File**) and enter the following program. Save it as *listing5-1.py*.

listing5-1.py

```
planets = { "Mercury": "The smallest planet, nearest the Sun",  
            "Venus": "Venus takes 243 days to rotate",  
            "Earth": "The only planet known to have native life",  
            "Mars": "The Red Planet is the second smallest planet",  
            "Jupiter": "The largest planet, Jupiter is a gas giant",  
            "Saturn": "The second largest planet is a gas giant",
```

```
"Uranus": "An ice giant with a ring system",  
"Neptune": "An ice giant and farthest from the Sun"  
}
```

```
❶ while True:  
❷     query = input("Which planet would you like information on? ")  
❸     print(planets[query])
```

Listing 5-1: Your first dictionary program

This program doesn't use Pygame Zero, so you can run it by clicking **Run ▶ Run Module** at the top of the IDLE window. (It will still work if you run it using `pgzrun`, but it's easier to use the menu.) When you run the program, it asks you which planet you want information on using the `input()` built-in function ❷. Try entering **Earth** or **Jupiter** for the planet name.

```
Which planet would you like information on? Earth  
The only planet known to have native life  
Which planet would you like information on? Jupiter  
The largest planet, Jupiter is a gas giant
```

Whichever planet name you enter is stored in the variable `query`. That variable is then used to look up the information for that planet in the `planets` dictionary ❸. Instead of using an index number inside the square brackets as we did with a list, we use the word we entered to get the information, and that word is stored in the variable `query`.

In Python, we can use a `while` ❶ loop to repeat a set of instructions. Unlike a `for` loop, which we use to repeat a certain number of times, a `while` loop usually repeats until something changes. Often in a game, the `while` command will check a variable to decide whether to keep repeating instructions. For example, the instruction `while lives > 0` could keep a game going until the player runs out of lives. When the `lives` variable changes to 0, the instructions in the loop would stop repeating.

The `while True` command we use in *listing5-1.py* will keep repeating forever, because it means “while `True` is `True`,” which is always. For this `while True` command to work, make sure you capitalize `True` and place a colon at the end of the line.

Under the `while` command, we use four spaces to indent the instructions that should repeat. Here, we've indented the lines that ask you for a planet name and then give you

the planet information, so they're the instructions that repeat. After you enter a planet name and get the information, the program asks you for another planet name, and another, forever. Or until you stop the program by pressing CTRL-C, at least.

Although this program works, it isn't complete yet. You might get an unhelpful error if you enter a planet name that isn't in the dictionary. Let's fix the code so it returns a useful message instead.

ERROR-PROOFING THE DICTIONARY

When you enter a key that isn't in the dictionary, you'll see an error message. Python looks for an exact match. So, if you try to look up something that isn't in the dictionary or make even a tiny spelling mistake, you won't get the information you want.

Dictionary keys, like variable names, are case sensitive, so if you type `earth` instead of `Earth`, the program will crash. If you enter a planet that doesn't exist, this is what happens:

Which planet would you like information on? Pluto

Traceback (most recent call last):

```
File "C:\Users\Sean\Documents\Escape\listing5-1.py", line 13, in <module>
    print(planets[query])
```

```
KeyError: 'Pluto'
```

```
>>>
```

Poor Pluto! After 76 years of service, it was disqualified as a planet in 2006, so it's not in our `planets` dictionary.

TRAINING MISSION #1

Can you add an entry for Pluto in the dictionary? Pay special attention to the position of the quotes, colon, and comma. You can add it at any position in the dictionary.

When the program looks for an item in the dictionary that isn't there, it stops the program and drops you back at the Python shell prompt. To avoid this, we need the program to check whether the word entered is one of the keys in the dictionary before it tries to use it.

You can see which keys are in the dictionary by entering the dictionary name followed by a dot and `keys()`. The technical jargon for this is a *method*. Roughly speaking, a method is a set of instructions that you can attach to a piece of data using a period. Run the following code in the Python shell:

```
>>> print(planets.keys())
dict_keys(['Mars', 'Pluto', 'Jupiter', 'Earth', 'Uranus', 'Saturn', 'Mercury',
'Neptune', 'Venus'])
```

You might notice something odd here. When I completed Training Mission #1, I added Pluto to the dictionary as the last item. But in this output, it's in second place in my list of keys. When you add items to a list, they're placed at the end, but in a dictionary, that is not always the case. It depends on which version of Python you're using. (The latest version does keep dictionary items in the same order you added them.) As mentioned earlier, the order of the keys in the dictionary doesn't matter, though. Python figures out where the keys are in the dictionary, so you never need to think about it.

To stop the program from crashing when a user asks for information on a planet that isn't in the dictionary, modify your program with the new lines shown in [Listing 5-2](#).

listing5-2.py

```
--snip--
while True:
    query = input("Which planet would you like information on? ")
    ❶ if query in planets.keys():
    ❷     print(planets[query])
    else:
    ❸     print("No data available! Sorry!")
```

Listing 5-2: Error proofing the dictionary lookup

Save the program as *listing5-2.py*, and run it by clicking **Run ▶ Run Module**. Check that it works by entering a planet correctly, and then enter another planet that isn't in the list of keys. Here's an example:

```
Which planet would you like information on? Venus
Venus takes 243 days to rotate
Which planet would you like information on? Tatooine
```

We protect our program from crashing by making it check whether the key in `query` exists in the dictionary before the program tries to use it ❶. If the key does exist, we use the query as we did before ❷. Otherwise, we send a message to users telling them that we don't have that information in our dictionary ❸. Now the program is much friendlier.

PUTTING LISTS INSIDE DICTIONARIES

Our planet dictionary is a bit limited at the moment. What if we want to add extra information, such as whether the planet has rings and how many moons it has? To do so, we can use a list to store multiple pieces of information about a planet and then put that list inside the dictionary.

For example, here is a new entry for Venus:

```
"Venus": ["Venus takes 243 days to rotate", False, 0]
```

The square brackets mark the start and end of the list, and there are three items in the list: a short description, a `True` or `False` value that indicates whether or not the planet has rings, and the number of moons it has. Because Venus doesn't have rings, the second entry is `False`. It also doesn't have any moons, so the third entry is `0`.

RED ALERT

True and False values need to start with a capital letter and shouldn't be in quotes. The words turn orange when you type them correctly in IDLE.

Change your dictionary code so each key has a list, as shown in [Listing 5-3](#), keeping the rest of the code the same. Remember that dictionary entries are separated by commas, so there's a comma after the closing bracket for all the lists except the last one. Save your updated program as *listing5-3.py*.

I've slipped in information for Pluto too. Some speculate that Pluto might have rings, and exploration continues. By the time you read this book, our understanding of Pluto might have changed.

```
planets = { "Mercury": ["The smallest planet, nearest the Sun", False, 0],
            "Venus": ["Venus takes 243 days to rotate", False, 0],
            "Earth": ["The only planet known to have native life", False, 1],
            "Mars": ["The second smallest planet", False, 2],
            "Jupiter": ["The largest planet, a gas giant", True, 67],
            "Saturn": ["The second largest planet is a gas giant", True, 62],
            "Uranus": ["An ice giant with a ring system", True, 27],
            "Neptune": ["An ice giant and farthest from the Sun", True, 14],
            "Pluto": ["Largest dwarf planet in the Solar System", False, 5]
        }
--snip--
```

Listing 5-3: Putting a list in a dictionary

Run the program by selecting **Run ▶ Run Module**. Now when you ask for information on a planet, the program should display the entire list for that planet:

```
Which planet would you like information on? Venus
['Venus takes 243 days to rotate', False, 0]
Which planet would you like information on? Mars
['The second smallest planet', False, 2]
```

EXTRACTING INFORMATION FROM A LIST INSIDE A DICTIONARY

We know how to get a list of information from a dictionary, so the next step is to get individual pieces of information from that list. For example, the `False` entry doesn't mean much by itself. If we can separate it from the list, we can add an explanation beside it so the results are easier to understand. We previously used lists inside *lists* for the room map in [Chapter 4](#). Now, as then, we'll use index numbers to get items from a list in a *dictionary*.

Because `planets[query]` is the entire list, we can see just the description (the first item in the list) by using `planets[query][0]`. We can see whether it has rings or not by using `planets[query][1]`. Briefly, here is what we're doing:

1. We're using the planet name, stored in the variable `query`, to access a particular list from the `planets` dictionary.

2. We're using an index number to take an individual item from that list.

Modify your program to look like [Listing 5-4](#). As before, change only the lines that are not grayed out. Save your program as *listing5-4.py*, and run it by clicking **Run ▶ Run Module**.

listing5-4.py

```
--snip--
while True:
    query = input("Which planet would you like information on? ")
    if query in planets.keys():
❶      print(planets[query][0])
❷      print("Does it have rings? ", planets[query][1])
    else:
        print("Databanks empty. Sorry!")
```

Listing 5-4: Displaying information from the list stored in the dictionary

When you run the *listing5-4.py* program, you should see something like the following:

```
Which planet would you like information on? Earth
The only planet known to have native life
Does it have rings? False
Which planet would you like information on? Saturn
The second largest planet is a gas giant
Does it have rings? True
```

This should work for every planet in the dictionary!

When you enter the name of a planet that's in the dictionary, the program now prints the first item from its list of information, which is the description ❶. On the next line, the program asks itself whether that planet has rings and then shows you the `True` or `False` answer, which is the second item in that planet's list of information ❷. You can display some text and some data using the same `print()` instruction, by separating them with a comma. The display is much clearer than printing the entire list, and the information is easier to understand.

TRAINING MISSION #2

Can you modify the program to also tell you how many moons the planet has?

MAKING THE SPACE STATION OBJECTS DICTIONARY

Let's put our knowledge of how to use dictionaries, and lists inside dictionaries, to use in the space station. With all the furniture, life support equipment, tools, and personal effects required on the space station, there's a lot of information to keep track of. We'll use a dictionary called `objects` to store information about all the different items in the game.

We'll use numbers as the keys for the objects. It's simpler than using a word for each object. Also, using numbers makes it easier to understand the room map if you want to print it as we did in [Chapter 4](#). There's less risk of mistyping, too. When we create the code for the puzzles later, it'll be less obvious what the solution is, which means there will be fewer spoilers if you're building the game before playing it.

You might remember that we used the numbers 0, 1, and 2 to represent floor tiles, wall pillars, and soil in [Chapter 4](#). We'll continue using those numbers for those items, and the rest of the objects will use numbers 3 to 81.

Each entry in the dictionary is a list containing information about the item, similar to how we made the `planets` dictionary earlier in this chapter. The lists contain the following information for each object:

An object image file Different objects can use the same image file. For example, all the access cards use the same image.

A shadow image file We use shadows to enhance the 3D perspective in the game. The two standard shadows are `images.full_shadow`, which fills a full tile space and is for larger objects, and `images.half_shadow`, which fills half a tile space for smaller objects. Objects with a distinctive outline, such as the cactus, have their own shadow image file that is used only for that object. Some items, like the chair, have the shadow within the image file. Some items have no shadow, like the crater and any items the player can carry. When an image has no shadow, we write `None` where its shadow filename belongs in the dictionary. The word `None` is a special data type in Python. Like with `True` and `False`, you don't need any quotes around it, and it should start with a

capital letter. When you enter it correctly, `None` turns orange in the code.

A long description A long description is displayed when you examine or select an object while playing the game. Some of the long descriptions include clues, and others simply describe the environment.

A short description Typically just a few words, such as “an access card,” a short description is shown onscreen when you do something with the object while playing the game. For example, “You have dropped an access card.” A short description is only required for items that the player can pick up or use, such as an access card or the vending machine.

The game can reuse items in the `objects` dictionary. For example, if a room is made of 60 or more identical wall pillars, the game can just reuse the same wall pillar object. It only needs to be in the dictionary once.

There are some items that use the same image files but have other differences, which means we must store them separately in the dictionary. For example, the access cards have different descriptions depending on who they belong to, and the doors have different descriptions to tell you which key to use. Each access card and door has its own entry in the `objects` dictionary.

ADDING THE FIRST OBJECTS IN ESCAPE

Open *listing4-3.py*, which you created in [Chapter 4](#). This listing contains the game map and the code to generate the room map. We’ll add to this program to continue building the *Escape* game.

First, we need to set up some additional variables. Before the adventure begins, a research craft, called the Poodle lander, crash-lands on the planet surface. We’ll store coordinates for a random crash site in these new variables. We’ll add these variables now because the map object (number 27) will require them for its description.

Add the new lines in [Listing 5-5](#) to the `VARIABLES` section, marked out with a hashed box, in your existing *listing4-3.py* file. I recommend adding them at the end of your other variables, just above where the `MAP` section begins, so your listing and my listing are consistent. Save your program as *listing5-5.py*. The program won’t do anything new if you run it now, but if you want to try it, enter `pgzrun listing5-5.py`.

listing5-5.py

```

--snip--
#####

## VARIABLES ##
#####

--snip--

DEMO_OBJECTS = [images.floor, images.pillar, images.soil]

LANDER_SECTOR = random.randint(1, 24)
LANDER_X = random.randint(2, 11)
LANDER_Y = random.randint(2, 11)

#####
##  MAP  ##
#####
--snip--

```

Listing 5-5: Adding the crash site location variables

These new instructions create variables to remember the sector (or room number) the Poodle landed on, and its *x* and *y* position in that sector. The instructions use the `random.randint()` function, which picks a random number between the two numbers you give it. These instructions run once at the start of the game, so the lander location is different each time you play but doesn't change during the game.

Now let's add the first chunk of the objects data, shown in [Listing 5-6](#). This section provides the data for objects 0 to 12. Because the player cannot pick up or use these objects, they don't have a short description.

Place this section of the listing just above the `MAKE MAP` section of your existing program (*listing5-5.py*). To help you find your way around the listing, you can press CTRL-F in IDLE to search for a particular word or phrase. For example, try searching for *make map* to see where to start adding the code in [Listing 5-6](#). After searching, click Close on the search dialog box. Remember that if you get lost in the listing, you can always refer to the complete game listing in [Appendix A](#).

If you prefer not to type the data, use the file *data-chapter5.py*, in the listings folder. It contains the `objects` dictionary, so you can copy and paste it into your program. You can

start by just pasting the first 12 items.

listing5-6.py

```
--snip--

assert len(GAME_MAP)-1 == MAP_SIZE, "Map size and GAME_MAP don't match"

#####
## OBJECTS ##
#####

objects = {
    0: [images.floor, None, "The floor is shiny and clean"],
    1: [images.pillar, images.full_shadow, "The wall is smooth and cold"],
    2: [images.soil, None, "It's like a desert. Or should that be dessert?"],
    3: [images.pillar_low, images.half_shadow, "The wall is smooth and cold"],
    4: [images.bed, images.half_shadow, "A tidy and comfortable bed"],
    5: [images.table, images.half_shadow, "It's made from strong plastic."],
    6: [images.chair_left, None, "A chair with a soft cushion"],
    7: [images.chair_right, None, "A chair with a soft cushion"],
    8: [images.bookcase_tall, images.full_shadow,
        "Bookshelves, stacked with reference books"],
    9: [images.bookcase_small, images.half_shadow,
        "Bookshelves, stacked with reference books"],
    10: [images.cabinet, images.half_shadow,
        "A small locker, for storing personal items"],
    11: [images.desk_computer, images.half_shadow,
        "A computer. Use it to run life support diagnostics"],
    12: [images.plant, images.plant_shadow, "A spaceberry plant, grown here"]
}

#####
## MAKE MAP ##
#####
--snip--
```

Listing 5-6: Adding the first objects

Remember that the colors of the code can help you spot errors. If your text sections aren't green, you've left out the opening double quotes. If there is too much green, you might have forgotten the closing double quotes. Some of the lists continue on the next line, and Python knows the list isn't complete until it sees the closing bracket. If you struggle to get any of the listings to work, you can use my version of the code (see [“Using My Example Listings” on page 21](#)) and pick up the project from any point.

[Listing 5-6](#) looks similar to our earlier `planets` dictionary: we use curly brackets to mark the start and end of the dictionary, and each entry in the dictionary is a list so it is inside square brackets. The main difference is that this time the key is a number instead of a word.

Save your new program as *listing5-6.py*. This program uses Pygame Zero for the graphics, so you need to run your new program by entering `pgzrun listing5-6.py`. It should work the same as it did previously because we've added new data but haven't done anything with that data yet. It's worth running the program anyway, because if you see an error message in the command line window, you can fix the new code before you go any further.

VIEWING OBJECTS WITH THE SPACE STATION EXPLORER

To see the objects, we have to tell the game to use the new dictionary. Change the following line in the `EXPLORER` part of the program from this:

```
image_to_draw = DEMO_OBJECTS[room_map[y][x]]
```

to the following:

```
image_to_draw = objects[room_map[y][x]][0]
```

This small change makes the *Explorer* code use our new `objects` dictionary instead of the `DEMO_OBJECTS` list we told it to use previously.

Notice that we're now using lowercase letters instead of capital letters. In this program, I use capital letters for constants whose values won't change. The `DEMO_OBJECTS` list never changed: it was only used for looking up the image filenames. But the `objects` dictionary will sometimes have its content changed as the game is played.

The other difference is that `[0]` is on the end of the line now. This is because when we pull an item from the `objects` dictionary, it gives us an entire list of information. But we

just want to use the image here, which is the first item in that list, so we use the index number `[0]` to extract it.

Save the program and run it again, and you should see that the rooms look the same as before. That's because we haven't added any new objects yet, and we kept the object numbers for the floor, wall, and soil the same as the index numbers we were using for them before.

DESIGNING A ROOM

Let's add some items to the room display. In the `EXPLORER` section of the code, add the new lines shown in [Listing 5-7](#):

listing5-7.py

```
--snip--

#####
## EXPLORER ##
#####

def draw():
    global room_height, room_width, room_map
    print(current_room)
    generate_map()
    screen.clear()
    room_map[2][4] = 7
    room_map[2][6] = 6
    room_map[1][1] = 8
    room_map[1][2] = 9
    room_map[1][8] = 12
    room_map[1][9] = 9
--snip--
```

Listing 5-7: Adding some objects in the room display

These new instructions add objects to the `room_map` list at different positions before the room is displayed.

Remember that `room_map` uses the y-coordinate before the x-coordinate. The first index

number says how far from the back of the room the objects are; the smaller the number the nearer to the back they are. The smallest useful number is usually 1 because the wall is in row 0.

The second number says how far across the room the objects are, from left to right. There's usually a wall in column 0, so 1 is the smallest useful number for this position too.

The number on the other side of the equal sign is the key for a particular object. You can check which object each number represents by looking at the `objects` dictionary in [Listing 5-6](#).

So this line:

```
room_map[1][1] = 8
```

places object 8 (a tall bookcase) into the top-left corner of the room. And this line:

```
room_map[2][6] = 6
```

places a chair (object 6) three rows from the top and seven positions from the left. (Remember that index numbers start at 0.)

Save your program as *listing5-7.py* and enter `pgzrun listing5-7.py` to run it. [Figure 5-1](#) shows what you should see now.

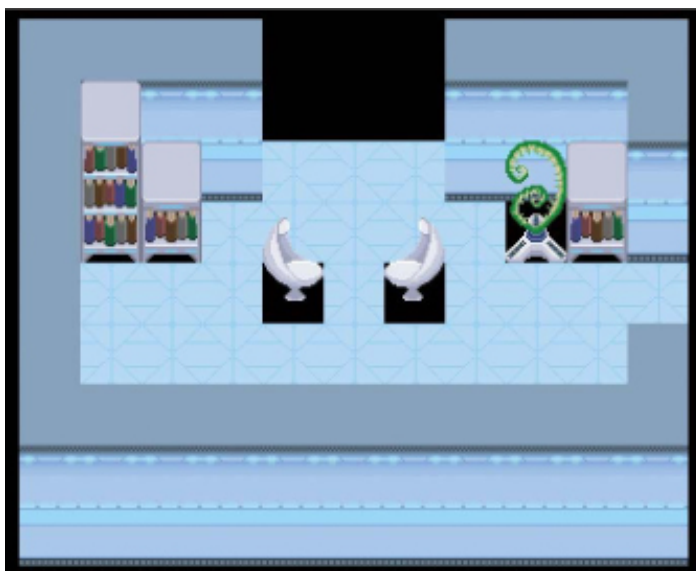


Figure 5-1: Cozy! Some objects displayed in the space station Explorer program

Because the *Explorer* program is just a demo, some things don't work yet. For example,

some objects have a black square under them because there's no floor tile there. Also, all the rooms look the same because we've coded the objects into the `EXPLORER` section, so they appear in every room we display. This means you can't view all the rooms anymore, because the objects won't fit in some of them. As a result, you can't use the arrow keys to look at all the rooms. The program doesn't display wide objects such as the bed correctly yet, either. We'll fix all of these problems later, but we can continue building and testing the space station in the meantime.

TRAINING MISSION #3

Experiment with the code you've added to the *Explorer* program to reposition the furniture to your liking. Playing with this code is a great way to learn how to position objects in the rooms. If you want to play with a bigger room, change the value of `current_room` in the `VARIABLES` section from 31 to 40 (which is the biggest room in the game). Save your program as *mission5-3.py* and run it using `pgzrun mission5-3.py`. You'll need to keep a safe copy of the existing *Explorer* code (*listing5-7.py*) to use in Training Mission #4.

ADDING THE REST OF THE OBJECTS

So far we've added objects 0 to 12 to the `objects` dictionary. There are 81 objects in total in the game, so let's add the rest now by adding the new lines in Listing 5-8. Remember to add a comma after item 12 before adding the rest of the items in the dictionary.

When the same filename or a similar description is used for more than one object, you can just copy and paste it. To copy code, click and hold down the mouse button at the beginning of the chunk of code, move the mouse to highlight it, and then press `CTRL-C`. Then click the mouse where you want to paste that code and press `CTRL-V`. Remember too that you can copy and paste the whole dictionary from the *data-chapter5.py* file if you want to save time typing.

Save the program as *listing5-8.py*. You can test that the program still works by entering `pgzrun listing5-8.py`, although you won't see anything new yet.

This is Listing 5-8:

listing5-8.py

```
#####
```

```
## OBJECTS ##
```

```
#####
```

```
objects = {
```

```
    o: [images.floor, None, "The floor is shiny and clean"],
```

```
    --snip--
```

```
    12: [images.plant, images.plant_shadow, "A spaceberry plant, grown locally"],
```

```
❶ 13: [images.electrical1, images.half_shadow,
      "Electrical systems used for powering the space station"],
```

```
    14: [images.electrical2, images.half_shadow,
      "Electrical systems used for powering the space station"],
```

```
    15: [images.cactus, images.cactus_shadow, "Ouch! Careful on the cactus!"],
```

```
    16: [images.shrub, images.shrub_shadow,
      "A space lettuce. A bit limp, but amazing it's growing here!"],
```

```
    17: [images.pipes1, images.pipes1_shadow, "Water purification pipes"],
```

```
    18: [images.pipes2, images.pipes2_shadow,
      "Pipes for the life support systems"],
```

```
    19: [images.pipes3, images.pipes3_shadow,
      "Pipes for the life support systems"],
```

```
❷ 20: [images.door, images.door_shadow, "Safety door. Opens automatically \
for astronauts in functioning spacesuits."],
```

```
    21: [images.door, images.door_shadow, "The airlock door. \
For safety reasons, it requires two person operation."],
```

```
    22: [images.door, images.door_shadow, "A locked door. It needs " \
      + PLAYER_NAME + "'s access card"],
```

```
    23: [images.door, images.door_shadow, "A locked door. It needs " \
      + FRIEND1_NAME + "'s access card"],
```

```
    24: [images.door, images.door_shadow, "A locked door. It needs " \
      + FRIEND2_NAME + "'s access card"],
```

```
    25: [images.door, images.door_shadow,
      "A locked door. It is opened from Main Mission Control"],
```

```
    26: [images.door, images.door_shadow,
      "A locked door in the engineering bay."],
```

```
❸ 27: [images.map, images.full_shadow,
      "The screen says the crash site was Sector: " \
      + str(LANDER_SECTOR) + " // X: " + str(LANDER_X) + \
      " // Y: " + str(LANDER_Y)],
```

```
    28: [images.rock_large, images.rock_large_shadow,
```

"A rock. Its coarse surface feels like a whetstone", "the rock"],
29: [images.rock_small, images.rock_small_shadow,
"A small but heavy piece of Martian rock"],
30: [images.crater, None, "A crater in the planet surface"],
31: [images.fence, None,
"A fine gauze fence. It helps protect the station from dust storms"],
32: [images.contraption, images.contraption_shadow,
"One of the scientific experiments. It gently vibrates"],
33: [images.robot_arm, images.robot_arm_shadow,
"A robot arm, used for heavy lifting"],
34: [images.toilet, images.half_shadow, "A sparkling clean toilet"],
35: [images.sink, None, "A sink with running water", "the taps"],
36: [images.globe, images.globe_shadow,
"A giant globe of the planet. It gently glows from inside"],
37: [images.science_lab_table, None,
"A table of experiments, analyzing the planet soil and dust"],
38: [images.vending_machine, images.full_shadow,
"A vending machine. It requires a credit.", "the vending machine"],
39: [images.floor_pad, None,
"A pressure sensor to make sure nobody goes out alone."],
40: [images.rescue_ship, images.rescue_ship_shadow, "A rescue ship!"],
41: [images.mission_control_desk, images.mission_control_desk_shadow, \
"Mission Control stations."],
42: [images.button, images.button_shadow,
"The button for opening the time-locked door in engineering."],
43: [images.whiteboard, images.full_shadow,
"The whiteboard is used in brainstorming and planning meetings."],
44: [images.window, images.full_shadow,
"The window provides a view out onto the planet surface."],
45: [images.robot, images.robot_shadow, "A cleaning robot, turned off."],
46: [images.robot2, images.robot2_shadow,
"A planet surface exploration robot, awaiting set-up."],
47: [images.rocket, images.rocket_shadow, "A 1-person craft in repair."],
48: [images.toxic_floor, None, "Toxic floor - do not walk on!"],
49: [images.drone, None, "A delivery drone"],
50: [images.energy_ball, None, "An energy ball - dangerous!"],
51: [images.energy_ball2, None, "An energy ball - dangerous!"],
52: [images.computer, images.computer_shadow,
"A computer workstation, for managing space station systems."],

53: [images.clipboard, **None**,
"A clipboard. Someone has doodled on it.", "the clipboard"],

54: [images.bubble_gum, **None**,
"A piece of sticky bubble gum. Spaceberry flavour.", "bubble gum"],

55: [images.yoyo, **None**, "A toy made of fine, strong string and plastic. \
Used for antigrav experiments.", PLAYER_NAME + "'s yoyo"],

56: [images.thread, **None**,
"A piece of fine, strong string", "a piece of string"],

57: [images.needle, **None**,
"A sharp needle from a cactus plant", "a cactus needle"],

58: [images.threaded_needle, **None**,
"A cactus needle, spearing a length of string", "needle and string"],

59: [images.canister, **None**,
"The air canister has a leak.", "a leaky air canister"],

60: [images.canister, **None**,
"It looks like the seal will hold!", "a sealed air canister"],

61: [images.mirror, **None**,
"The mirror throws a circle of light on the walls.", "a mirror"],

62: [images.bin_empty, **None**,
"A rarely used bin, made of light plastic", "a bin"],

63: [images.bin_full, **None**,
"A heavy bin full of water", "a bin full of water"],

64: [images.rags, **None**,
"An oily rag. Pick it up by a corner if you must!", "an oily rag"],

65: [images.hammer, **None**,
"A hammer. Maybe good for cracking things open...", "a hammer"],

66: [images.spoon, **None**, "A large serving spoon", "a spoon"],

67: [images.food_pouch, **None**,
"A dehydrated food pouch. It needs water.", "a dry food pack"],

68: [images.food, **None**,
"A food pouch. Use it to get 100% energy.", "ready-to-eat food"],

69: [images.book, **None**, "The book has the words 'Don't Panic' on the \
cover in large, friendly letters", "a book"],

70: [images.mp3_player, **None**,
"An MP3 player, with all the latest tunes", "an MP3 player"],

71: [images.lander, **None**, "The Poodle, a small space exploration craft. \
Its black box has a radio sealed inside.", "the Poodle lander"],

72: [images.radio, **None**, "A radio communications system, from the \
Poodle", "a communications radio"],

```

73: [images.gps_module, None, "A GPS Module", "a GPS module"],
74: [images.positioning_system, None, "Part of a positioning system. \
Needs a GPS module.", "a positioning interface"],
75: [images.positioning_system, None,
     "A working positioning system", "a positioning computer"],
76: [images.scissors, None, "Scissors. They're too blunt to cut \
anything. Can you sharpen them?", "blunt scissors"],
77: [images.scissors, None,
     "Razor-sharp scissors. Careful!", "sharpened scissors"],
78: [images.credit, None,
     "A small coin for the station's vending systems",
     "a station credit"],
79: [images.access_card, None,
     "This access card belongs to " + PLAYER_NAME, "an access card"],
80: [images.access_card, None,
     "This access card belongs to " + FRIEND1_NAME, "an access card"],
81: [images.access_card, None,
     "This access card belongs to " + FRIEND2_NAME, "an access card"]
}

```

```

❷ items_player_may_carry = list(range(53, 82))
    # Numbers below are for floor, pressure pad, soil, toxic floor.
❸ items_player_may_stand_on = items_player_may_carry + [0, 39, 2, 48]

```

```

#####
## MAKE MAP ##
#####
--snip--

```

Listing 5-8: Completing the objects data for the Escape game

Some of the lists for objects extend over more than one line in the program ❶. This is fine because Python knows the list isn't complete until it sees a closing bracket. To break a string (or any other piece of code) over more than one line, you can use a \ at the end of the line ❷. The line breaks in *listing5-8.py* are just there to make the code fit onto the book page neatly: onscreen, the code can extend to the right if you want it to.

Object 27 is a map showing the Poodle's crash site. Its long description includes the variables that you set in [Listing 5-5](#) for the Poodle's position. The `str()` function is used

to convert the numbers in those variables into strings so they can be combined with other strings to make up the long description ❸.

We've also set up some additional lists we'll need in the game: `items_player_may_carry` stores the numbers of the objects the player can pick up ❹. These are objects 53 to 81. Because they're grouped together, we can set up the `items_player_may_carry` list using a range. A *range* is a sequence of numbers that starts from the first number given and finishes at the one *before* the last number. (We used ranges in [Chapter 3](#).) We turn that range into a list using `list(range(53 to 82))`, which makes a list of all the numbers from 53 to 81.

If you add more objects that a player can carry later, you can add them to the end of this list. For example, to add new objects numbered 89 and 93 that the player can carry, use `items_player_may_carry = list(range(54, 82)) + [89, 93]`. You can also add new objects to the end of the `objects` list and just extend the range used to set up `items_player_may_carry`.

The other new list is `items_player_may_stand_on`, which specifies whether a player is allowed to stand on a particular item ❺. Players can only stand on objects small enough to be picked up and on the different types of floor. We make this list by adding the object numbers for the different floor types to the `items_player_may_carry` list.

After you've entered [Listing 5-8](#), you've completed the `OBJECTS` section of the *Escape* game! But we haven't put the objects into the game map yet. We'll start to do that in [Chapter 6](#).

TRAINING MISSION #4

Experiment with some of the new objects you just added to the game. By modifying the code, can you . . .

- Swap the tall bookcase for a bin (object 62)?
- Swap the spaceberry plant for a small rock (object 29)?
- Swap the chair on the right for a patch of toxic floor (object 48)?

To understand which instruction places which object, you can either use the coordinates in your existing code or look up the object numbers in the `objects` dictionary (onscreen or in the listings in this chapter). Run your program to make sure it works.

ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ To get information from a dictionary, you use the key for that information. The key can be a word or a number and can also be stored in a variable.
- ☐ If you try to use a key that isn't in the dictionary, you'll cause an error.
- ☐ To avoid an error, check whether the key is in the dictionary before the program tries to use it.
- ☐ You can put lists inside dictionaries. Then you can use the dictionary key followed by the list index to get a particular item from the list. For example: `planets["Earth"][1]`.
- ☐ The *Escape* game uses the `objects` dictionary to store information about all the objects in the game. Each item in the dictionary is a list.
- ☐ You can use the index number of that list to access the object's image file, shadow image file, and long and short description.

MISSION DEBRIEF

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

TRAINING MISSION #1

Make sure you add a comma after Neptune's entry, and place the quotes and colon in the correct place in Pluto's entry.

```
planets = { "Mercury": "The smallest planet, nearest the Sun",
            "Venus": "Venus takes 243 days to rotate",
            "Earth": "The only planet known to have native life",
            --snip--
            "Neptune": "An ice giant and farthest from the Sun",
            "Pluto": "The largest dwarf planet in the Solar System"
          }
```

TRAINING MISSION #2

Modify the code by adding the line in color as shown.

```
while True:
    query = input("Which planet would you like information on? ")
    if query in planets.keys():
        print(planets[query][0])
        print("Does it have rings? ", planets[query][1])
        print("How many moons? ", planets[query][2])
    else:
        print("No data available! Sorry!")
```

TRAINING MISSION #3

You can create any room design to complete this mission. Here's one suggestion: delete the existing instructions that add objects to the room, and use the following instructions instead. Run the program to see what this change does!

```
room_map[2][6] = 12
room_map[1][9] = 10
room_map[1][1] = 7
room_map[1][3] = 1
```

TRAINING MISSION #4

Edit the EXPLORER section of your program as shown here:

```
room_map[2][4] = 7
room_map[2][6] = 48
room_map[1][1] = 62
room_map[1][2] = 9
room_map[1][8] = 29
room_map[1][9] = 10
```

6

INSTALLING THE SPACE STATION EQUIPMENT



In [Chapter 5](#), you prepared information about all the equipment you'll use on your mission. In this chapter, you'll install some of that equipment in the space station and use the *Explorer* to view any room or planet surface location. This is your first chance to explore the design of the Mars base that will become your home.

UNDERSTANDING THE DICTIONARY FOR THE SCENERY DATA

There are two different types of objects on the space station:

- **Scenery** is the equipment that stays in the same place throughout the *Escape* game and includes furniture, pipes, and electronic equipment.
- **Props** are items that can appear, disappear, or move around during the game. They include things the player can create and pick up. Props also include doors, which appear in the room when they're closed and disappear when they're open.

The data for positioning scenery and the data for props are stored separately and organized differently. In this chapter, we'll just add the scenery data.

Our program already knows the image and description to use for all the objects in the game, because they're in the `objects` dictionary you created in [Chapter 5](#). Now we'll tell the program where to put the scenery objects in the space station. To do that, we'll create a new dictionary called `scenery`. This is how we'll structure the entry for one room:

```
room number: [[object number, y, x], [object number, y, x]]
```

The key for the dictionary will be the room number. For each room number, the dictionary stores a list, with a square bracket at the start and the end of it. Each item in that list is another list that tells the program where in the room to put *one* object. Here, I've made one object red and the other green so you can see where they start and end.

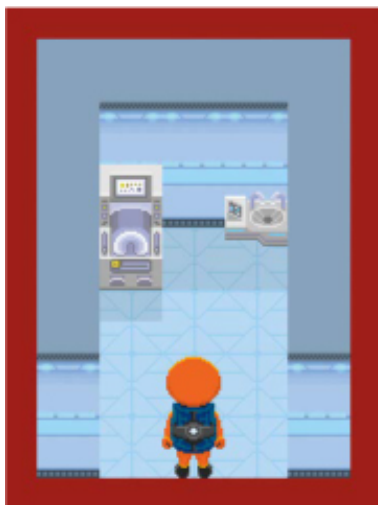
These are the three pieces of information you need for each object:

The object number This is the same as the number that is used as the key in the `objects` dictionary. For example, number 5 represents a table.

The object's *y* position This is the object's position in the room, from back to front. The back wall is usually in row 0, so we typically start placing objects at 1. The largest useful number will be the room height minus 2: we subtract 1 because the map positions start at 0 and subtract another 1 for the space the front wall occupies. In practice, it's a good idea to leave a bit more space at the front of the room, because the front wall can obscure other items. You can check the size of the room in the `GAME_MAP` code you added in [Chapter 4](#).

The object's *x* position This tells the program how far across the room from left to right the object should be. Again, a wall is usually in position 0. The largest useful number will generally be the room width minus 2.

To get a better understanding of these numbers, let's take a look at [Figure 6-1](#), which shows one of the rooms on the space station as a screenshot and a map. In this image, the sink (S) is in the second row from the back, so its *y* position is 1. Remember that the wall in the first row at the back is in position *y* = 0. The sink's *x* position is 3. There are two other tile spaces to the left of it, and the wall is in position *x* = 0.



| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | | | | | |
| 1 | | T | | S | |
| 2 | | | | | |
| 3 | | | | | |
| 4 | | | P | | |

Figure 6-1: An example space station room as seen in the game (left) and represented by a map (right). T = toilet, S = sink, P = player.

Let's look at the data for this room. Don't enter this code yet. I'll give you all the scenery data shortly.

```
scenery = {
--snip--
30: [[34,1,1], [35,1,3]],
--snip--
}
```

This code tells the program about the objects in room 30. Room 30 has object number 34, a toilet, in the top-left corner at position $y = 1$ and $x = 1$, and object number 35, a sink, at position number $y = 1$ and $x = 3$, quite close to the toilet.

You can have the same object in the room several times by adding a list for each position and using the same object number for them. For example, you could fill the room with toilets in different positions if you wanted to, although that would be a rather bizarre thing to do.

You don't need to include the walls in the scenery data, because the program automatically adds them to the room when it creates the `room_map` list, as you've already seen.

Even though putting the information for each item into a list means adding more brackets, it's much easier to understand the data at a glance. The brackets help you see how many items are in the room, which numbers are the object numbers, and which are the position numbers.

ADDING THE SCENERY DATA

Open *listing5-8.py*, the final listing in [Chapter 5](#). This listing contains your game map and objects data. Now we'll add the scenery data to it.

[Listing 6-1](#) shows the scenery data. Add this new SCENERY section before the `MAKE MAP` section. Make sure the placement of the brackets and commas is correct. Remember that each piece of scenery needs a list of three numbers, and each list is separated with a comma too. If you prefer not to type all the data in, use the file *data-chapter6.py*, which is in the *listings* folder. It contains the scenery dictionary for you to copy and paste into your program.

listing6-1.py

--snip--

```
items_player_may_stand_on = items_player_may_carry + [0, 39, 2, 48]
```

```
#####
```

```
## SCENERY ##
```

```
#####
```

```
# Scenery describes objects that cannot move between rooms.
```

```
# room number: [[object number, y position, x position]...]
```

```
scenery = {
```

```
    26: [[39,8,2]],
```

```
    27: [[33,5,5], [33,1,1], [33,1,8], [47,5,2],
```

```
        [47,3,10], [47,9,8], [42,1,6]],
```

```
    28: [[27,0,3], [41,4,3], [41,4,7]],
```

```
    29: [[7,2,6], [6,2,8], [12,1,13], [44,0,1],
```

```
        [36,4,10], [10,1,1], [19,4,2], [17,4,4]],
```

```
    30: [[34,1,1], [35,1,3]],
```

```
    31: [[11,1,1], [19,1,8], [46,1,3]],
```

```
    32: [[48,2,2], [48,2,3], [48,2,4], [48,3,2], [48,3,3],
```

```
        [48,3,4], [48,4,2], [48,4,3], [48,4,4]],
```

```
    33: [[13,1,1], [13,1,3], [13,1,8], [13,1,10], [48,2,1],
```

```
        [48,2,7], [48,3,6], [48,3,3]],
```

```
    34: [[37,2,2], [32,6,7], [37,10,4], [28,5,3]],
```

```
    35: [[16,2,9], [16,2,2], [16,3,3], [16,3,8], [16,8,9], [16,8,2], [16,1,8],
```

```
        [16,1,3], [12,8,6], [12,9,4], [12,9,8],
```



```

        [15,4,6], [12,7,1], [12,7,11]],
36: [[4,3,1], [9,1,7], [8,1,8], [8,1,9],
      [5,5,4], [6,5,7], [10,1,1], [12,1,2]],
37: [[48,3,1], [48,3,2], [48,7,1], [48,5,2], [48,5,3],
      [48,7,2], [48,9,2], [48,9,3], [48,11,1], [48,11,2]],
38: [[43,0,2], [6,2,2], [6,3,5], [6,4,7], [6,2,9], [45,1,10]],
39: [[38,1,1], [7,3,4], [7,6,4], [5,3,6], [5,6,6],
      [6,3,9], [6,6,9], [45,1,11], [12,1,8], [12,1,4]],
40: [[41,5,3], [41,5,7], [41,9,3], [41,9,7],
      [13,1,1], [13,1,3], [42,1,12]],
41: [[4,3,1], [10,3,5], [4,5,1], [10,5,5], [4,7,1],
      [10,7,5], [12,1,1], [12,1,5]],
44: [[46,4,3], [46,4,5], [18,1,1], [19,1,3],
      [19,1,5], [52,4,7], [14,1,8]],
45: [[48,2,1], [48,2,2], [48,3,3], [48,3,4], [48,1,4], [48,1,1]],
46: [[10,1,1], [4,1,2], [8,1,7], [9,1,8], [8,1,9], [5,4,3], [7,3,2]],
47: [[9,1,1], [9,1,2], [10,1,3], [12,1,7], [5,4,4], [6,4,7], [4,1,8]],
48: [[17,4,1], [17,4,2], [17,4,3], [17,4,4], [17,4,5], [17,4,6], [17,4,7],
      [17,8,1], [17,8,2], [17,8,3], [17,8,4],
      [17,8,5], [17,8,6], [17,8,7], [14,1,1]],
49: [[14,2,2], [14,2,4], [7,5,1], [5,5,3], [48,3,3], [48,3,4]],
50: [[45,4,8], [11,1,1], [13,1,8], [33,2,1], [46,4,6]]
}

checksum = 0
check_counter = 0
for key, room_scenery_list in scenery.items():
    for scenery_item_list in room_scenery_list:
        ❶ checksum += (scenery_item_list[0] * key
                      + scenery_item_list[1] * (key + 1)
                      + scenery_item_list[2] * (key + 2))
        check_counter += 1
    print(check_counter, "scenery items")
    ❷ assert check_counter == 161, "Expected 161 scenery items"
    ❸ assert checksum == 200095, "Error in scenery data"
    print("Scenery checksum: " + str(checksum))

#####
## MAKE MAP ##
#####

```


Listing 6-1: Adding the scenery data

Save your listing as *listing6-1.py*, and run it using `pgzrun listing6-1.py` in the command line. We've added some data, but we haven't told the program to do anything with it, so you won't see any change. But if you made a mistake entering the data, the program should stop and display the message `Error in scenery data`. In this case, go back and double-check your code against the book. Check that you entered the checksum number correctly first! ❸

The second half of this listing is a safety measure, called a *checksum*. It checks that all the data is present and correct by making a calculation involving the data and then checking the result against the correct answer. If there's a mistake in the data you've entered, this bit of code will stop the program until you fix it. This stops your game from running with bugs in it. (Some errors could get through, but this code catches most mistakes.)

The program uses the `assert` instruction to check the data. The first instruction checks that the program has the right number of data items. If it doesn't, the program stops and shows an error message ❷. The program also checks whether the checksum (the result from the calculation) is the expected number, and if it isn't, it stops the program ❸. Notice that one of the instructions in Listing 6-1 spreads across three lines ❶: Python knows we haven't finished the instruction until we close the final parenthesis.

TIP

If you want to change the scenery data, to redesign rooms or to add your own rooms, you will need to turn off the checksum. This is because the calculation based on your changed data will be different, so the checksum will fail and the program won't run. You can simply put a `#` symbol before the two lines that start with `assert` ❷❸ to switch them off. As you know, the `#` symbol is used for a comment, and Python ignores everything after it on the same line. It can be a handy off switch when you're building or testing programs.

ADDING THE PERIMETER FENCE FOR THE PLANET SURFACE

You might have noticed that we haven't added any scenery for rooms 1 to 25 yet. Our data starts at room 26. As you might remember, the first 25 locations are outside on the planet surface. For simplicity, we'll still call them rooms, although they have no walls.

Figure 6-2 shows rooms 1 to 25 on the map. A fence, shown as a dotted line in Figure 6-2, surrounds the outside of these rooms. The fence stops people from wandering out of the compound and off the game map.

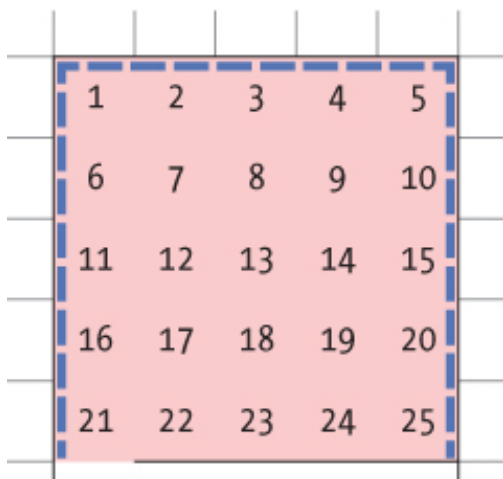


Figure 6-2: Adding the fence around the planet surface locations

We need to add fences at the following locations:

- On the left in rooms 1, 6, 11, 16, and 21
- At the top in rooms 1, 2, 3, 4, and 5
- On the right in rooms 5, 10, 15, 20, 25

Each outside room has one item of planet surface scenery too, which is randomly chosen from a small selection of suitable items that includes rocks, shrubs, and craters. For the game, it doesn't matter where these items are placed, so they can also be randomly positioned.

Listing 6-2 shows the code that generates the random planet surface scenery and adds the fences. Add the code to the end of the SCENERY section you just created, and save your program as *listing6-2.py*. You can use `pgzrun listing6-2.py` to check whether the program reports any errors.

listing6-2.py

```
print("Scenery checksum: " + str(checksum))
```

```
for room in range(1, 26): # Add random scenery in planet locations.
❶ if room != 13: # Skip room 13.
❷     scenery_item = random.choice([16, 28, 29, 30])
❸     scenery[room] = [[scenery_item, random.randint(2, 10),
                        random.randint(2, 10)]]

# Use loops to add fences to the planet surface rooms.
❹ for room_coordinate in range(0, 13):
❺     for room_number in [1, 2, 3, 4, 5]: # Add top fence
❻         scenery[room_number] += [[31, 0, room_coordinate]]
❼       for room_number in [1, 6, 11, 16, 21]: # Add left fence
❽         scenery[room_number] += [[31, room_coordinate, 0]]
           for room_number in [5, 10, 15, 20, 25]: # Add right fence
❾         scenery[room_number] += [[31, room_coordinate, 12]]

❿ del scenery[21][-1] # Delete last fence panel in Room 21
    del scenery[25][-1] # Delete last fence panel in Room 25

#####
## MAKE MAP ##
#####

--snip--
```

Listing 6-2: Generating random planet surface scenery

You don't need to understand this code to enjoy building and playing *Escape*, but if you want to dig deeper, I'll explain the code in more detail.

The first section in [Listing 6-2](#) adds the random scenery. For each room, `random.choice()` ❷ chooses a scenery item randomly. In the same way that `random.randint()` gave us a random number (like rolling dice), `random.choice()` gives us a random item (like a grab bag or lucky dip game). The item is chosen from the list `[16, 28, 29, 30]`. Those object numbers represent a shrub, a large rock, a small rock, and a crater, respectively.

We also add a new entry to the `scenery` dictionary for the room ❸. This entry contains the random scenery item and random *y* and *x* positions for that item. The *y* and *x*

positions place the item inside the room but not too near the edge.

The `!=` operator ❶ means “not equal to,” so scenery is added only if the room number is *not* 13. Who knows? Maybe it’ll be useful to have an empty space on the planet surface when you’re on your mission...

In the second part of [Listing 6-2](#), we add the fences. All the planet surface locations are 13 tiles high and 13 tiles wide, so we can use one loop ❷ to add the top and side fences. The loop’s variable, `room_coordinate`, counts from 0 to 12, and each time around the loop, fence panels are put in place at the top and the sides of the appropriate rooms.

Inside the `room_coordinate` loop, there are three loops for the `room_number`. The first `room_number` loop ❸ adds a fence along the top row of the top rooms. Instead of using a `range()`, this time we’re looping through a list. Each time through the list, the variable `room_number` takes the next number from the list `[1, 2, 3, 4, 5]`. We add a piece of scenery to the scenery list for the room, using `+=` ❹. This is scenery item 31 (a fence), in the top row of the room (at position `y = 0`). The `room_coordinate` value is used for the `x` position. This puts the top fence into rooms 1 to 5, in the top row of those rooms.

There are two other `room_number` loops inside the `room_coordinate` loop. The first one adds the left fence to rooms 1, 6, 11, 16, and 21 ❺. This time, the program uses the `room_coordinate` variable for the `y` position and uses `0` for the `x` position ❻. This puts fence panels along the left edge of those rooms. The second loop adds the right edge fence to rooms 5, 10, 15, 20, and 25. This also uses the `room_coordinate` for the `y` position of the fence panel but uses 12 for the `x` coordinate, putting a fence along the right edge of those rooms ❼.

We don’t want side fence panels where the outside area joins the space station wall. [Figure 6-3](#) shows a map of room 21. The bottom-left corner of the room should be wall, so there shouldn’t be a fence panel here. The loops we used just added a fence panel here, though, so we use an instruction ❽ to delete the last item of scenery added to this room, and to room 25, which is on the other side of the compound (see [Figure 6-2](#)). It’s easier to add these two panels and take them out again than it is to write code that avoids putting these fence panels in. The index number `-1` is a handy shortcut for referring to the last item in a list.

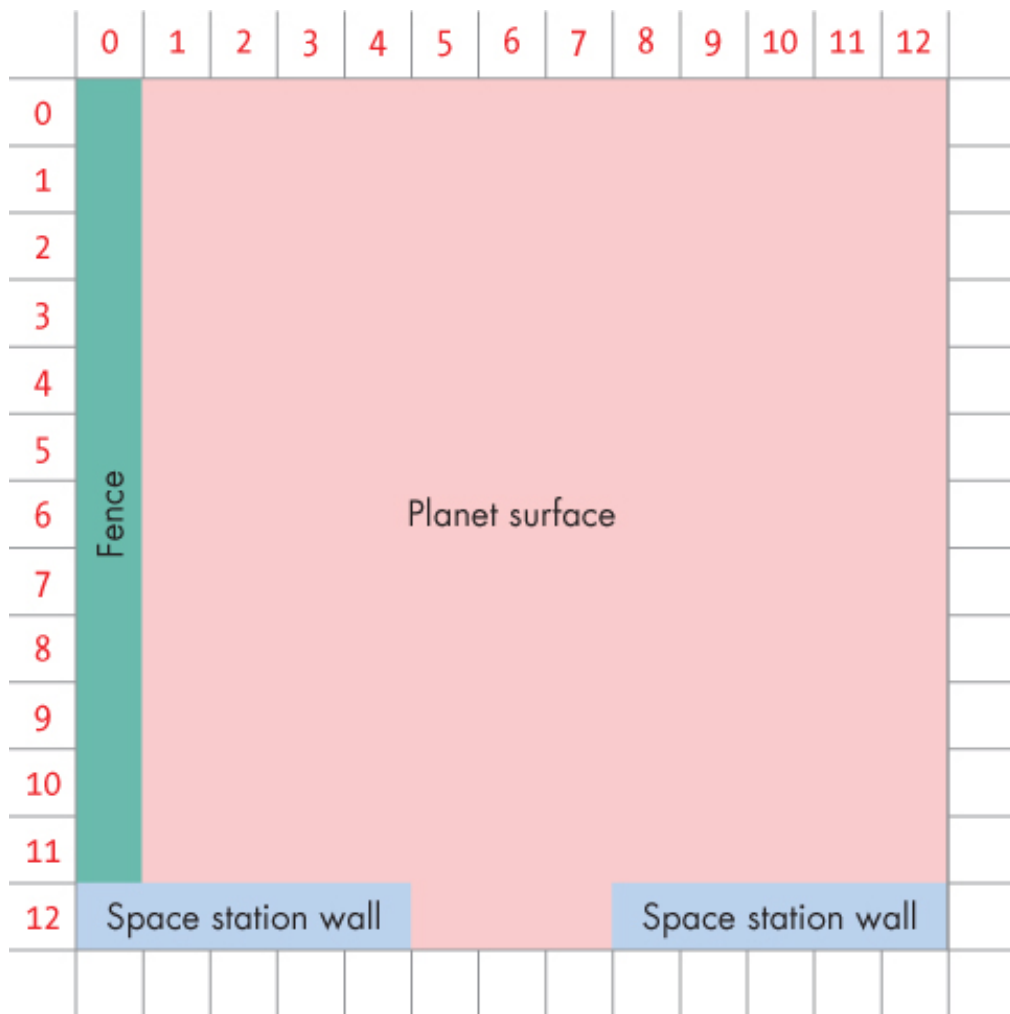


Figure 6-3: Map showing how the fence touches the wall in an outside room next to the space station

Using random scenery and loops to position fences enables us to have a large area to explore without having to type in data for over 200 fence panels and scenery items.

TIP

If you're customizing the game and don't want to add random scenery or fences in rooms 1 to 25, you can delete the code sections shown in Listing 6-2.

LOADING THE SCENERY INTO EACH ROOM

Now that we've added scenery data to the program, let's add some code so we can see the scenery in the space station! You might remember that the `generate_map()` function creates the `room_map` list for the room you're currently exploring. The `room_map` list is used to display and navigate the room.

So far, the `generate_map()` function just calculates the size of the room and where the doors are, and places the floor and walls. We need to add some code to extract the scenery from our new dictionary and add it to the `room_map`. But first, we'll make one small but important adjustment to the program. In the `VARIABLES` section, near the start of the program, add the new line shown in [Listing 6-3](#). Save your program as *listing6-3.py*.

listing6-3.py

```
--snip--

#####

## VARIABLES ##

#####

--snip--

LANDER_SECTOR = random.randint(1, 24)
LANDER_X = random.randint(2, 11)
LANDER_Y = random.randint(2, 11)

TILE_SIZE = 30

#####

##  MAP  ##

#####

--snip--
```

Listing 6-3: Setting up the `TILE_SIZE` variable

This line creates a variable to store the size of a tile. Using it makes the program easier to read because we can replace the number 30 with a more meaningful phrase. Instead of seeing the number 30 in the code and having to remember what it represents, we can see the words `TILE SIZE` instead, which gives us a hint about what the code is doing.

Next, find the `MAKE MAP` section of the program: it comes before the `EXPLORER` section. Add [Listing 6-4](#) to the end of the `MAKE MAP` section to place the scenery in the current room. All the code in [Listing 6-4](#) belongs to the `generate_map()` function, so we need to indent the

first line by four spaces and then indent the remaining lines as shown. Save your program as *listing6-4.py*.

listing6-4.py

```
--snip--

def generate_map():
    --snip--

❶ if current_room in scenery:
❷     for this_scenery in scenery[current_room]:
❸         scenery_number = this_scenery[0]
❹         scenery_y = this_scenery[1]
❺         scenery_x = this_scenery[2]
❻         room_map[scenery_y][scenery_x] = scenery_number

❽         image_here = objects[scenery_number][0]
❾         image_width = image_here.get_width()
❿         image_width_in_tiles = int(image_width / TILE_SIZE)
❶❶ for tile_number in range(1, image_width_in_tiles):
            room_map[scenery_y][scenery_x + tile_number] = 255

#####
## EXPLORER ##
#####

--snip--
```

Listing 6-4: Additional code for `generate_map()` that adds the scenery for the current room to the `room_map` list

Let's break this down. The line at ❶ checks whether there's an entry for the current room in the `scenery` dictionary. This check is essential because some rooms in our game might not have any scenery, and if we try to use a dictionary key that doesn't exist, Python stops with an error.

We then set up a loop ❷ that cycles through the scenery items for the room and copies them into a list called `this_scenery`. The first time through the loop, `this_scenery` contains

the list for the first scenery item. The second time, it contains the list for the second item, and so on until it reaches the final scenery item for the current room.

Each scenery item has a list containing its object number, *y* position, and *x* position. The program extracts these details from `this_scenery` using index numbers and puts them into variables called `scenery_number` ❸, `scenery_y` ❹, and `scenery_x` ❺.

Now the program has all the information it needs to add the scenery item to `room_map`. You might remember that `room_map` stores the object number of the item in each position in the room. It uses the *y* position and *x* position in the room as list indexes. This program uses the `scenery_y` and `scenery_x` values as list indexes to put the item `scenery_number` into `room_map` ❻.

If all our objects were one tile wide, that is all we would need to do. But some objects are wider and cover several tiles. For example, a wide object positioned in one tile might cover two more tiles to its right, but at the moment, the program only sees it in that one tile.

We need to add something to `room_map` in those additional spaces so the program knows the player can't walk on those tiles. I've used the number 255 to represent a space that doesn't have an object in it but also cannot be walked on.

Why the number 255? It's a large enough number to give you space to add many more objects to the game if you want to, allowing for 254 items in the `objects` dictionary. Also, it feels like a nice number to me: it's the highest number you can write with one byte of data (that mattered when I started writing games in the 1980s, and the computer only had about 65,000 bytes of memory to store all its data, graphics, and program code).

First, we need to figure out how wide an image is so we know how many tiles it fills. We use `scenery_number` as the dictionary key to get information about the object from the `objects` dictionary ❼. We know the `objects` dictionary returns a list of information, the first item of which is the image. So we use the index 0 to extract the image and put it into the variable `image_here`.

Then we can use Pygame Zero to find out the width of an image by adding `get_width()` after its name ❽. We put that number into a variable called `image_width`. Because we need to know how many tiles the image covers, the program divides the image width (in pixels) by the tile size, 30, and makes it an integer (a whole number) ❾. We must convert the number to an integer because we're going to use it in the `range()` function ❿, which can only take integers. If we didn't convert the number, the width would be a

floating-point number—a number with a decimal point.

Finally, we set up a loop that adds the value 255 in the spaces to the right of the scenery item, wherever the tile is covered 10.

If an image is 90 pixels wide, we divide it by the tile size of 30 and store the result, 3, in `image_width_in_tiles`. Then the loop counts to 2 using `range()` because we give it a range of 1 to `image_width_in_tiles` 10. We add the loop numbers to the `x` position of the object, and those positions in `room_map` are marked with 255. Large objects that cover three tiles now have 255 in the next two spaces to the right.

Now our program contains all the scenery and can add it to the `room_map`, ready for display. Next, we'll make some small changes to the `EXPLORER` section so we can tour the space station.

UPDATING THE EXPLORER TO TOUR THE SPACE STATION

The `EXPLORER` part of the program lets you view all the rooms on the space station and move around the map using the arrow keys. Let's update that section so you can see all the scenery in place.

If your *Explorer* code includes any lines for adding scenery to the `room_map`, you'll need to switch them off now. Although they're a good way to experiment with a room design, they force the same scenery into every room and override the real room designs. Because these lines might include your ideas for room designs, rather than deleting them, you can comment them out so Python will ignore them. Click and drag the mouse to highlight all the lines at once, and then click **Format ▸ Comment Out Region** (or use the keyboard shortcut ALT-3). Comment symbols will be added at the start of the highlighted lines, as shown in [Listing 6-5](#):

listing6-5.py

--snip--

```
#####  
## EXPLORER ##  
#####
```

```
def draw():  
    global room_height, room_width, room_map
```

```

print(current_room)
generate_map()
screen.clear()
## room_map[2][4] = 7
## room_map[2][6] = 6
## room_map[1][1] = 8
## room_map[1][2] = 9
## room_map[1][8] = 12
## room_map[1][9] = 10
--snip--

```

Listing 6-5: Commenting out code in the `EXPLORER` section

Now we need to make a small change to the code that displays the room so it doesn't try to draw an image for a floor space marked with 255. That space will be covered by an image to the left of it, and we don't have an entry in the `objects` dictionary for 255.

Listing 6-6 shows the new line you need to add to the `EXPLORER` part of the program where indicated. The `if` statement makes sure the instructions that draw an object run only if the object number is not (!=) 255.

After adding the line, indent the existing code that comes after it by four spaces. The indentation tells Python that those instructions belong to the `if` instruction. You can either type four spaces at the start of the next two lines, or you can highlight them and click **Format ▸ Indent Region**.

listing6-6.py

```

--snip--

#####
## EXPLORER ##
#####

--snip--

for y in range(room_height):
    for x in range(room_width):
        if room_map[y][x] != 255:
            image_to_draw = objects[room_map[y][x]][0]

```

```
screen.blit(image_to_draw,  
            (top_left_x + (x*30),  
             top_left_y + (y*30) - image_to_draw.get_height()))
```

--snip--

Listing 6-6: Updating the Explorer so it doesn't try to show image 255

Now you're ready to take a tour of the base. Save the program as *listing6-6.py* and run it by entering `pgzrun listing6-6.py`. Use the arrow keys to move around the map and familiarize yourself with the layout of the space station. As before, the *Explorer* program allows you to move any direction around the map, even if a wall would block your path when playing the game.

All the scenery should be in place in the rooms. Wide objects should display correctly now, and you should be able to view all the rooms again because of the changes you made earlier in [Listing 6-5](#). Some objects will still have a black square under them because there's no floor tile underneath, but we'll fix that in [Chapter 8](#).

The space station map and scenery are now complete. It's time to move into the space station. In the next chapter, you'll teleport down to the surface and set foot on Mars at last.

TRAINING MISSION #1

Can you add your own room design to the scenery data? Room number 43 has been left empty for you to fill. It is 9×9 tiles in size, so you can place objects in positions 1 to 7 in each direction (remember the wall!). You could base your design on a room you created previously in the *Explorer* in [Chapter 5](#) or invent a new layout. Remember that you need to turn off the `assert` instructions to stop the checksum complaining when the `scenery` numbers don't add up.

Your program's `objects` dictionary (shown in [Chapter 5](#)) tells you the number of each object. Use object numbers between 1 and 47 to ensure that you don't create any problems now that might affect the code when you complete and play the *Escape* game later.

If you get stuck, try building my example, which is shown in the [Mission Debrief](#) on [page 110](#). Change the value of `current_room` in the `VARIABLES` section to 43 so you can see your redesigned room when you first run the program.

Remember to change `current_room` back to 31 when you've finished.

ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ Items that cannot move during the *Escape* game are called *scenery*.
- ☐ The `scenery` dictionary uses the room number as its key and provides a list of the fixed items in each room.
- ☐ Each scenery item is stored as a list containing the object number, *y* position, and *x* position.
- ☐ Checksums check whether the data has been changed or entered incorrectly.
- ☐ Loops can be used to add items to the `scenery` dictionary. Some scenery can be positioned randomly, too.
- ☐ The `generate_map()` function takes items that are in the current room from the `scenery` dictionary and puts them in the `room_map` list. Then the items can be displayed in the room.
- ☐ The number 255 in `room_map` represents a space that is covered by a wide object, when the object doesn't start in that space.

MISSION DEBRIEF

Here is the answer for the training mission in this chapter.

TRAINING MISSION #1

To use my design for room 43, add the line shown here to the scenery dictionary. The result looks like Figure 6-4.

```
--snip--
41: [[4,3,1], [10,3,5], [4,5,1], [10,5,5], [4,7,1],
    [10,7,5], [12,1,1], [12,1,5]],
43: [[18,1,1], [18,1,4], [14,1,6], [52,4,5], [52,4,2]],
44: [[46,4,3], [46,4,5], [18,1,1], [19,1,3],
    [19,1,5], [52,4,7], [14,1,8]],
--snip--
```

To stop the checksum from halting the program, you also need to comment out the two assert instructions in the SCENERY part of the program:

```
--snip--
print(check_counter, "scenery data items")
##assert check_counter == 161, "Expected 161 scenery items"
##assert checksum == 200095, "Error in scenery data"
print("Scenery checksum: " + str(checksum))
--snip--
```

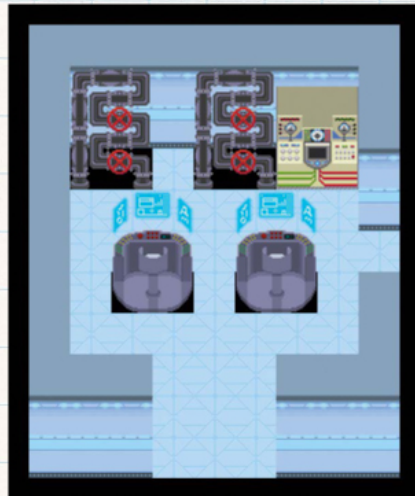


Figure 6-4: My custom design for room 43

7

MOVING INTO THE SPACE STATION



Now that we've outfitted the space station with scenery, life support systems, and other equipment, it's time to move in. In this chapter, you'll see yourself in the space station for the first time, and you'll be able to move around and explore the rooms. You might feel a bit stiff from the journey to begin with, but you'll soon be walking all over the base.

You'll discover how to animate the astronaut and use the keyboard controls to move them around. You'll also add code to enable the astronaut to move between rooms. Is there life on Mars? There is now.

ARRIVING ON THE SPACE STATION

We'll use [Listing 6-6](#) from [Chapter 6](#) as a starting point in this chapter, so open *listing6-6.py*. We'll add code to show you in your space suit in the space station. Eventually, you'll be able to move around using the arrow keys.

DISABLING THE ROOM NAVIGATION CONTROLS IN THE EXPLORER SECTION

So far, we've been using the arrow keys in the `EXPLORER` section to show different rooms on the map. We're going to start using those keys to move the astronaut around the

rooms. First, we need to disable the existing controls. Scroll down to the `EXPLORER` part of the program and highlight the instructions shown in [Listing 7-1](#). Click **Format ▶ Comment Out Region** to turn those instructions into comments so the program will ignore them. (You can also just delete them if you prefer.) Save your program as *listing7-1.py*.

listing7-1.py

```
--snip--
##def movement():
##    global current_room
##    old_room = current_room
##
##    if keyboard.left:
##        current_room -= 1
##    if keyboard.right:
##        current_room += 1
##    if keyboard.up:
##        current_room -= MAP_WIDTH
##    if keyboard.down:
##        current_room += MAP_WIDTH
##
##    if current_room > 50:
##        current_room = 50
##    if current_room < 1:
##        current_room = 1
##
##    if current_room != old_room:
##        print("Entering room:" + str(current_room))
##
##clock.schedule_interval(movement, 0.08)
--snip--
```

Listing 7-1: Turning off the keyboard controls in the `EXPLORER` section

Now we can add code that uses the arrow keys to move the astronaut.

ADDING NEW VARIABLES

Let's start by setting up some variables. The most important of these are your starting

coordinates where you'll teleport in. As before, we add variables to the `VARIABLES` part of the program, near the start. Add the new lines in [Listing 7-2](#). Save your program as `listing7-2.py`.

listing7-2.py

```
--snip--
TILE_SIZE = 30

❶ player_y, player_x = 2, 5
❷ game_over = False

❸ PLAYER = {
    "left": [images.spacesuit_left, images.spacesuit_left_1,
             images.spacesuit_left_2, images.spacesuit_left_3,
             images.spacesuit_left_4
            ],
    "right": [images.spacesuit_right, images.spacesuit_right_1,
              images.spacesuit_right_2, images.spacesuit_right_3,
              images.spacesuit_right_4
             ],
    "up": [images.spacesuit_back, images.spacesuit_back_1,
           images.spacesuit_back_2, images.spacesuit_back_3,
           images.spacesuit_back_4
          ],
    "down": [images.spacesuit_front, images.spacesuit_front_1,
             images.spacesuit_front_2, images.spacesuit_front_3,
             images.spacesuit_front_4
            ]
}

❹ player_direction = "down"
❺ player_frame = 0
❻ player_image = PLAYER[player_direction][player_frame]
   player_offset_x, player_offset_y = 0, 0

--snip--
```

Listing 7-2: Adding player variables

The `VARIABLES` section already includes a value for `current_room`, which is the room you'll start in. (If you changed the value of `current_room` while experimenting in [Chapter 6](#), make sure you change it back to 31.) We make new `player_y` and `player_x` variables ❶ to contain numbers for your starting position in the room. Here, we're setting up two variables in a single line. The numbers are put into the variables in the same order they're listed, so 2 goes into `player_y` (the first number goes into the first variable), and 5 goes into `player_x`. These variables will change as you move around the rooms on the space station and will be used to check where you are and draw you in the correct place. Your position is measured using the same tile coordinates as for the scenery positions.

We also set up a `game_over` variable ❷ to tell the program whether the game has ended. At the start of the program, we set the variable to `False`. It will stay `False` until the game ends and then become `True`. The program checks this variable to see whether the player is allowed to move. It would be odd if the player kept moving after they died!

Next, we'll set up the images for the player's walking animation. Animation is a trick of the eye. You start with a series of similar pictures with slight differences that show small movements. When you switch between them quickly, you can fool the eye into thinking the image is moving. In our game, we'll use a series of images of the astronaut walking that show the legs in different positions. When we switch between them quickly, the astronaut's legs will look like they're moving.

TIP

The key to making animation work is to make sure the images are similar enough. If the images are too different, the effect doesn't work.

Each image in an animation is known as a *frame*. [Table 7-1](#) shows the animation frames we'll use. We'll number our frames starting at 0, which will be the resting position when the astronaut isn't walking. When the player is walking up the screen, we see their back because they're walking away from us in the room.

Table 7-1: The Animation Frames for the Astronaut

| Key | Frame 0 | Frame 1 | Frame 2 | Frame 3 | Frame 4 |
|-----|---------|---------|---------|---------|---------|
|-----|---------|---------|---------|---------|---------|

left



right



up



down



The `PLAYER` dictionary ❸ stores the animation frames. The direction names—up, down, left, and right—are the dictionary keys. Each dictionary entry is a list that has the image of the player standing, plus four animation frames for that direction of walking (see

Table 7-1). The `PLAYER` dictionary will be used together with the direction the player is facing ④ and the number of the animation frame ⑤ to display the correct image as the player walks or stands still. The `player_image` variable ⑥ stores the current image of the astronaut.

TIP

Appendix B at the back of the book describes the important variables in the *Escape* program, so look there if you can't remember what a particular variable does.

TELEPORTING ONTO THE SPACE STATION

Get ready to beam down! With the starting coordinates in place, let's add the code to make you appear in the space station.

Listing 7-3 shows the lines you need to add to the `EXPLORER` part of the program. As before, you only need to add the new lines. Don't change the other lines. Just use them to find your way around the program code. The first new line ① is indented by eight spaces because it's inside a function and also inside a loop. Save your program as *listing7-3.py*.

listing7-3.py

```
--snip--
for y in range(room_height):
    for x in range(room_width):
        if room_map[y][x] != 255:
            image_to_draw = objects[room_map[y][x]][0]
            screen.blit(image_to_draw,
                (top_left_x + (x*30),
                 top_left_y + (y*30) - image_to_draw.get_height()))
①         if player_y == y:
②             image_to_draw = PLAYER[player_direction][player_frame]
③             screen.blit(image_to_draw,
                (top_left_x + (player_x*30)+(player_offset_x*30),
                 top_left_y + (player_y*30)+(player_offset_y*30))
```

```
- image_to_draw.get_height()))
```

```
--snip--
```

Listing 7-3: Drawing the player in the room

These new instructions draw you in the room. The `y` loop draws the room from back to front. The `x` loop draws the scenery in each row from left to right.

After each row is drawn, the program checks whether the player is standing in that row ❶. This instruction should be lined up with the `for x in range(room_width)` line rather than indented further, because it's not inside the `x` loop. It will run once, after the `x` loop has finished.

If the player *is* in the row the program has just drawn, the next line ❷ puts the picture of the player into the variable `image_to_draw`. The image is taken from the `PLAYER` dictionary of animation frames, using the player's direction and the animation frame number.

The last new line ❸ draws the player using the `image_to_draw` variable you just set up, which contains the picture. It also uses the player's `x` and `y` position variables to work out where to draw the image on the screen. [Chapter 3](#) explains how the position onscreen is calculated (see “[Working Out Where to Draw Each Item](#)” on [page 56](#)). The `player_offset_x` and `player_offset_y` variables were set up in [Listing 7-2](#) and are used to position the player partway between tiles as they walk between them. You'll learn more about these variables shortly.

Get ready to teleport! Brace yourself! Take a deep breath.

Run your program using `pgzrun listing7-3.py`. If your teleportation was successful, you should be on the space station (see [Figure 7-1](#)). If not, check the program changes you made in this chapter.

One side effect of teleporting is that at first you can't move. As we add more code, you'll find that the side effect wears off.

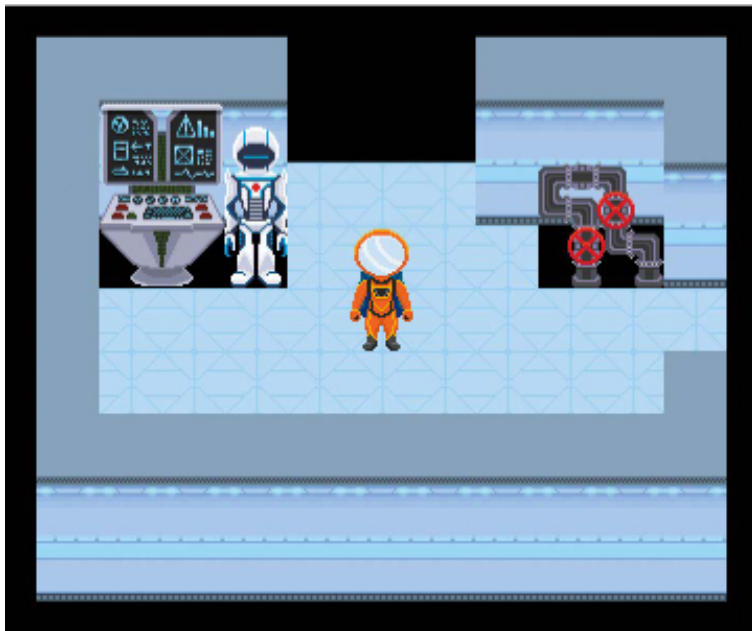


Figure 7-1: The astronaut arrives!

ADDING THE MOVEMENT CODE

Now we'll add a totally new section called `GAME LOOP`. This is the heart of the program. The `game_loop()` function will run several times a second and enable you to move. Later in the book, we'll add more instructions here that enable you to do things with the objects you find.

Add this new section between the `MAKE MAP` and `EXPLORER` sections. [Listing 7-4](#) shows you what it looks like. Save the program as *listing7-4.py*.

listing7-4.py

```
--snip--
    for tile_number in range(1, image_width_in_tiles):
        room_map[scenery_y][scenery_x + tile_number] = 255

#####

## GAME LOOP ##

#####

❶ def game_loop():
❷     global player_x, player_y, current_room
    global from_player_x, from_player_y
    global player_image, player_image_shadow
    global selected_item, item_carrying, energy
```



```
global player_offset_x, player_offset_y
global player_frame, player_direction
```

```
③ if game_over:
    return

④ if player_frame > 0:
    player_frame += 1
    time.sleep(0.05)
    if player_frame == 5:
        player_frame = 0
        player_offset_x = 0
        player_offset_y = 0

⑤ # save player's current position
    old_player_x = player_x
    old_player_y = player_y

⑥ # move if key is pressed
    if player_frame == 0:
        if keyboard.right:
            from_player_x = player_x
            from_player_y = player_y
            player_x += 1
            player_direction = "right"
            player_frame = 1
        elif keyboard.left: #elif stops player making diagonal movements
            from_player_x = player_x
            from_player_y = player_y
            player_x -= 1
            player_direction = "left"
            player_frame = 1
        elif keyboard.up:
            from_player_x = player_x
            from_player_y = player_y
            player_y -= 1
            player_direction = "up"
            player_frame = 1
        elif keyboard.down:
            from_player_x = player_x
```

```

    from_player_y = player_y
    player_y += 1
    player_direction = "down"
    player_frame = 1

7  # If the player is standing somewhere they shouldn't, move them back.
    # Keep the 2 comments below - you'll need them later
    if room_map[player_y][player_x] not in items_player_may_stand_on: #\
        # or hazard_map[player_y][player_x] != 0:
            player_x = old_player_x
            player_y = old_player_y
8    player_frame = 0

9    if player_direction == "right" and player_frame > 0:
        player_offset_x = -1 + (0.25 * player_frame)
    if player_direction == "left" and player_frame > 0:
        player_offset_x = 1 - (0.25 * player_frame)
    if player_direction == "up" and player_frame > 0:
        player_offset_y = 1 - (0.25 * player_frame)
    if player_direction == "down" and player_frame > 0:
        player_offset_y = -1 + (0.25 * player_frame)

#####
## EXPLORER ##
#####

--snip--

```

Listing 7-4: Adding player movement

At the very end of the program, you also need to add a new section called `START`, which will make the `game_loop()` function run every 0.03 seconds. [Listing 7-5](#) shows you the lines to add. This instruction isn't indented, because it doesn't belong to a function. Python runs the instructions that *aren't* inside a function in the order they appear in the program, from top to bottom. This instruction runs after all the variables, map, scenery, and prop data have been set up and the functions have been defined in the instructions above. Save your program as *listing7-5.py*.

[listing7-5.py](#)

```
--snip--  
#####  
##  START  ##  
#####
```

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

```
--snip--
```

Listing 7-5: Setting the `game_loop()` function to run regularly

Run the program using `pgzrun listing7-5.py`. You should be in the room (as shown in [Figure 7-1](#)) and be able to move using the arrow keys! You might notice your legs disappear when you walk up the screen. This is a side effect of teleportation that will wear off when we improve the code for drawing rooms in [Chapter 8](#).

At this point, the program won't work properly if you walk out the door, but it should stop you from walking through walls or furniture. If you *can* walk through objects, double-check the new code you just added. If you still have problems, carefully check the line that sets up the `items_player_may_stand_on` list at the end of the `OBJECTS` part of the program.

UNDERSTANDING THE MOVEMENT CODE

If you want to play the game and customize it with your own designs, you don't need to understand how the code in this chapter works. You can simply replace the images and the data for maps, scenery, and props. This movement code, and the code for moving between rooms, which you'll add later in this chapter, should keep working. However, if you want to understand how the code works and want to see how you could add animation to your programs, I'll break it down now. This code is the real engine of the game, so in many ways it's the most exciting bit!

If you're getting a sense of déjà vu, it's because you've already seen much of this code. In [Chapter 2](#), for your spacewalk, you used code to change the player's position using keyboard controls and a function called `game_loop()` to control movement. Let's refresh our memories and see what's new in [Listing 7-4](#).

In [Listing 7-4](#), we define a function called `game_loop()` ❶ at the start of this new section. The `clock.schedule_interval()` function we added at the end of the program (see [Listing 7-5](#)) makes this `game_loop()` function run every 0.03 seconds. Each time the `game_loop()`

function runs, it checks whether you've pressed an arrow key or are walking and, if so, updates your position.

At the start of `game_loop()`, we tell Python which variables are global variables ❷ (see [“Understanding the Spacewalk Listing”](#) on page 27 for a refresher on why we need to do this). Some of these aren't used yet, but we'll need them later.

Then we check the `game_over` variable. If it's set to `True` ❸, the `game_loop()` function finishes without running any of its other instructions because the game is over. This variable stops the player from moving when the game ends. For now, it won't do anything, because nothing in our program causes the game to end.

The `game_loop()` function checks whether the player is already walking ❹. It takes four animation frames to walk one tile across the screen. If the player is moving, the `player_frame` variable contains a number between 1 and 4, which represents the animation frame being used. If the player is walking, the program increases the `player_frame` variable by 1 to move to the next animation frame. That means the `draw()` function in the `EXPLORER` section will show the next animation frame the next time it runs.

When `player_frame` reaches 5, it means all the animation frames have been shown and the animation has ended. In that case, the program resets `player_frame` to 0 to end the animation. When the animation ends, the program also resets the `player_offset_x` and `player_offset_y` variables. I'll tell you what these do in a minute.

Next, we see whether the player has pressed a key to start a new walking animation. Before we let the player move, we save their current position ❺ by storing the *x* position in the variable `old_player_x` and the *y* position in the variable `old_player_y`. We will use these variables to move the player back if they try to walk somewhere they shouldn't, such as into a wall pillar.

The program then uses a familiar block of code to change the player's *x* and *y* position variables if an arrow key is pressed ❻. We measure the player's position in tiles, the same units we use for positioning the scenery. This is different from when we used pixels as the measurement in [Chapter 1](#).

When the player presses the right arrow key, the program adds 1 to the *x* position. If the player presses the left arrow key, it subtracts 1. We use similar code to change the *y* position if the player presses the up or down arrow keys.

When the player moves, the global variables `from_player_x` and `from_player_y` store the

position the player is walking from. These variables will be used later to check whether the player has been hit by a hazard while walking. The `player_direction` variable is also set to the direction they're moving, and the `player_frame` is set to 1, the first frame in the animation sequence.

As in [Chapter 1](#), we use `elif` to combine our checks for a keypress. This ensures the player cannot change the *x* and *y* positions at the same time to move diagonally. In our 3D room, walking diagonally would enable the player to walk through obstacles, squeezing through impossible gaps.

After moving the player, we check whether the new position puts them somewhere they're allowed to be [7](#). We do this by using `room_map` to see what item is in the position they're standing at and checking it against the list `items_player_may_stand_on`. There is some code I've commented out here too, which we'll need to enable later to stop players from walking through hazards.

We can use the keyword `in` to check whether something is in a list. By using the keyword `not` with it, we can see whether something is missing from a list. The following line means "If the number in the map where the player is standing isn't in the list of items, the player is allowed to stand on . . ."

```
if room_map[player_y][player_x] not in items_player_may_stand_on:
```

If the player is standing on something that isn't in the `items_player_may_stand_on` list, we reset the player's *x* and *y* positions to their position before they moved.

All of this happens so fast that the player doesn't notice anything. If they try to walk into a wall, it looks like they never went anywhere! This is a simpler way of stopping the player from walking through walls than checking whether each movement is allowed before making it.

The program also sets the `player_frame` variable to 0 if the player's position must be reset [8](#). This turns off the player animation again.

When you press the right arrow key, the astronaut steps one tile to the right. It takes four frames to animate this, so the astronaut is displayed at positions that are partway across the tile while this animation plays out. The `player_offset_x` and `player_offset_y` variables are used to work out where to draw the astronaut. These variables are calculated at the end of the `game_loop()` function [9](#). The `draw()` function (see [Listing 7-3](#)) multiplies the offset values by the size of a tile (30 pixels) because images are drawn in

pixels. For example, if the offset is 0.25 tiles, the astronaut is drawn roughly 7 pixels away from the center of the new tile. The computer will round the number because you can't position something using half a pixel.

Look at the left side of [Figure 7-2](#). For the first animation frame when the astronaut is walking left, we need to add three-quarters of a tile to the player's new tile position (0.75). For the second animation frame, we add half a tile (0.5) to the player's new tile position before drawing it. For the third animation frame, we add a quarter of a tile to the player's new tile position.



Figure 7-2: Understanding how the astronaut is positioned during the animation

We can calculate these offset numbers using the frame number. Here's the calculation for walking left:

$$\text{player_offset_x} = 1 - (0.25 * \text{player_frame})$$

Check that this calculation makes sense by working out the numbers on your own. For example, here is the calculation when the animation frame is 2:

$$0.25 \times 2 = 0.5$$

$$1 - 0.5 = 0.5$$

In [Figure 7-2](#), 0.5 is the correct offset for frame 2.

When the player walks right, we need to subtract part of a tile from the player's position, so the offsets are negative. Look at the right side of [Figure 7-2](#). For frame 1, adding -0.75 puts the astronaut three-quarters of a tile to the *left* of their new position.

We can work out the x offset for walking right using the frame number too. Here's the formula:

```
player_offset_x = -1 + (0.25 * player_frame)
```

TRAINING MISSION #1

Can you check that the formula works? Use it to find the offset values for frames 1 and 3, and check that they match the offset values in [Figure 7-2](#).

The offsets for the y direction work the same. When the astronaut is moving up, we calculate the y offset using the same formula as the left offset. When the astronaut is moving down, we calculate the y offset using the same formula as the right offset.

In summary, the `game_loop()` function does this:

- If you're not walking, it starts the walking animation when you press a key.
- If you are walking, it works out the next animation frame and the positions partway across the tile to use when drawing you.
- If you've reached the end of the animation sequence, it resets it so you can move again. The movement is fluid, so if you hold down a key, you'll cycle through animation frames 1 to 4 and won't see the standing position until you stop walking.

MOVING BETWEEN ROOMS

Now that you're on your feet, you'll want to explore the space station fully. Let's add

some code to the `game_loop()` function that lets you walk into the next room. Add the new code in [Listing 7-6](#), which goes after we check for keypresses and before we check whether the player is standing somewhere they shouldn't be. Make sure you include the instructions with comment symbols (`#`) at the start. We'll need them later.

The grayed-out lines in [Listing 7-6](#) show you where to add the new code. Save your program as *listing7-6.py*. Run it using `pgzrun listing7-6.py` and then walk around the space station! This is a good time to look around, before the doors are fitted and certain areas are locked down.

listing7-6.py

```
--snip--

def game_loop():

--snip--
    player_direction = "down"
    player_frame = 1
    # check for exiting the room
❶ if player_x == room_width: # through door on RIGHT
    #clock.unschedule(hazard_move)
❷    current_room += 1
❸    generate_map()
❹    player_x = 0 # enter at left
❺    player_y = int(room_height / 2) # enter at door
❻    player_frame = 0
❼    #start_room()
❽    return

❾ if player_x == -1: # through door on LEFT
    #clock.unschedule(hazard_move)
    current_room -= 1
    generate_map()
    player_x = room_width - 1 # enter at right
    player_y = int(room_height / 2) # enter at door
    player_frame = 0
    #start_room()
    return
```

```

10 if player_y == room_height: # through door at BOTTOM
    #clock.unschedule(hazard_move)
    current_room += MAP_WIDTH
    generate_map()
    player_y = 0 # enter at top
    player_x = int(room_width / 2) # enter at door
    player_frame = 0
    #start_room()
    return

if player_y == -1: # through door at TOP
    #clock.unschedule(hazard_move)
    current_room -= MAP_WIDTH
    generate_map()
    player_y = room_height - 1 # enter at bottom
    player_x = int(room_width / 2) # enter at door
    player_frame = 0
    #start_room()
    return

# If the player is standing somewhere they shouldn't, move them back.
if room_map[player_y][player_x] not in items_player_may_stand_on: #\
#     or hazard_map[player_y][player_x] != 0:
    player_x = old_player_x
--snip--

```

Listing 7-6: Enabling the player to move between rooms

To see how this code works, let's use an example room map. [Figure 7-3](#) shows a room 9 tiles wide and 9 tiles high with exits on each wall. We'll use this image to understand the player's position when they've left the room.

As you know, the positions on the map are numbered starting at 0 in the top left. The yellow squares show where the player might be if they walked out of the room:

- If the player's *y* position is -1 , they've walked out of the top exit.
- If the player's *x* position is -1 , they've walked out of the left exit.

- If the player's y position is the same as the `room_height` variable, they've walked out of the bottom. The tile positions are numbered starting at 0, so if the player goes into row 9 in a room that has 9 rows, they've already left the room.
- Similarly, if the player's x position is the same as the `room_width` variable, they've walked out of the right exit.

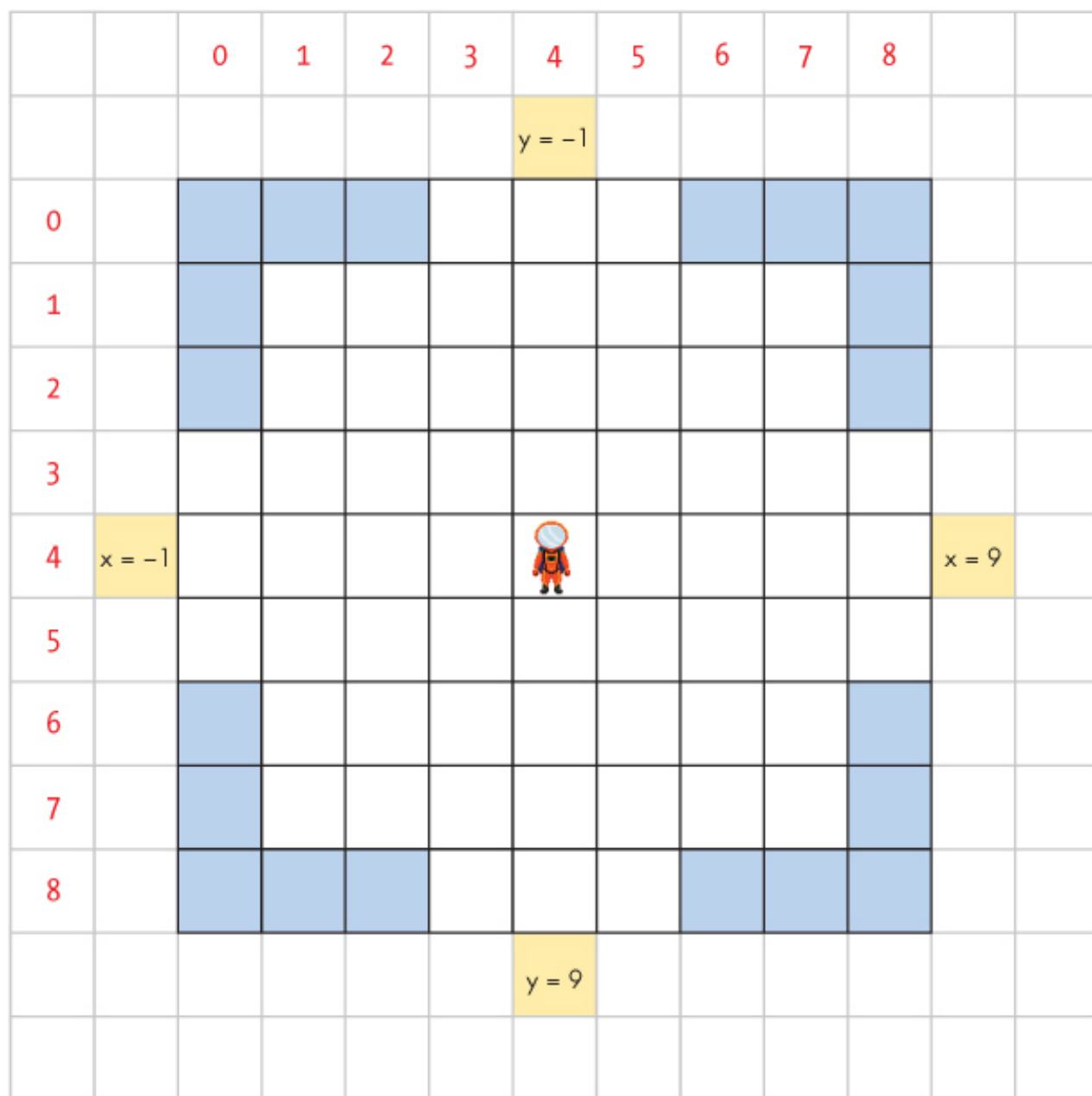


Figure 7-3: Working out whether the player has walked through an exit

The new code lines check whether the player position means they've walked out of the room. If the player's x position is the same as `room_width` ❶, they're outside the door on the right, as shown in Figure 7-3.

When a player leaves the room, we need to change the number of the room they're in, which is stored in the `current_room` variable. When they go through a door on the right,

the room number increases by 1 ❷. Look at the room map again (flip back to [Figure 4-1](#) on [page 60](#)) to see that this makes sense: room numbers increase from left to right. For example, if the player is in room 33 and walks through the exit on the right, they end up in room 34.

The program then generates a new `room_map` list ❸ to use in displaying and navigating the new room. The player is repositioned at the opposite side of the room ❹, so it looks like they've walked through the doorway. If the player exits to the right of the room, they enter the next room from the left ❺.

Rooms are lots of different sizes, so we also need to change the player's *y* position to put them in the middle of the doorway. Otherwise, the player might emerge from a wall! We set the player's position to be half the height of the room ❻, which means they're right in the middle of the doorway. When they enter the room, we reset the player animation, too ❼.

I've included a couple of features here that we'll need later, so make sure you include the `clock.unschedule(hazard_move)` ❶ and `start_room()` ❼ instructions. The `start_room()` function will display the room name when the player enters a new room. We'll talk about those instructions more later in the book.

Finally, the `return` instruction exits the `game_loop()` function ❽. Any further instructions in the function won't run this time around. When the function starts again, it will start from the top as usual.

The next code block ❾ checks whether the player went through the left door. To go through the left door, the program does the following:

- Checks whether the `player_x` variable contains -1 (see [Figure 7-3](#)).
- Subtracts 1 from the current room number to go into the room on the left.
- Sets the player's *x* position to be just inside the doorway on the right. This position is the `room_width` minus 1. (You can check this in [Figure 7-3](#). In a room that has a `room_width` of 9, the player's *x* position should be 8.)
- Sets the player's *y* position to the middle using the `room_height`. This is the same approach as walking through the right exit.

The same code structure is used for the top and bottom exits ❿. However, the program

checks the player's *y* position to see if they used an exit and sets their new position to enter through a top or bottom doorway.

This time, we change the room number by 5 instead of 1 because that's how many rooms wide the game map is (see [Figure 4-1](#)). For example, if you're in room 37 and you go through the top exit, you end up in room 32 (which is 37 minus 5). If you're in room 37 and go through the bottom exit, you end up in room 42 (37 plus 5). We stored the number 5 in the variable `MAP_WIDTH` earlier, and the program uses it here.

Now you're able to freely explore the space station. In the next chapter, we'll fix the remaining few bugs in the room display.

ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ The player's position in the *Escape* game is measured in tiles, just like the scenery.
- ☐ The `game_loop()` function controls player movement and is scheduled to run every 0.03 seconds.
- ☐ If the player moves somewhere they aren't allowed to be, they're put back in their previous position so fast you won't see them move.
- ☐ The program checks the player's *x* and *y* positions to see whether they've walked out of an exit. If they have, they'll appear in the middle of the opposite exit in the next room.
- ☐ The animation frames are stored in the `PLAYER` dictionary and have a list of images for each direction. The dictionary key is the direction name, and an index number gets the particular frame needed.
- ☐ Frame 0 is the standing-still position. Frames 1, 2, 3, and 4 show the astronaut walking.
- ☐ The `game_loop()` function increases the animation frame number used when the player is walking.
- ☐ The `player_offset_x` and `player_offset_y` variables are used to position the astronaut correctly when walking into a new tile.

MISSION DEBRIEF

Here is the answer for the training mission in this chapter.

TRAINING MISSION #1

For frame 1:

$$0.25 \times 1 = 0.25$$

$$-1 + 0.25 = -0.75$$

For frame 3:

$$0.25 \times 3 = 0.75$$

$$-1 + 0.75 = -0.25$$

8

REPAIRING THE SPACE STATION



While wandering around the space station, you must have noticed that some things don't look quite right. To get the program up and running quickly, we used the `EXPLORER` section to display the rooms. However, it has a few drawbacks:

- Sometimes a blank space is shown beneath the scenery because there's no floor there.
- When you walk to the front of the room, the front wall hides the astronaut.
- The astronaut's legs disappear when walking to the back of the screen.
- The rooms are all drawn in the top left of the game window. This makes it look uneven and inconsistent, because there's much more space on the right of the rooms than on the left, and wider rooms leave less space on the right than narrow rooms do.
- There are no shadows, making it harder to understand the position of objects in the room.

In this chapter, we'll fix these glitches and also add a function for displaying messages

at the top of the window. These messages will give players information about the space station and their progress in the game.

As you read through the chapter, you'll learn how to send information to a Python function and discover how to draw rectangles using Pygame Zero. By the end of the chapter, the space station will look great!

SENDING INFORMATION TO A FUNCTION

For the first time, we'll need to send information to a function. You've already seen how to send information to the `print()` function by putting it between the parentheses. For example, you can output a message like this:

```
print("Learn your emergency evacuation drill")
```

When that instruction runs, the `print()` function receives information you put in the brackets, and displays it in the command line window or the Python shell.

We can also send information to functions we've made.

CREATING A FUNCTION THAT RECEIVES INFORMATION

To experiment with functions, we'll build a function that adds two numbers that we send it. Click **File** ▶ **New** to open a new window, and enter the program in [Listing 8-1](#).

listing8-1.py

```
❶ def add(first_number, second_number):  
❷     total = first_number + second_number  
❸     print(first_number, "+", second_number, "=", total)  
  
❹ add(5, 7)  
   add(2012, 137)  
   add(1234, 4321)
```

Listing 8-1: Sending information to a function

Save the program as *listing8-1.py*. Because it doesn't use any Pygame Zero features, you can run it by clicking **Run** ▶ **Run Module** or by pressing F5. (If you do run it using Pygame Zero, the results will appear in the command line window, and the game

window will be empty.)

When you run the program, you should see the following output:

5 + 7 = 12

2012 + 137 = 2149

1234 + 4321 = 5555

We create a new function called `add()` ❶. After we've defined `add()`, we can run it by using its name ❷ and send it numbers by putting them in the parentheses, using commas between them ❸. The function will then add those two numbers.

HOW IT WORKS

To enable the function to receive the numbers, we give it two variables to store the numbers in when we define it. I've called them `first_number` and `second_number` ❶ to make the program easier to understand, but the variable names could be anything. These are local variables: they only work inside this function.

When you use the function, it takes the first item it receives and puts it into the variable `first_number`. The second item goes into `second_number`.

Of course, it doesn't matter which order you add two numbers in, so it doesn't matter what order you send the numbers in. The instructions `add(5, 7)` and `add(7, 5)` give the same result. But some functions will need you to send the information in the same order the function expects to receive it. For example, if the function were subtracting numbers, you'd get a different result if you sent the numbers in the wrong order. The only way to know what information a function expects to receive is to take a look at its code.

The body of the function is quite simple. It creates a new variable called `total`, which stores the result of adding the two numbers ❷. The program then prints a line that contains the first number, a plus sign, the second number, an equal sign, and the total ❸.

In the last three instructions, we send the function three pairs of numbers to add ❹.

This simple demonstration shows you how information (or *arguments*) can be sent to a function. You can make functions that take more arguments than just two, and even take lists, dictionaries, or images. Functions make it easy to reuse sets of instructions,

and sending arguments means we can reuse those instructions with different information. For example, [Listing 8-1](#) uses the same `print()` instruction three times, to display the sum of three different number pairs. In this case, we've avoided repeating the `print()` instruction and the one that sets up the `total` variable. More sophisticated functions can avoid repeating a lot of code, and this can make the program much easier to write and understand.

TRAINING MISSION #1

Try modifying the program to subtract one number from another rather than adding. What happens when you change the order of the numbers you send to the new function? You might want to change more than just the calculation to make sure the function is easy to use.

Now we're ready to add some new functions to *Escape* to draw objects on the space station.

ADDING VARIABLES FOR SHADOWS, WALL TRANSPARENCY, AND COLORS

To fix our space station, we'll create new display functions for the *Escape* game, using our newfound knowledge of functions. Before we make these new functions, we need to set up new variables for the functions to use.

Open *listing7-6.py*, the last listing you saved in [Chapter 7](#). Find the `VARIABLES` section near the start of the program, and add the new lines shown in [Listing 8-2](#). Save the program as *listing8-2.py*. As always, it's a good idea to run the program (using `pgzrun listing8-2.py`) to check for any new errors.

listing8-2.py

--snip--

```
#####  
## VARIABLES ##  
#####
```

--snip--

```
player_image = PLAYER[player_direction][player_frame]
player_offset_x, player_offset_y = 0, 0
```

❶ `PLAYER_SHADOW = {`

```
    "left": [images.spacesuit_left_shadow, images.spacesuit_left_1_shadow,
             images.spacesuit_left_2_shadow, images.spacesuit_left_3_shadow,
             images.spacesuit_left_3_shadow
            ],
```

```
    "right": [images.spacesuit_right_shadow, images.spacesuit_right_1_shadow,
              images.spacesuit_right_2_shadow,
              images.spacesuit_right_3_shadow, images.spacesuit_right_3_shadow
             ],
```

```
    "up": [images.spacesuit_back_shadow, images.spacesuit_back_1_shadow,
           images.spacesuit_back_2_shadow, images.spacesuit_back_3_shadow,
           images.spacesuit_back_3_shadow
          ],
```

```
    "down": [images.spacesuit_front_shadow, images.spacesuit_front_1_shadow,
             images.spacesuit_front_2_shadow, images.spacesuit_front_3_shadow,
             images.spacesuit_front_3_shadow
            ]
}
```

❷ `player_image_shadow = PLAYER_SHADOW["down"][0]`

❸ `PILLARS = [`

```
    images.pillar, images.pillar_95, images.pillar_80,
    images.pillar_60, images.pillar_50
]
```

❹ `wall_transparency_frame = 0`

❺ `BLACK = (0, 0, 0)`

`BLUE = (0, 155, 255)`

`YELLOW = (255, 255, 0)`

`WHITE = (255, 255, 255)`

`GREEN = (0, 255, 0)`

`RED = (128, 0, 0)`

```
#####  
##  MAP  ##  
#####
```

--snip--

Listing 8-2: Adding the variables needed for the new display functions

We add a `PLAYER_SHADOW` dictionary ❶ that's similar to the `PLAYER` dictionary. It contains animation frames for the astronaut's shadow on the floor. As the astronaut moves, the shadow also changes shape. The `player_image_shadow` ❷ stores the astronaut's current shadow, like the `player_image` variable that stores the astronaut's current animation frame (or the standing image).

Later in this chapter, we'll add animation that fades out the front wall when you walk behind it so you can still see the astronaut. Here, we set up a list of the animation frames ❸ and a `wall_transparency_frame` variable to remember the one that's being shown now ❹. You'll learn more about how these work later on.

We've also set up some names that we can use to refer to color numbers ❺. Colors in Pygame Zero are stored as tuples. A tuple is like a list whose content you can't change, and it uses parentheses instead of square brackets. You've seen tuples used for coordinates when drawing on the screen (see [Chapter 1](#)). Colors are stored as three numbers that specify the amount of red, green, and blue in the color, in that order. The scale for each color ranges from 0 to 255. This color is bright red:

```
(255, 0, 0)
```

The red is at its maximum (255), and there's no green (0) or blue (0) in the color.

Because we've set up these color variables, we can now use the name `BLACK` instead of using the tuple `(0, 0, 0)` to represent black. Using color names will make the program easier to read.

[Table 8-1](#) shows you some of the color combinations that you might want to use in your programs. You can also try different numbers to invent your own colors.

Table 8-1: Some Example RGB Color Numbers

| |
|--|
| |
|--|

▲

| Red | Green | Blue | Description |
|-----|-------|------|---|
| 255 | 0 | 0 | Bright red |
| 0 | 255 | 0 | Bright green |
| 0 | 0 | 255 | Bright blue |
| 0 | 0 | 50 | Very dark blue (nearly black!) |
| 255 | 255 | 255 | White (all the colors at maximum strength) |
| 255 | 255 | 150 | Creamy yellow (slightly less blue than white) |
| 230 | 230 | 230 | Silver (a slightly toned-down white) |
| 200 | 150 | 200 | Lilac |
| 255 | 100 | 0 | Orange (maximum red with a dash of green) |
| 255 | 105 | 180 | Pink |

DELETING THE EXPLORER SECTION

We need to add a new `DISPLAY` section with some new functions that will improve the game's appearance onscreen. The `EXPLORER` section has enabled us to get up and running quickly, but we're going to build a new and better `draw()` function in this chapter that

replaces the one we've used so far. To avoid any problems caused by `EXPLORER` code still being in the program, we're going to remove it. Your `EXPLORER` section might have more or fewer lines than mine does in [Figure 8-1](#), depending on whether you deleted some of it in earlier chapters.

To delete the entire `EXPLORER` section, follow these steps:

1. Find the `EXPLORER` part of the program near the end of the code.
2. Click the start of the `EXPLORER` comment box, hold down the mouse button, and drag the mouse to the bottom of the section (see [Figure 8-1](#)). The section ends just above where the `START` section begins.
3. Press `DELETE` or `BACKSPACE` on the keyboard.

There's one instruction in the `EXPLORER` section that we still need: it runs the `generate_map()` function to set up the room map for the first room. You'll need to add that instruction to the end of the program as a single line, as shown in [Listing 8-3](#).

listing8-3.py

```
--snip--
#####
##  START  ##
#####

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

Listing 8-3: Generating the map for the first room

The `generate_map()` line will run after the variables have been set up and will make the map for the current room.


```
listing8-2.py - C:/Users/Sean/Documents/2018/Mission Python/9 downloads/listing8-2.py (3.5.2)
File Edit Format Run Options Window Help

#####
## EXPLORER ##
#####

def draw():
    global room_height, room_width, room_map
    generate_map()
    screen.clear()
    ##
    room_map[2][4] = 7
    room_map[2][6] = 6
    room_map[1][1] = 0
    room_map[1][2] = 9
    room_map[1][8] = 12
    room_map[1][9] = 9

    for y in range(room_height):
        for x in range(room_width):
            if room_map[y][x] != 255:
                image_to_draw = objects[room_map[y][x]][0]
                screen.blit(image_to_draw,
                    (top_left_x + (x*30),
                     top_left_y + (y*30) - image_to_draw.get_height()))
            if player_y == y:
                image_to_draw = PLAYER[player_direction][player_frame]
                screen.blit(image_to_draw,
                    (top_left_x + (player_x*30)+(player_offset_x*30),
                     top_left_y + (player_y*30)+(player_offset_y*30)
                      - image_to_draw.get_height()))

##def movement():
##
##    global current_room
##    old_room = current_room
##
##    if keyboard.left:
##        current_room -= 1
##    if keyboard.right:
##        current_room += 1
##    if keyboard.up:
##        current_room -= MAP_WIDTH
##    if keyboard.down:
##        current_room += MAP_WIDTH
##
##    if current_room > 50:
##        current_room = 50
##    if current_room < 1:
##        current_room = 1
##
##    if current_room != old_room:
##        print("Entering room:" + str(current_room))
##
##clock.schedule_interval(movement, 0.08)

#####
## START ##
#####

clock.schedule_interval(game_loop, 0.03)
```

Figure 8-1: Deleting the *EXPLORER* section

Save your new listing as *listing8-3.py* and run it using `pgzrun listing8-3.py`. If all is going to plan, you should see no error messages in the command line window. The game window shows the inky blackness of space because we haven't added the new code to draw anything yet.

ADDING THE DISPLAY SECTION

Now we'll add the new *DISPLAY* section to replace the deleted *EXPLORER* section. This section contains most of the code for updating the screen display. It includes code for drawing the room, showing messages, and changing the transparency of the front wall.

ADDING THE FUNCTIONS FOR DRAWING OBJECTS

First, we'll make some functions to draw an object, a shadow, or the player at a particular tile position. Between the `GAME LOOP` and `START` sections, add the new `DISPLAY` section shown in [Listing 8-4](#) to your program. Save this program as *listing8-4.py* and run it using `pgzrun listing8-4.py`. Again, you won't see anything in the game window yet.

If there are any errors in the command line window, you can use them to help you fix the program. It's better to test as you add code to the program than to add a lot of code and not know where the mistakes might be.

listing8-4.py

--snip--

```
if player_direction == "down" and player_frame > 0:
    player_offset_y = -1 + (0.25 * player_frame)
```

```
#####
```

```
## DISPLAY ##
```

```
#####
```

```
❶ def draw_image(image, y, x):
```

```
❷     screen.blit(
        image,
        (top_left_x + (x * TILE_SIZE),
         top_left_y + (y * TILE_SIZE) - image.get_height())
    )
```

```
❸ def draw_shadow(image, y, x):
```

```
    screen.blit(
        image,
        (top_left_x + (x * TILE_SIZE),
         top_left_y + (y * TILE_SIZE))
    )
```

```
def draw_player():
```

```
❹     player_image = PLAYER[player_direction][player_frame]
```

```
❺     draw_image(player_image, player_y + player_offset_y,
                  player_x + player_offset_x)
```

```

❸ player_image_shadow = PLAYER_SHADOW[player_direction][player_frame]
❹ draw_shadow(player_image_shadow, player_y + player_offset_y,
               player_x + player_offset_x)

#####

##  START  ##

#####

clock.schedule_interval(game_loop, 0.03)
generate_map()

```

Listing 8-4: Adding the first functions in the `DISPLAY` section

The first new function, `draw_image()` ❶, draws a given image on the screen. When we use it, we give it the image we want to draw and the *y* and *x* tile positions of the object in the room. The function will work out where on the screen to draw the image (the pixel position), based on the tile position in the room. For example, we might use the function like this:

```
draw_image(player_image, 5, 2)
```

This line draws the player image at position *y* = 5 and *x* = 2 in the room.

When we define the `draw_image()` function, we set it up to give the image the name `image`, put the *y* position into the *y* variable, and put the *x* position into the *x* variable ❶.

Although the `draw_image()` function is several lines long, its only instruction is `screen.blit()`, which draws the image at the position we specify ❷. This instruction is virtually the same as the one we used in the old `EXPLORER` section, so take a look at [Chapter 3](#) for a refresher on how it works.

TIP

Make sure all the parentheses are in the correct places. You need a pair around all the `screen.blit()` arguments and another pair around the *y* and *x* positions because they make up a single tuple. You also need a pair around the multiplication parts of the position calculations. If the program doesn't work, start checking for errors by counting the opening and closing parentheses to make sure you have the same number of each of them.

We then add a new `draw_shadow()` function ❸. This is similar to the function for drawing an image, except that the image's height is not subtracted when calculating its onscreen position. This is what places the shadow *below* the main image. [Figure 8-2](#) shows the astronaut and their shadow based on the same tile position. Remember that the *y* position given to `screen.blit()` is for the top edge of the image.

`top_left_y + (y * TILE_SIZE) - image.get_height()`

`top_left_y + (y * TILE_SIZE)`



Figure 8-2: Working out the position of the image and the shadow

The third new function, `draw_player()`, draws the astronaut. First, it puts the correct astronaut animation frame into `player_image` ❹. It then uses the new `draw_image()` function to draw it ❺. The `draw_image()` function requires the following arguments:

- The variable `player_image`, which contains the image to draw.
- The result after adding the global variables for `player_y` and `player_offset_y`. This is the *y* position in tiles, which might include a decimal part (such as 5.25).
- The result after adding `player_x` and `player_offset_x` for the *x* position in tiles. (See [“Understanding the Movement Code”](#) on [page 119](#) for more information on how the offset variables are used for animation.)

We use similar code to draw the player's shadow: the correct animation frame from the `PLAYER_SHADOW` dictionary is put into `player_image_shadow` ❻. Then the `draw_shadow()` function is used to draw it ❼. The `draw_shadow()` function uses the same tile positions as the `draw_image()` function.

DRAWING THE ROOM

Now that we've created the functions for drawing objects and the player, we can add the code to draw the room. The new `draw()` function in [Listing 8-5](#) adds shadows for scenery and the player, and fixes the visual glitches we saw previously.

Add the new code at the end of the `DISPLAY` section, save your program as *listing8-5.py*, and run it using `pgzrun listing8-5.py`. As if you've flicked the lights on, the shadows appear in front of the objects. The game won't look quite right yet because all the rooms will be drawn in the top left of the window, and sometimes a room won't be cleared properly when you leave it. We'll fix this in a moment. At this point, you shouldn't see any error messages.

listing8-5.py

--snip--

```
def draw_player():
    player_image = PLAYER[player_direction][player_frame]
    draw_image(player_image, player_y + player_offset_y,
               player_x + player_offset_x)
    player_image_shadow = PLAYER_SHADOW[player_direction][player_frame]
    draw_shadow(player_image_shadow, player_y + player_offset_y,
               player_x + player_offset_x)
```

```
def draw():
```

```
    if game_over:
        return
```

❶ **# Clear the game arena area.**

```
box = Rect((0, 150), (800, 600))
screen.draw.filled_rect(box, RED)
box = Rect((0, 0), (800, top_left_y + (room_height - 1)*30))
```

❷ `screen.surface.set_clip(box)`

```
floor_type = get_floor_type()
```

❸ **for y in range(room_height): # Lay down floor tiles, then items on floor.**

```
    for x in range(room_width):
        draw_image(objects[floor_type][0], y, x)
        # Next line enables shadows to fall on top of objects on floor
        if room_map[y][x] in items_player_may_stand_on:
            draw_image(objects[room_map[y][x]][0], y, x)
```

❹ **# Pressure pad in room 26 is added here, so props can go on top of it.**

```
    if current_room == 26:
```

```

draw_image(objects[39][0], 8, 2)
image_on_pad = room_map[8][2]
if image_on_pad > 0:
    draw_image(objects[image_on_pad][0], 8, 2)

```

```

5 for y in range(room_height):
    for x in range(room_width):
        item_here = room_map[y][x]
        # Player cannot walk on 255: it marks spaces used by wide objects.
        if item_here not in items_player_may_stand_on + [255]:
            image = objects[item_here][0]

```

```

6 if (current_room in outdoor_rooms
    and y == room_height - 1
    and room_map[y][x] == 1) or \
    (current_room not in outdoor_rooms
    and y == room_height - 1
    and room_map[y][x] == 1
    and x > 0
    and x < room_width - 1):
    # Add transparent wall image in the front row.
    image = PILARS[wall_transparency_frame]

```

```

draw_image(image, y, x)

```

```

7 if objects[item_here][1] is not None: # If object has a shadow
    shadow_image = objects[item_here][1]
    # if shadow might need horizontal tiling

```

```

8 if shadow_image in [images.half_shadow,
                    images.full_shadow]:
    shadow_width = int(image.get_width() / TILE_SIZE)
    # Use shadow across width of object.
    for z in range(0, shadow_width):
        draw_shadow(shadow_image, y, x+z)
    else:
        draw_shadow(shadow_image, y, x)

```

```

9 if (player_y == y):
    draw_player()

```

10 `screen.surface.set_clip(None)`

```
#####  
##  START  ##  
#####
```

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

Listing 8-5: The new `draw()` function

As with the movement code in [Chapter 7](#), you don't need to know how the `draw()` function works, even if you want to customize the program. I will explain the `draw()` function in the next section, so if you don't want to know how it works just yet, skip to [“Positioning the Room on Your Screen”](#) on page 141.

UNDERSTANDING THE NEW `DRAW()` FUNCTION

You can think of the new `draw()` function as a more elaborate version of the code used for the `EXPLORER` section previously. I'll give you an overview of how each bit works.

Clearing the Game Arena

The program starts by clearing the game arena ❶ where the space station will be drawn. It does this by drawing a big red rectangle, wiping out the previous screen display. The areas at the top and the bottom that give the player information are separate, so they're not changed.

There are two steps for putting a rectangle on the screen. First, you create the shape using a Pygame object called a *Rect*, which works like this (don't type this in):

```
box = Rect((left position, top position), (width, height))
```

The name can be almost anything you like, but I use the name `box` in my programs. The position and size are tuples, so they have parentheses around them.

Second, you draw the `Rect` you created on the screen by using an instruction like this (again, don't type this in):

```
screen.draw.filled_rect(box, color)
```

The first item in parentheses is the `box` Rect you previously created. The second item is the color of the rectangle you want to draw. This can be a tuple of the red, green, and blue numbers that make up the number. In [Listing 8-5](#), I've used the name `RED`, which we set up in the `VARIABLES` section earlier.

You can also use a Rect shape to create a *clipping area* ❷. This is like an invisible window through which you view the screen. If the program draws something outside the window, it can't be seen. I've set up a clipping area that's the height of the room to stop the player's shadow from spilling out of the bottom of the game when they're in the front doorway.

Drawing the Room

The room is drawn in two stages. First, the program draws the floor tiles and anything that the player can walk on ❸. Drawing them first enables scenery, the player, and shadows to be drawn on top of them. This solves the problem of black holes appearing under scenery, because there will be floor tiles in those spaces before the scenery is drawn on top.

Second, the program adds the scenery in the room, including its shadows ❹, using new loops. Because this is drawn after the floor for the whole room has been drawn, the shadows will be drawn on top of floor tiles and items on the floor. The shadows are transparent, so you can still see the object underneath the shadow. The scenery drawing loops also add transparent walls ❺ and draw the player on top of the floor ❹.

As always, the room is drawn from back to front to ensure that objects near the front of the room appear to be in front of objects near the back.

We've also added a small chunk of code for a special object that's only used in one place in the game. Room 26 has a pressure pad on the floor that you might want to drop things on when you're playing the game (maybe heavy things or things you can make heavy . . .). The special code here ensures that both the floor pad and the object on it are visible.

After the floor tiles have been drawn, the `draw()` function checks whether the current room is room 26: if it is, it draws the floor pad and then any object that is on top of it ❺.

RED ALERT

If you're customizing the game with your own map, delete this piece of code to remove the floor pad from the game. Start with the comment line ④, and remove the instructions down to (and including) the `draw_image(objects[image_on_pad][0], 8, 2)` instruction.

Making the Front Wall Transparent

When the program is drawing the front row of the room (when the `y` loop equals `room_height - 1`), it checks whether it needs to draw a semitransparent wall instead of the solid wall object taken from the room map ⑥. The semitransparent wall is used if the player is standing behind it (see [Figure 8-3](#)).

On the planet surface, the program makes the whole wall transparent. Inside the space station, a transparent wall panel is used only if it's *not* in one of the bottom corner positions (see [Figure 8-3](#)). The corners always use a solid wall panel. The reason is that it looks odd if you see the solid edge wall start in the second row from the bottom.

Later on, we'll add the code to animate the transparency on the wall, by changing the number in `wall_transparency_frame`. You won't see the semitransparent wall yet in the game.

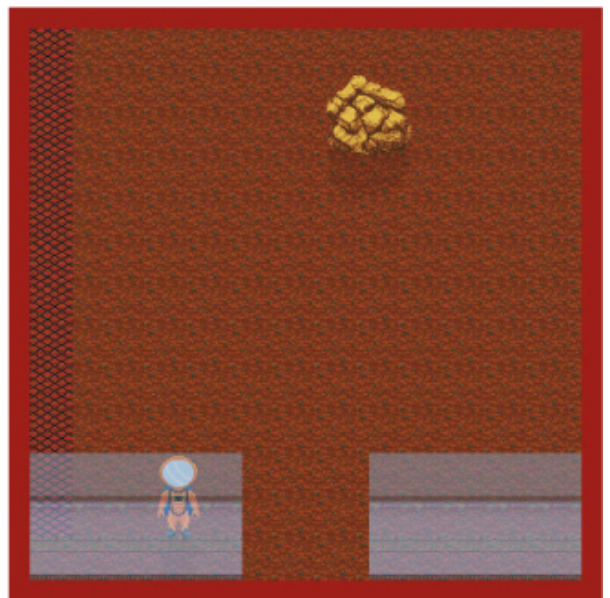


Figure 8-3: The transparent wall at the front of the room, as seen in the final game

Adding Shadows

If an object has a shadow, the shadow is taken from the `objects` dictionary and put into

`shadow_image` ⑦. Then the program checks whether it should use `half_shadow` or `full_shadow`, which fill half a tile or a whole tile respectively. These two standard shadows are used with blocky items (such as electrical units and walls) that don't need a distinctive shadow outline. The program checks whether the `shadow_image` is in a list that contains those two standard images ⑧.

That's a simple, and easy-to-read, way to check whether `shadow_image` is one of two things. If you're checking for three or more things, this technique can make the program much easier to read than having lots of `if` comparisons using `==` and combining them with `or`.

If the shadow is one of the standard images, the program then works out how wide the shadow should be in tiles. That is calculated by taking the width of the object casting the shadow and dividing it by the width of a tile (30 pixels). For example, an image that is 90 pixels wide will be 3 tiles wide.

The program then creates a loop to draw the standard shadow images, using the variable `z`. It starts at 0 and runs until the width of the shadow minus 1. That's because a `range` leaves out the last item: `range(0, 3)` would give us the numbers 0, 1, and 2. The `z` values are added to the `x` position from the main loop and are used to draw the shadow tiles. Figure 8-4 shows an object with a width of 3 tiles. The `z` loop takes the values 0, 1, and 2 to add the shadow in the correct place.

By drawing the player in position after the floor has been laid, we make sure the astronaut's legs don't disappear when walking up the screen ⑨.

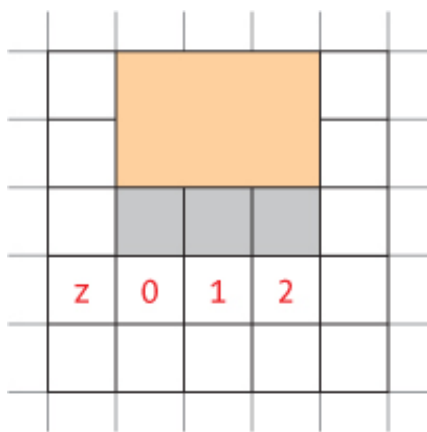


Figure 8-4: An object that is 3 tiles wide could have a standard shadow below it that is used three times.

The `draw()` function ends by turning off the clipping area that stopped shadows from spilling out of the bottom of the game area ⑩.

POSITIONING THE ROOM ON YOUR SCREEN

Now let's fix the problem of the room appearing in the top left of your screen. The program uses two variables to position the room: `top_left_x` and `top_left_y`. At the moment, these are set to 100 and 150, which means the room is always drawn in the top left of the window. We'll add some code that will change these variables depending on the size of the room so the room is drawn in the middle of the window (see [Figure 8-5](#)). The screen layout will look better, and it will make the game easier to play too.

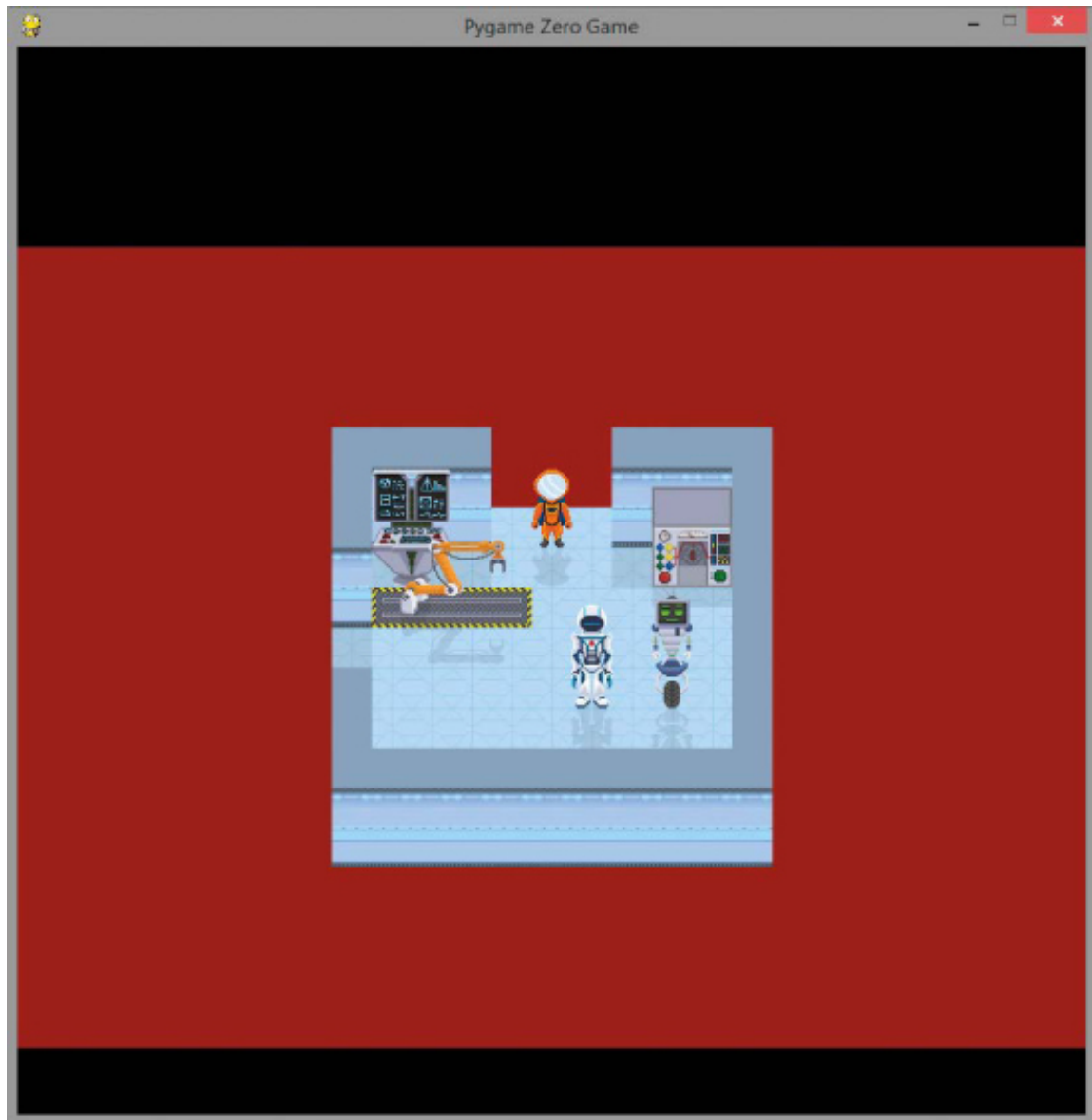


Figure 8-5: A room centered in the window

Add the new lines shown in [Listing 8-6](#) to the end of the `generate_map()` function, which is in the `MAKE_MAP` section of the program. Because they're inside a function, you need to indent each line by four spaces.

Save the program as `listing8-6.py` and run it using `pgzrun listing8-6.py`. As [Figure 8-5](#) shows, each room should be centered on the screen now.

```
--snip--
```

```
def generate_map():
```

```
--snip--
```

```
    for tile_number in range(1, image_width_in_tiles):  
        room_map[scenery_y][scenery_x + tile_number] = 255
```

- ```
❶ center_y = int(HEIGHT / 2) # Center of game window
❷ center_x = int(WIDTH / 2)
❸ room_pixel_width = room_width * TILE_SIZE # Size of room in pixels
❹ room_pixel_height = room_height * TILE_SIZE
❺ top_left_x = center_x - 0.5 * room_pixel_width
❻ top_left_y = (center_y - 0.5 * room_pixel_height) + 110
```
- 

*Listing 8-6: Creating variables to put the room in the middle of the game window*

These instructions are inside the `generate_map()` function, which sets up the `room_map` list for each room when the player enters it. The `generate_map()` function now also sets up the `top_left_x` and `top_left_y` variables that remember where the room should be drawn in the window.

The new code in [Listing 8-6](#) starts by working out where the middle of the window is. The `HEIGHT` and `WIDTH` variables store the window's size in pixels. Dividing them by 2 gives us the coordinates of the center of the window. We store these in the `center_y` ❶ and `center_x` ❷ variables.

The program then works out how wide the image of the room is in pixels ❸. That will be the width of the room in tiles multiplied by the size of a tile. The result is stored in `room_pixel_width`. A similar calculation is done for the room height ❹.

To put the room image in the middle of the room, we want half the room to be to the left of the center line and half to the right. So we subtract half the room width in pixels from the center line ❺ and start drawing the room there.

A similar calculation is used for `top_left_y` except we add 110 to the result ❻. We need to add 110 because our final screen layout will use an area at the top of the screen as an

information panel. We nudge the room image down a bit to make room for the panel.

## MAKING THE FRONT WALL FADE IN AND OUT


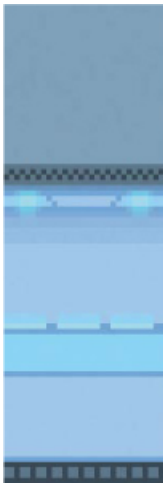
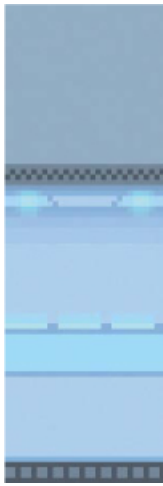


At this point, there are some dead spots in the game where the player can't be seen. In the middle of the room, we can avoid that by making sure objects are not so tall that they obstruct the player. We need a tall wall at the front of the room, though.

Blocking the player with a wall at the front of the room can cause all sorts of problems: if you drop something, you won't be able to find it, or if something is hurting you, you won't be able to see it! The solution is to make the wall fade away when the player approaches it.

The `draw()` function already draws the front wall pillars using animation frames. The wall animation has five frames (numbered from 0 to 4) in the `PILLARS` list. The first frame is the solid wall, and the last frame shows the wall at its most translucent (see [Table 8-2](#)). As the animation frame number increases, the wall becomes more transparent. The current frame is stored in the variable `wall_transparency_frame`.

Because of the way the transparency works in the images, when the transparent wall is drawn on top of the player, the player can be seen through it.

**Table 8-2:** The Animation Frames for the Front Wall

| Frame number | 0                                                                                   | 1                                                                                   | 2                                                                                    | 3                                                                                     | 4                                                                                     |
|--------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Image        |  |  |  |  |  |

Listing 8-7 shows the new function called `adjust_wall_transparency()`, which will fade the wall in and out. Add it at the end of the `DISPLAY` section, after the `draw()` function you just completed, and before the `START` section. You also need to add a line at the end of the program, outside the function, which will schedule it to run regularly. This line is also in Listing 8-7.

Save your updated program as *listing8-7.py* and run it using `pgzrun listing8-7.py`. If you walk behind the front wall, it now fades to transparent so you can be seen through it (see Figure 8-3 earlier in this chapter). When you walk away again, the wall changes back to being solid again.

### *listing8-7.py*

---

```
--snip--
```

```

DISPLAY ##
#####
```

```
--snip--
```

```
screen.surface.set_clip(None)
```

```
def adjust_wall_transparency():
 global wall_transparency_frame
```

- ❶ `if (player_y == room_height - 2`
- ❷  `and room_map[room_height - 1][player_x] == 1`
- ❸  `and wall_transparency_frame < 4):`
- ❹  `wall_transparency_frame += 1 # Fade wall out.`
  
- ❺ `if ((player_y < room_height - 2`
- ❻  `or room_map[room_height - 1][player_x] != 1)`
- ❼  `and wall_transparency_frame > 0):`
- ❽  `wall_transparency_frame -= 1 # Fade wall in.`

```

START
```



```
#####
```

```
clock.schedule_interval(game_loop, 0.03)
generate_map()
❸ clock.schedule_interval(adjust_wall_transparency, 0.05)
```

---

### *Listing 8-7: Making the front wall see-through when you approach it*

The final line we added in [Listing 8-7](#) makes the function `adjust_wall_transparency()` run once every 0.05 seconds ❸. This makes the wall fade in or out as necessary as the player walks around the room.

Let's see how this new function works. If the player is standing behind the wall, the following two statements are true:

- Their *y* position will be equal to `room_height - 2` ❶. As [Figure 8-6](#) shows, the bottom row of the map is `room_height - 1`. So we check whether the player is in the row above that.
- There is a piece of wall in the bottom row of the room that is in line with the player's *x* position ❷. In [Figure 8-6](#), the red square marks a position where we can't see the player. The bottom row in front of them contains a 1 for the wall. The green square shows where we can see the player, because they're in the doorway. Here, the bottom row of the room map contains a 0.



|  |  |  |  |   |   |                                                                                   |   |                                                                                   |   |   |   |   |   |  |
|--|--|--|--|---|---|-----------------------------------------------------------------------------------|---|-----------------------------------------------------------------------------------|---|---|---|---|---|--|
|  |  |  |  | 0 | 1 | 2                                                                                 | 3 | 4                                                                                 | 5 | 6 | 7 | 8 |   |  |
|  |  |  |  | 1 | 1 | 1                                                                                 |   |                                                                                   |   | 1 | 1 | 1 | 0 |  |
|  |  |  |  | 1 |   |                                                                                   |   |                                                                                   |   |   |   | 1 | 1 |  |
|  |  |  |  | 1 |   |                                                                                   |   |                                                                                   |   |   |   | 1 | 2 |  |
|  |  |  |  |   |   |                                                                                   |   |                                                                                   |   |   |   |   | 3 |  |
|  |  |  |  |   |   |                                                                                   |   |                                                                                   |   |   |   |   | 4 |  |
|  |  |  |  |   |   |                                                                                   |   |                                                                                   |   |   |   |   | 5 |  |
|  |  |  |  | 1 |   |                                                                                   |   |                                                                                   |   |   |   | 1 | 6 |  |
|  |  |  |  | 1 |   |  |   |  |   |   |   | 1 | 7 |  |
|  |  |  |  | 1 | 1 | 1                                                                                 | 0 | 0                                                                                 | 0 | 1 | 1 | 1 | 8 |  |

Figure 8-6: Working out whether the player is behind the wall

If the player is behind the wall ❶ ❷ and the wall transparency is not set to maximum ❸, the wall transparency is increased by 1 ❹.

If either of the following is true, it means the player *isn't* hidden by the wall:

- Their *y* position is less than `room_height - 2` ❺. The player can be seen, at least in part, if they're farther back in the room.
- There is not a piece of wall in the bottom row of the room in line with their *x* position ❻.

In these cases, if the wall transparency is set to more than the minimum ❼, it's reduced by one ❽.

The `draw()` function uses the value of `wall_transparency_frame` to work out which image from the animation frames in the `PILLARS` list to use in the front row.

The effect is that the wall gradually fades in and out, depending on whether the player is behind it or not. This fading happens fast enough that players won't be delayed by it

but not so fast that it vanishes instantly, which would be distracting.

## DISPLAYING HINTS, TIPS, AND WARNINGS

There are times when the *Escape* game uses text to tell you what's going on. For example, it might use text to tell you what happens when you do something with an object, or provide a description of it.

The final function in the `DISPLAY` section of the program writes messages at the top of the game window. There are two lines of text:

- The first line, positioned at 15 pixels from the top of the window, tells players about what they're doing. For example, it displays object descriptions and tells them what happens when they use the objects.
- The second line, positioned at 50 pixels from the top of the window, is for important messages.

The lines of text are separated like this so important messages don't get covered up by less important messages. If the game needs to tell you about a life-threatening situation, you don't want that message to be replaced with one that tells you about the new room you've entered!

Add the new code in [Listing 8-8](#) to the end of the `DISPLAY` section, after where you added the wall transparency code in [Listing 8-7](#). Save the listing as *listing8-8.py*. You can test it by running it with `pgzrun listing8-8.py`, but you won't see any difference yet. In a moment, we'll add some instructions to use this new `show_text()` function.

*listing8-8.py*

---

--snip--

```
if ((player_y < room_height - 2
 or room_map[room_height - 1][player_x] != 1)
 and wall_transparency_frame > 0):
 wall_transparency_frame -= 1 # Fade wall in.
```

```
❶ def show_text(text_to_show, line_number):
 if game_over:
 return
```

```

❷ text_lines = [15, 50]
❸ box = Rect((0, text_lines[line_number]), (800, 35))
❹ screen.draw.filled_rect(box, BLACK)
❺ screen.draw.text(text_to_show,
 (20, text_lines[line_number]), color=GREEN)

```

```

#####

START

#####

```

```
--snip--
```

---

### *Listing 8-8: Adding the text display function*

We'll use the `show_text()` ❶ function like this (don't type this in):

---

```
show_text("message", line_number)
```

---

The line number will be either 0 for the top row or 1 for the second row, which is reserved for important messages. At the start of the function, the message is put into the variable `text_to_show` and the row number goes into `line_number` ❶.

We use a list called `text_lines` to remember the vertical positions (in pixels) of the two lines of text ❷. We also define a box ❸ and fill it with black ❹, to clear the row of text before the new message is drawn.

Finally, we use the `screen.draw.text()` function in Pygame Zero to put the text on the screen ❺. This function takes the text, the text's *x* and *y* position, and the text color. The position numbers go inside parentheses (they make up a tuple).

In [Listing 8-8](#) ❺, the *x* position is 20 pixels from the left, and the vertical position is taken from the `text_lines` list, using the number in `line_number` as the list index.

## SHOWING THE ROOM NAME WHEN YOU ENTER THE ROOM

To test the `show_text()` function, let's add the `start_room()` function, which displays the name of the room when you walk into it. Put this function in the `GAME LOOP` section before the `game_loop()` function, as shown in [Listing 8-9](#). Save your program as *listing8-9.py*. When you run it, you won't see anything new yet.

```
--snip--

#####
GAME LOOP
#####

def start_room():
 show_text("You are here: " + room_name, o)

def game_loop():
--snip--
```

---

### *Listing 8-9: Adding the `start_room()` function*

This function uses the `room_name` variable, which we set up in the `generate_map()` function. It contains the name of the current room, taken from the `GAME_MAP` list. The room name is combined with the text "You are here: " and is sent to the `show_text()` function.

Now we need to set our new `start_room()` function to run whenever the player enters a new room. We included the code to do this in [Listing 7-6 in Chapter 7](#), but we commented it out. Now we're ready for it! Anywhere we have the code `#start_room()` we want to replace it with `start_room()`. That `#` is working as an "off switch," telling Python to ignore the instruction. To turn the instruction on, we remove the `#` sign.

Rather than manually finding all the lines that need to change, we'll get IDLE to do it for us. Follow these steps, and refer to [Figure 8-7](#):

1. Click **Edit ▸ Replace** (or press CTRL-H) in IDLE to show the replace text dialog box.
2. Enter `#start_room()` into the Find box.
3. Enter `start_room()` into the Replace With box.
4. Click **Replace All**.

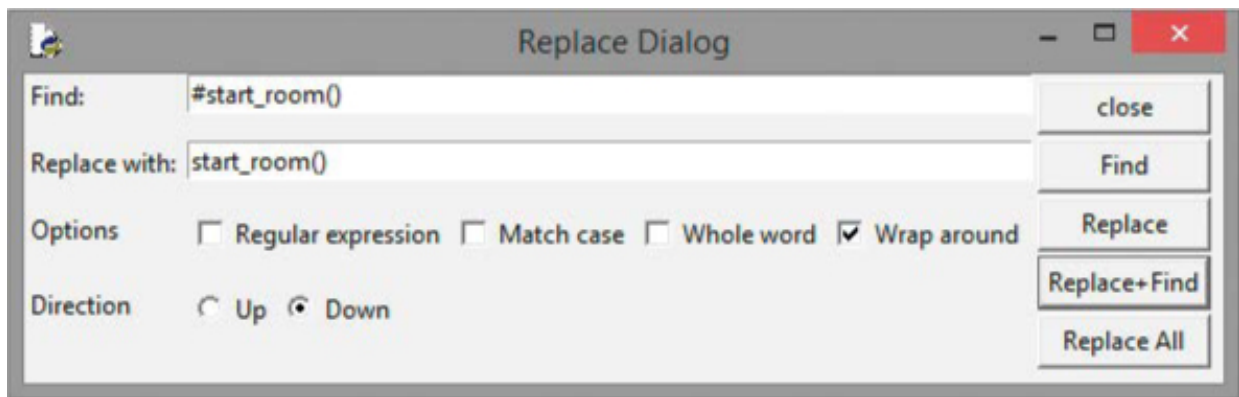


Figure 8-7: Enabling the `start_room()` function when the player enters a new room

IDLE should replace the instruction in four places and will jump to the last one in the listing, as shown in Listing 8-10 (there's no need to type this listing in).

Save the listing as *listing8-10.py* and run the program using `pgzrun listing8-10.py`. A message should appear announcing each new room as you enter it. It's triggered by walking through the door, so it doesn't work in the first room.

#### *listing8-10.py*

---

```
--snip--
if player_y == -1: # through door at TOP
 #clock.unschedule(hazard_move)
 current_room -= MAP_WIDTH
 generate_map()
 player_y = room_height - 1 # enter at bottom
 player_x = int(room_width / 2) # enter at door
 player_frame = 0
 start_room()
 return
--snip--
```

---

Listing 8-10: Enabling the `start_room()` function when the player leaves the room

This completes the `DISPLAY` section of the *Escape* game! We'll make a few small changes later to show enemies, but otherwise we've laid the foundation for the rest of the game.

In the next chapter, we'll start unpacking your personal effects as we add props to the game.

## ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ A piece of information sent to a function is called an *argument*.
- ☐ To send information to a function, you put it between the parentheses after the function name. You can send several arguments if you separate them with commas. For example: `add(5, 7)`.
- ☐ To enable a function to accept information, you set up local variables to receive the arguments when you define the function.
- ☐ The `DISPLAY` section of the program draws the room, animates the transparent wall, and displays text messages.
- ☐ The `show_text()` function takes two arguments: the string you want to display and the row number (0 or 1). Row 1 is reserved for important messages.
- ☐ You define a `Rect` by giving Python tuples for its position and size.
- ☐ The `screen.draw.filled_rect()` function draws a filled rectangle.
- ☐ The colors in Pygame Zero use the RGB (red, green, blue) format. For example `(255, 100, 0)` is orange: maximum red, a dash of green, and no blue.
- ☐ If you want to replace some code throughout the whole program, you can use the Replace All option in IDLE.



# MISSION DEBRIEF

Here is the answer for the training mission in this chapter.

## TRAINING MISSION #1

As well as changing the calculation, remember to change the name of the function, and change the plus sign to a minus sign inside the `print()` function. The program would still work if you didn't do this, but it would be confusing to use an `add()` function to subtract.

If you change the order that you send the numbers in, the function still subtracts the second number from the first number, but you get a different result. This is why it's important to make sure you send information to a function in the same order it expects to receive it. With some functions, if you send arguments in the wrong order, you'll trigger a Python error.

---

```
def subtract(first_number, second_number):
 total = first_number - second_number
 print(first_number, "-", second_number, "=", total)

subtract(5, 7)
subtract(2012, 137)
subtract(1234, 4321)
```

---

# 9

## UNPACKING YOUR PERSONAL ITEMS



Now that the space station is operational, it's time to unpack your personal items and the various tools and pieces of equipment you'll need as you carry out your work.

In this chapter, you'll build the code for objects that can move between rooms (*props*). When you play the game, you'll be able to discover new items, pick them up, move them around, and use them to solve puzzles.

### ADDING THE PROPS INFORMATION

You've already added some information about props in [Chapter 5](#) when you added the image filenames and descriptions to the `objects` dictionary. The `objects` dictionary contains information about *what* an item is. In this chapter, we'll add information to tell the game *where* the props go.

You might be wondering why we're handling the props separately from the scenery. We do this because their information is used in different ways: the `scenery` dictionary stores information using the room as the key. This makes sense, because the program needs to get information about all the scenery in a room at the same time. After the scenery information is added to the room map, the `scenery` dictionary is not needed again until the player enters a new room.

By contrast, props move around, so the information for a prop might be needed at any time in any room. If that information is buried in a list of scenery items, it's harder to find and change.

We'll create a new dictionary called `props` to store information about props. We'll use the object number as a key, and each entry will be a list that contains the following:

- The number of the room the prop is in
- The *y* position of the prop in the room (in tiles)
- The *x* position of the prop in the room (in tiles)

For example, here's the entry for the hammer, which is object 65:

---

65: [50, 1, 7]

---

It's in room 50, at *y* position 1 and *x* position 7.

Objects that are not in the game world or that are being carried by the player will have a room number of 0, which is not a real location in the game. Some objects aren't in the game world until they've been created or after they've been destroyed, for example. These would be stored in room 0.

#### TIP

The `props` and `objects` dictionaries use the same keys. If you want to know what item 65 is in the `props` dictionary, read its details in the `objects` dictionary.

Listing 9-1 shows the code for adding the props information to the game. Open *listing8-10.py*, your final program from the previous chapter. Add the new `PROPS` section after the `show_text()` function in the `DISPLAY` section and before the `START` section. Only add the new lines, and save the new program as *listing9-1.py*. If you want to avoid typing the data, you can copy and paste it from the *data-chapter9.py* file.

You can run the program with `pgzrun listing9-1.py`. It won't do anything new yet, but you can check whether there are any error messages in the command line window.

---

```

--snip--
 screen.draw.text(text_to_show,
 (20, text_lines[line_number]), color=GREEN)

#####

PROPS

#####

Props are objects that may move between rooms, appear or disappear.
All props must be set up here. Props not yet in the game go into room 0.
object number : [room, y, x]
❶ props = {
 20: [31, 0, 4], 21: [26, 0, 1], 22: [41, 0, 2], 23: [39, 0, 5],
 24: [45, 0, 2],
❷ 25: [32, 0, 2], 26: [27, 12, 5], # two sides of same door
 40: [0, 8, 6], 53: [45, 1, 5], 54: [0, 0, 0], 55: [0, 0, 0],
 56: [0, 0, 0], 57: [35, 4, 6], 58: [0, 0, 0], 59: [31, 1, 7],
 60: [0, 0, 0], 61: [36, 1, 1], 62: [36, 1, 6], 63: [0, 0, 0],
 64: [27, 8, 3], 65: [50, 1, 7], 66: [39, 5, 6], 67: [46, 1, 1],
 68: [0, 0, 0], 69: [30, 3, 3], 70: [47, 1, 3],
❸ 71: [0, LANDER_Y, LANDER_X], 72: [0, 0, 0], 73: [27, 4, 6],
 74: [28, 1, 11], 75: [0, 0, 0], 76: [41, 3, 5], 77: [0, 0, 0],
 78: [35, 9, 11], 79: [26, 3, 2], 80: [41, 7, 5], 81: [29, 1, 1]
}

checksum = 0
for key, prop in props.items():
❹ if key != 71: # 71 is skipped because it's different each game.
 checksum += (prop[0] * key
 + prop[1] * (key + 1)
 + prop[2] * (key + 2))
❺ print(len(props), "props")
 assert len(props) == 37, "Expected 37 prop items"
 print("Prop checksum:", checksum)
❻ assert checksum == 61414, "Error in props data"

❼ in_my_pockets = [55]

```

```
selected_item = 0 # the first item
item_carrying = in_my_pockets[selected_item]
```

```
#####
START
#####
--snip--
```

---

### *Listing 9-1: Adding the props information to Escape*

We start the new `PROPS` section by creating the dictionary to store the information about the props ❶. This dictionary lists the position locations for all the props, starting with some doors (20 to 24) and including a rescue ship (40) and the carryable items starting at 53.

There is just one oddity to draw your attention to. We count doors as props rather than scenery, because they're not always there: when they're open, they're removed from the room. Most doors stay open when they're opened until the game ends. However, the door that connects rooms 27 and 32 can also shut, meaning players can see it from both sides. As a result, we need two props to represent this door ❷, showing it in the top of room 27 and the bottom of room 32. These two doors are object numbers 25 and 26.

Prop 71 is the Poodle lander, which crash-landed on the planet surface before the game began. We use the `LANDER_Y` and `LANDER_X` variables from the `VARIABLES` part of the program ❸ to position the lander, because its location will change with each new game. The Poodle landed with such force that it might have become covered with Martian soil. It lives in room 0 until the player can dig it up.

As with the scenery information (see [Chapter 6](#)), I've used a checksum here to help you spot whether you made an error entering the data. It might not be possible to play the game all the way to the end if a mistake is made here. The only prop missing from the checksum calculation is number 71, because its position uses different random numbers in each game ❹.

If you want to change the props data, the easiest thing to do is to comment out the two checksum instructions ❺ like this to turn them off:

---

```
#assert len(props) == 37, "Expected 37 prop items"
#assert checksum == 61414, "Error in props data"
```

---

The program shows the checksum total and number of data items in the command line window ❹, so if you change the props data, you can use this information to update the numbers in the two `assert` instructions so they are correct for your customized data. If you do this, you can continue using these lines rather than commenting them out.

The program also sets up two new variables and a list we'll need later in the chapter. The `in_my_pockets` ❺ list stores all the items the player has picked up, also known as their *inventory*. One of these items is always selected, so the player is ready to do something with it. The `selected_item` variable stores its index number in the `in_my_pockets` list. The `item_carrying` variable stores the object number of the item the player has selected. You can think of the `item_carrying` variable as being the number of the object in their hands. I'll tell you more about these variables later in this chapter.

## ADDING PROPS TO THE ROOM MAP

We've added the information about where the props are located, so now let's display the props. We'll make it so that when props are located in the current room, they're put into the `room_map` list as the player enters the room. Then the `draw()` function uses that list to draw the room.

We'll place the instructions to add the props to the room map into the `MAKE MAP` part of the program, inside the `generate_map()` function. We'll simply add these instructions after the instructions you added in [Chapter 8](#) for working out the `top_left_x` and `top_left_y` variables, just above the start of the `GAME LOOP` section.

Because the new instructions are all part of the `generate_map()` function, you need to indent them by at least four spaces.

Add the new instructions shown in [Listing 9-2](#) to your program, and save it as *listing9-2.py*. Run the program with `pgzrun listing9-2.py`. You should see that new objects have appeared in some of the rooms, as shown in [Figure 9-1](#).



Figure 9-1: *That door wasn't there a minute ago! That air canister might come in handy, though.*

*listing9-2.py*

---

--snip--

```
top_left_x = center_x - 0.5 * room_pixel_width
top_left_y = (center_y - 0.5 * room_pixel_height) + 110
```

```
❶ for prop_number, prop_info in props.items():
❷ prop_room = prop_info[0]
 prop_y = prop_info[1]
 prop_x = prop_info[2]
❸ if (prop_room == current_room and
❹ room_map[prop_y][prop_x] in [0, 39, 2]):
❺ room_map[prop_y][prop_x] = prop_number
❻ image_here = objects[prop_number][0]
 image_width = image_here.get_width()
 image_width_in_tiles = int(image_width / TILE_SIZE)
❷ for tile_number in range(1, image_width_in_tiles):
 room_map[prop_y][prop_x + tile_number] = 255
```

```
#####
```

```
GAME LOOP
```

```
#####
```

--snip--

---



---

### *Listing 9-2: Adding the props to the room map for the current room*

In the new code, we start by setting up a loop to go through the items in the `props` dictionary ❶. For each item, the dictionary key goes into the variable `prop_number`, and the list with the position information goes into the list `prop_info`.

To make the program easier to read, I've set up some variables to store the information from the `prop_info` list ❷. The program extracts the information for the room number (and puts it into `prop_room`) and the *y* and *x* positions (which go into the `prop_y` and `prop_x` variables).

We add a check to see whether the `prop_room` matches the room the player is in ❸ and whether the prop is sitting on the floor ❹. The floor check puts the three different floor types in a list (0 for inside, 2 for soil, and 39 for the pressure pad in room 26). The program checks the prop's position to see what's in that location in the room map. If it's one of these floor types, it means the object is sitting on the floor in full view. If not, the prop is hidden inside an item of scenery and shouldn't be visible yet. For example, if a cabinet is in the prop's location instead of the floor, the prop won't be shown onscreen. The player can still find the prop by examining the cabinet at that location, though.

If the prop is in the room and on the floor, the room map is updated with the prop number ❺.

Some props, like doors, are wider than one tile. So we add the number 255 to any tiles that the prop covers other than the first one ❻. This is similar to the code we used to mark wide scenery earlier in the `generate_map()` function (see [Listing 6-4 on page 106](#)).

## GETTING INFORMATION FROM A FUNCTION: ROLLING DICE

In [Chapter 8](#), you learned how to send information (or *arguments*) to a function. Let's look more closely at how to get information *back* from a function. We'll use this skill to create a function that tells us what object the player is standing on.

[Listing 9-3](#) shows a simple program that sends a number back from a function and puts it into a variable. This isn't part of the *Escape* game, so create a new file by clicking **File** ▶ **New** first.

Save the program as *listing9-3.py*. This program doesn't use Pygame Zero, so you can run it using **Run ▶ Run Module** in the script window. The program simulates a 10-sided die.

*listing9-3.py*

---

```
❶ import random

❷ def get_number():
❸ die_number = random.randint(1, 10)
❹ return die_number

❺ random_number = get_number()
❻ print(random_number)
```

---

*Listing 9-3: A 10-sided die simulator shows how to send a number back from a function.*

This program starts by telling Python to use the `random` module ❶, which gives Python new functions for making random choices. We then create a new function called `get_number()` ❷, which generates a random number between 1 and 10 ❸ and puts the result into a variable called `die_number`.

Normally, when you start a function (known as *calling* a function in Python jargon), you use its name, like this:

---

```
get_number()
```

---

This time, we not only start the function, but tell Python to put the result from the function into a variable called `random_number` ❺. When the function sends its result back using the `return` command ❹, the result goes into the `random_number` variable. The main part of the program can then print out its value ❻.

This code shows that the way to get information from a function is to set up a variable to store the information when the function is started ❺ and to use the `return` instruction to send that information back when the function finishes ❹. You can send strings and lists back too, not just numbers. Where possible, this is the best way to enable other parts of the program to use information from a function. This technique enables the main part of the program to get

information from a function's local variable (in this case `dice_number`), which would usually only be visible inside that function.

You won't need this program again, so you can close it when you've finished experimenting with it.

## FINDING AN OBJECT NUMBER FROM THE ROOM MAP

Shortly, we'll add the code to enable you to pick up objects in the space station. First we need a way to find out which object is being picked up.

When the player interacts with scenery or props, we need to find the number of the object they're using. Normally, this is simple. If the room map shows that the object number of the prop at the player's location is 65, that's a hammer. The program can show a description of the hammer, and let the player pick it up or use it.

Identifying the object number gets tricky with wide objects that span multiple tiles. We use the number 255 to mark tiles covered by a wide object, but that number doesn't correspond to a prop. The program needs to work out what the real object number is by moving left in the room map until it finds a number that isn't 255.

For example, if the player examines the rightmost third of a door, the program would see that this position contains 255, so it would check the position to the left. That position also contains 255, so the program would check farther left. If that tile contains a number other than 255, the program knows it's found the real object number, which might be 20 (one of the doors), for example. Using the object number 20, the program can then let the player examine or open the door.

We'll create two functions that will work out the object number, shown in [Listing 9-4](#). You need to add these to [Listing 9-2](#), so click **File** ▶ **Open** to open *listing9-2.py* again if necessary. We'll start a new section of the program called `PROP_INTERACTIONS`. Put this after the `PROPS` section. This new section will be where we put the code for picking up and dropping props.

Save the updated program as *listing9-4.py*. It won't do anything new yet, but you can run it using `pgzrun listing9-4.py` to check that you haven't added any mistakes. Look in the command line window for any error messages.

[listing9-4.py](#)

---

```

--snip--
in_my_pockets = [55]
selected_item = 0 # the first item
item_carrying = in_my_pockets[selected_item]

#####

PROP INTERACTIONS

#####

❶ def find_object_start_x():
❷ checker_x = player_x
❸ while room_map[player_y][checker_x] == 255:
❹ checker_x -= 1
❺ return checker_x

❻ def get_item_under_player():
❼ item_x = find_object_start_x()
❽ item_player_is_on = room_map[player_y][item_x]
❾ return item_player_is_on

--snip--

```

---

#### *Listing 9-4: Finding the real object number*

Before we get into how this code works, I'll explain how the game loop lets players interact with props and scenery:

1. When the player presses a movement key, the program changes the player's position (even if that puts them somewhere impossible, like inside a wall).
2. The program carries out any actions the player requires using the object at the player's location. This means the player and the object are in the same position in the room at this time.
3. If the player is standing somewhere they're not allowed to be (such as inside a wall), the program moves them back to where they were.

The entire process happens so fast you never see the player go inside the wall or other piece of scenery. This way, the player can use a movement key plus an action key to

examine or use the scenery. For example, you can walk into a wall and press the spacebar to examine the wall and see a description of it. This process also works with an object the player is standing on, such as a prop on the floor.

The first new function we added in [Listing 9-4](#) is `find_object_start_x()` ❶. This function finds the start position of whatever object is at the player's position, going left to find the real object number if the location contains 255.

To do this, the function sets the variable `checker_x` to be the same as the player's *x* position ❷. We use a loop that keeps going for as long as the room map contains 255 at the *x* position of `checker_x` and at the player's *y* position ❸. Inside that loop is a single instruction to reduce `checker_x` by 1 ❹, moving 1 tile to the left. When the loop finishes, `checker_x` contains the left position where the object begins. That number is then sent back ❺ to the instruction that started the function.

The second new function is `get_item_under_player()` ❻, which works out which object is at the player's position. It uses the first function to find out where the object starts and stores the *x* position in the variable `item_x` ❼. Then it looks at the room map data for that position to see what object is there ❽ and sends that number back to the instruction that started the function ❾.

## PICKING UP OBJECTS

Now that these functions are in place, we can create a couple of functions for picking up objects and then storing them in a player's inventory. Then we'll add some keyboard controls.

## PICKING UP PROPS

Add the two functions shown in [Listing 9-5](#) to the end of the `PROP INTERACTIONS` section of the program, just after where you added the code in [Listing 9-4](#).

Save this program as *listing9-5.py*. You can check for any errors by running it using `pgzrun listing9-5.py`, but you won't see any difference yet. This code adds some new functions but doesn't include any key controls to enable the player to use them.

*listing9-5.py*

---

--snip--

```
item_player_is_on = room_map[player_y][item_x]
return item_player_is_on
```

```

def pick_up_object():
 global room_map
 ❶ item_player_is_on = get_item_under_player()
 ❷ if item_player_is_on in items_player_may_carry:
 ❸ room_map[player_y][player_x] = get_floor_type()
 ❹ add_object(item_player_is_on)
 show_text("Now carrying " + objects[item_player_is_on][3], 0)
 sounds.pickup.play()
 time.sleep(0.5)
 ❺ else:
 show_text("You can't carry that!", 0)

 ❻ def add_object(item): # Adds item to inventory.
 global selected_item, item_carrying
 ❼ in_my_pockets.append(item)
 ❽ item_carrying = item
 ❾ selected_item = len(in_my_pockets) - 1
 display_inventory()
 ❿ props[item][0] = 0 # Carried objects go into room 0 (off the map).

def display_inventory():
 print(in_my_pockets)
--snip--

```

---

### *Listing 9-5: Adding the functions to pick up objects*

The function `pick_up_object()` will start when the player presses the *get* key (G) to pick up an item. It begins by putting the object number for the item at the player's position into the variable `item_player_is_on` ❶. If the item is carryable ❷, the rest of the function picks it up.

To remove the item from the floor, the program replaces the room map at the player's position with the object number for the floor (either soil or floor tiles) ❸. The `get_floor_type()` function is used to find out what the floor type should be in this room. When the room is redrawn, the item will disappear from the floor, so it looks like it's been picked up. The item is then added to the list of items the player is carrying, using the `add_object()` function ❹.

We then show a message onscreen telling the player they picked up an item and play a sound effect. We add a short delay of half a second using the `time.sleep(0.5)` instruction to make sure the confirmation message isn't overwritten if the player holds down the key too long.

If the item isn't carryable, we show a message telling them they can't carry it ❸. For example, scenery can't be carried, so we need to tell players that. Otherwise, they might just think they're pressing the wrong key or the program isn't working.

The `add_object()` function adds an item to the `in_my_pockets` list, which stores the items the player is carrying (their inventory). At the start of the function, the object number this function receives is put into the local variable `item` ❹. The item is added to the end of the `in_my_pockets_list` using `append()` ❺.

We use the global variable `item_carrying` to store the object number of whatever's in the player's hands, so it is set to be the object number of this item ❻. We set the `selected_item` variable as the last item in the list, meaning the item the player just picked up is selected ❼. These variables will be important when objects are used later on, and when the `display_inventory()` function shows the list of items on the screen. For now, that function just prints out the list in the command line window.

Finally, we set the item's position in the `props` dictionary to be room 0 ❽. This means the item just picked up is not shown in the game map anywhere. If we didn't do this, the item would reappear in the room again when the player next entered it.

## ADDING THE KEYBOARD CONTROLS

To enable the new functions to work their magic, we need to add the keyboard control too. We'll use the G key as our get key.

Place the new instructions, shown in [Listing 9-6](#), in the `game_loop()` function in the `GAME LOOP` section of the program. The new instruction belongs after the exit checks have been made and before the player is moved back if they're standing somewhere they shouldn't be.

*listing9-6.py*

---

```
--snip--
 player_frame = 0
 start_room()
```



```
return
```

- ❶ `if keyboard.g:`
- ❷ `pick_up_object()`

```
If the player is standing somewhere they shouldn't, move them back.
if room_map[player_y][player_x] not in items_player_may_stand_on: #\
or hazard_map[player_y][player_x] != 0:
--snip--
```

---

### *Listing 9-6: Adding the keyboard control*

You need to indent the first new instruction by four spaces ❶, because it's inside the `game_loop()` function. Indent the second one ❷ by four more spaces, because it belongs to the `if` instruction above. These instructions run the `pick_up_object()` function ❷ when the player presses the G key ❶.

Save the listing as *listing9-6.py*. When you run `pgzrun listing9-6.py`, you should be able to pick up objects.

Test it starting with the air canister in the first room. Just walk onto it and press G. You'll hear a sound and see a message, and the object will disappear from the room.

The command line window (where you entered the `pgzrun` instruction) will also show the inventory list every time you pick up an object, like this:

---

```
[55, 59]
```

---

Each time, you'll see a new item added to the end of the list. Item 55, the yoyo, is in your pocket at the start of the game.

## ADDING THE INVENTORY FUNCTIONALITY

Now you can pick up props that you find around the space station. We should add an easy way to see what you're carrying and to choose different items to use. We'll make a new `display_inventory()` function that displays a strip at the top of the game window showing the items the player is carrying.

We'll then add controls so the player can press the TAB key to select the next item in the list. The selected item has a box drawn around it, and its description is shown

underneath. Figure 9-2 shows you what it will look like.

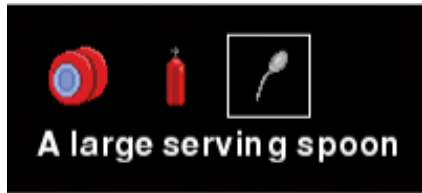


Figure 9-2: The inventory at the top of the game window

## DISPLAYING THE INVENTORY

Listing 9-7 shows you the code to add. Listing 9-5 included some code for the `display_inventory()` function. Replace that with the new code. Save this listing as *listing9-7.py*. When you run the program using `pgzrun listing9-7.py`, you'll be able to see items added to your inventory at the top of the screen as you collect them.

*listing9-7.py*

---

--snip--

```
selected_item = len(in_my_pockets) - 1
display_inventory()
props[item][0] = 0 # Carried objects go into room 0 (off the map).
```

```
def display_inventory():
```

- ❶ `box = Rect((0, 45), (800, 105))`  
`screen.draw.filled_rect(box, BLACK)`
  - ❷ `if len(in_my_pockets) == 0:`  
 `return`
  - ❸ `start_display = (selected_item // 16) * 16`
  - ❹ `list_to_show = in_my_pockets[start_display : start_display + 16]`
  - ❺ `selected_marker = selected_item % 16`
  - ❻ `for item_counter in range(len(list_to_show)):`  
 `item_number = list_to_show[item_counter]`  
 `image = objects[item_number][0]`
  - ❼ `screen.blit(image, (25 + (46 * item_counter), 90))`
- 
- ```
box_left = (selected_marker * 46) - 3
```

```

❸ box = Rect((22 + box_left, 85), (40, 40))
   screen.draw.rect(box, WHITE)
   item_highlighted = in_my_pockets[selected_item]
   description = objects[item_highlighted][2]
❹ screen.draw.text(description, (20, 130), color="white")

```

```
#####
```

```
## START ##
```

```
#####
```

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

```
generate_map()
```

```
clock.schedule_interval(adjust_wall_transparency, 0.05)
```

```
❺ clock.schedule_unique(display_inventory, 1)
```

Listing 9-7: Displaying the inventory

The new `display_inventory()` function starts by drawing a black box over the inventory area to clear it ❶. If the player isn't carrying anything, the function returns without taking any further action because there are no items to display ❷.

There is only room to show 16 items on the screen, but the player could carry many more items than that. If the `in_my_pockets` list is too long to fit on the screen, the program shows it 16 items at a time. The player can select any of the items shown on the screen by pressing the TAB key to move through them, from left to right. If the last item displayed is selected and they press TAB, the next chunk of the list is shown. If the player presses TAB on the final item in the list, the start of the list appears again.

We store the part of the `in_my_pockets` list currently displayed on the screen in another list called `list_to_show` and use a loop to display it ❸. The loop puts numbers into a variable called `item_counter`, which is used to extract the right image to draw each time, and also work out where to draw it ❹.

The clever bit is working out which items should go into `list_to_show`. In the `start_display` variable, we store the index number for the first item in `in_my_pockets` that the program should draw ❺. The `//` operator divides the selected item number by 16, rounding down. The result is then multiplied by 16 to get the index number for the first item in the batch. For example, if the selected item is number 9, you'd divide 9 by 16 (0.5625), round down (0), and multiply by 16 (still 0), getting a result of 0. That's the start of the list, which makes sense, because we know there's room for 16 items onscreen and that 9

is less than 16. If you wanted to see the group of items that includes item 22, you'd divide 22 by 16 (1.375), round down (1), and multiply by 16, getting a result of 16. That's the start of the next batch, because the first batch has index numbers that range from 0 to 15.

We create the `list_to_show` list using a technique called *list slicing*, which is simply using just a part of a list. When you give Python two list indexes with a colon between them, the program will cut out that part of the list. The section we're using starts at the `start_display` index and finishes 15 items later ❹. A list slice leaves out the last item, so we use `start_display + 16` as the end point.

We also need another calculation to work out which item to highlight as the selected item from the new list ❺. The item will have an index between 0 and 15, and we'll store it in `selected_marker`. We calculate it as the remainder after we divide the selected item number by 16. For example, if the selected item is number 18, it will be at index number 2 when the second group of items is displayed. (The first item is at index 0, remember.) Python has the modulo operator `%`, which you can use to get the remainder after a division.

To highlight the selected item on the screen, we draw a box around it using a `Rect` positioned at its left edge ❽. Unlike the filled rectangles you've seen (for example ❶), this instruction draws a hollow box with a white edge.

The description for the selected item is displayed underneath the inventory ❾, so players can TAB through their items to read their descriptions again.

Finally, when the program first runs, it needs to display the inventory. This is scheduled with a slight delay ❿ to avoid any problems that are caused by trying to use a `screen.blit()` instruction before Pygame Zero has finished starting up. While `clock.schedule_interval()` is used to run a function regularly, `clock.schedule_unique()` is used to run a function just once, after a delay.

ADDING THE TAB KEYBOARD CONTROL

When you run the program, you can see the inventory, but you have no way to cycle between items yet, so the latest item you collected is always selected. Let's add the keyboard control that enables you to TAB through the inventory to select different items.

Place the new instructions in [Listing 9-8](#) into the `game_loop()` function, just after where you added the keyboard control to get items in [Listing 9-6](#). You need to indent them by

at least four spaces because they're inside the `game_loop()` function.

Save this listing as *listing9-8.py*. When you run the program using `pgzrun listing9-8.py`, you'll be able to press the TAB key to select different items in your inventory. (The TAB key is usually on the left side of the keyboard and might have a picture of two arrows on it.)

Pick up a few items before testing the new keyboard control, or skip ahead to the next section to fill up your inventory with more items to test with.

listing9-8.py

```
--snip--
    if keyboard.g:
        pick_up_object()

❶ if keyboard.tab and len(in_my_pockets) > 0:
❷     selected_item += 1
❸     if selected_item > len(in_my_pockets) - 1:
        selected_item = 0
❹     item_carrying = in_my_pockets[selected_item]
❺     display_inventory()

❻ if keyboard.d and item_carrying:
❷     drop_object(old_player_y, old_player_x)

❸ if keyboard.space:
❹     examine_object()
--snip--
```

Listing 9-8: Enabling the TAB key to select items in the inventory

The first chunk of instructions runs when the player presses the TAB key, but only if the `in_my_pockets` list contains some items (so its length is more than 0) ❶.

To select the next item in the inventory, we increase the `selected_item` variable by 1 ❷ when the TAB key is pressed. This variable stores an index number (which starts at 0), so the program subtracts 1 from the length of the list to see whether the `selected_item` is now past the end of the list ❸. If it is, the selected item is reset to be the first item again, at 0.

We set the variable `item_carrying` as the object number of the selected item (which is taken from the `in_my_pockets` list) ❹. For example, if the `in_my_pockets` list contained the object numbers 55 and 65, and the `selected_item` was 0, `item_carrying` would contain 55 (the first item from `in_my_pockets`). Finally, the inventory is displayed using the `display_inventory()` function you created earlier ❺.

While we're working with this part of the program, we've added the keyboard controls for dropping and examining items too. When the player presses the D key and the `item_carrying` variable is not `False`, the `drop_object()` function runs ❻. This function is sent the player's old *y* and *x* positions as the location for dropping the item ❼. Remember that the player's current location might be inside a wall because of where we are in the game loop. We know that their most recent position before any movement is a safe place to drop something.

We also added the instructions to start the `examine_object()` function ❽ when the spacebar is pressed ❸.

Don't press D or the spacebar in the game yet: pressing them will cause the program to crash because we haven't added the functions for them. We'll add them shortly.

TESTING THE INVENTORY

We want to test the program properly, but at the moment you don't have many items in your inventory. To save time, we'll tweak the code to give you a fuller inventory so you can test the display and the TAB control.

We'll fill the `in_my_pockets` list with items when the game begins. The quickest way to do this is to change the instruction that sets up that list in the `PROPS` section of the program, like this (but don't do this yet!):

```
in_my_pockets = items_player_may_carry
```

That would mean you start the game carrying all the items it's possible to carry. If you do that, it might spoil your enjoyment of the game, though. You'll be carrying some items you might prefer not to see until later in the game. It'll make some of the puzzle solutions obvious.

Instead, I recommend you create a test list like this:

```
in_my_pockets = [55, 59, 61, 64, 65, 66, 67] * 3
```

This line creates a list that contains that sequence of items three times. You'll end up with an inventory that contains three of each item (which is impossible in the real game), but it will enable you to test that the inventory works correctly when it contains more than 16 items.

When you've finished testing, change the code back again. Otherwise, you might get unexpected results when playing the game. Here's what that line should look like:

```
in_my_pockets = [55]
```

DROPPING OBJECTS

Being able to collect stuff strewn all over the space station is great fun, but sometimes you'll want to put it down, so you can either work with it or leave it somewhere. We'll need two new functions for dropping items that will work a bit like the opposites of the functions for picking up items.

The `drop_object()` function (the opposite of the `pick_up_object()` function) will let you drop an object on the floor where the player was most recently standing. You added the keyboard control to start this function in [Listing 9-8](#).

The `remove_object()` function is like the `add_object()` function in reverse: it takes items out of the inventory and updates it.

Add the new functions, shown in [Listing 9-9](#), to the end of the `PROP INTERACTIONS` part of the program. Save the new program as *listing9-9.py*.

When you run the program using `pgzrun listing9-9.py`, you'll be able to drop objects. That includes the yoyo you start the game carrying and any new objects you pick up as you explore the space station.

listing9-9.py

```
--snip--
```

```
description = objects[item_highlighted][2]
screen.draw.text(description, (20, 130), color="white")
```

- ```
❶ def drop_object(old_y, old_x):
 global room_map, props
 ❷ if room_map[old_y][old_x] in [0, 2, 39]: # places you can drop things
```



```

❸ props[item_carrying][0] = current_room
 props[item_carrying][1] = old_y
 props[item_carrying][2] = old_x
❹ room_map[old_y][old_x] = item_carrying
 show_text("You have dropped " + objects[item_carrying][3], 0)
 sounds.drop.play()
❺ remove_object(item_carrying)
 time.sleep(0.5)
❻ else: # This only happens if there is already a prop here
 show_text("You can't drop that there.", 0)
 time.sleep(0.5)

```

```

def remove_object(item): # Takes item out of inventory
 global selected_item, in_my_pockets, item_carrying
❷ in_my_pockets.remove(item)
❸ selected_item = selected_item - 1
❹ if selected_item < 0:
 selected_item = 0
❺ if len(in_my_pockets) == 0: # If they're not carrying anything
 item_carrying = False # Set item_carrying to False
 else: # Otherwise set it to the new selected item
 item_carrying = in_my_pockets[selected_item]
 display_inventory()

```

```

#####
START
#####
--snip--

```

---

### *Listing 9-9: Adding the functions for dropping objects*

The `drop_object()` function needs two pieces of information: the player's old *y* and *x* positions. If the player moved this time through the `game_loop()` function, this will be the position they were in before they tried to move. If not, these numbers will be the same position as where they currently are. We know this is a sensible place to drop an item that won't put the object inside a wall. The player's old position goes into the variables `old_y` and `old_x` within this function ❶.

The program checks whether the room map at the player's old position is a type of

floor. If so, it's okay to drop a prop here, so the drop instructions are used. If not ❹, the player sees a message telling them they can't drop objects there. This will happen, for example, if there is already a prop in that position.

If the player can drop the item, we need to update the `props` dictionary. The variable `item_carrying` contains the number of the object the player is carrying. Its entry in the `props` dictionary is a list. The first list item (index 0) is the room the prop is in, the second item (index 1) is its *y* position, and the third item is its *x* position (index 2). These values are set to be the current room and the player's old position ❺.

The room map for the current room also needs to be updated, so the room contains the dropped item ❻. The game will show a message and play a sound to tell the player that they've successfully dropped something and then the item is removed from the inventory using the `remove_object()` function ❼.

The `remove_object()` function takes an item from the player's inventory and updates the `selected_item` variable. The object number sent to this function is stored in the variable `item`, and then `remove()` ❽ removes it from the `in_my_pockets` list. Now that the selected item has been removed, the number of the selected item is reduced by 1 ❾, so the previous item in the list is now selected. If this means the selected item is now less than 0, the selected item is reset to 0 ❿. This happens if the player drops the first item from their inventory.

If the player's hands are now empty, the `item_carrying` variable is set to `False` ⓫. Otherwise, it's set to the number of their selected item. Finally, `display_inventory()` redraws the inventory to show the item has been removed.

### TRAINING MISSION #1

It's time to do a safety drill. Can you pick up the air canister and deliver it to the sick bay? Drop it near the middle bed. To test whether the program is working correctly, leave the room after your delivery and come back to make sure it's still there.

## EXAMINING OBJECTS

As you explore the space station, you'll want to study objects closely to see how they might help with your mission. The *examine* instruction shows the long description for

an object and works for scenery and props. By examining an object, you can also sometimes find other objects. For example, when you examine a cupboard, you might find something inside it.

Pressing the spacebar triggers the `examine_object()` function. (You added the keyboard control in [Listing 9-8](#).) Place the new function, shown in [Listing 9-10](#), after the `remove_object()` function you added in [Listing 9-9](#).

Save your program as *listing9-10.py*. Run the program using `pgzrun listing9-10.py`. You can now examine objects by walking up to or onto them and pressing the spacebar. For example, if you press the up arrow key and the spacebar when you're against the wall at the back of the room, you can examine the wall.

### *listing9-10.py*

---

```
--snip--
 item_carrying = in_my_pockets[selected_item]
 display_inventory()

def examine_object():
 ❶ item_player_is_on = get_item_under_player()
 ❷ left_tile_of_item = find_object_start_x()
 ❸ if item_player_is_on in [0, 2]: # don't describe the floor
 return
 ❹ description = "You see: " + objects[item_player_is_on][2]
 ❺ for prop_number, details in props.items():
 # props = object number: [room number, y, x]
 ❻ if details[0] == current_room: # if prop is in the room
 # If prop is hidden (= at player's location but not on map)
 if (details[1] == player_y
 and details[2] == left_tile_of_item
 and room_map[details[1]][details[2]] != prop_number):
 ❼ add_object(prop_number)
 ❽ description = "You found " + objects[prop_number][3]
 sounds.combine.play()
 ❾ show_text(description, 0)
 ❿ time.sleep(0.5)

#####
START
```

```
#####
```

```
--snip--
```

---

### *Listing 9-10: Adding the code to examine objects*

**Listing 9-10** builds on the work you've already done adding functions in this chapter. We start by getting the number of the object the player wants to examine and storing it in `item_player_is_on` ❶. At this point in the `game_loop()` function, the player's position will be on or possibly inside the item they want to examine, if it's a piece of scenery. We put the starting `x` position of the item into the variable `left_tile_of_item` ❷. If there isn't an object to examine at the player's location, the function finishes without taking any further action ❸. Ignoring an empty space feels more natural than describing the floor, especially if you make a mistake with the controls. If there is an item at the player's location, the description of the object goes into the `description` variable, taken from the long description from the `objects` dictionary ❹.

The program then checks whether there's an item hidden *inside* the item the player is examining. We use a loop to go through all the items in the `props` dictionary ❺. If an item is in the current room at the player's position, but the room map at that position doesn't contain the prop number ❻, it means the item is hidden. We therefore add the hidden object to the player's inventory ❼, and give the player a message that tells them they found something. This message uses the object's short description to tell them what they've found ❽.

At the end of the function, the description is shown ❾, and we've put a short pause here to stop it being immediately overwritten if the player holds the key down ❿.

If you want to hide props inside scenery in your own game design, make sure you give players a strong hint about where you've hidden something. In *Escape*, you might find objects in cupboards. If you see something unusual, it's usually a good idea to examine it to learn about it, and you might find something else of interest. You won't need to search every chair, bed, and wall panel though.

If you decide to hide props in wide scenery (such as a bed), make sure you hide your prop in the `x` position of the scenery item, not in a space that would be covered by 255 in the room map.

Can you find the MP3 player? It's in the sleeping quarters that belong to the person you named `FRIEND2` in Chapter 4. If you're using my code, it's in Leo's sleeping quarters.

Now that all the props are unpacked, you can relax with your yoyo and see what else you can find. In the next chapter, you'll add a new section to the program that enables you to use the props you come across.

## ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ Information about the position of props is stored in the `props` dictionary.
- ☐ The prop number is the dictionary key, and each entry contains a list with the room number and *y* and *x* positions of the prop.
- ☐ To receive a number from a function, set up a variable to store that information when you call the function. For example, `variable_name = function_name()`.
- ☐ To send a number (or anything else) back from a function, use the `return` instruction.
- ☐ The `//` operator is used for division and rounds the result down, removing any decimal in the answer.
- ☐ The `%` operator gives you the remainder after dividing two numbers: `5 % 2` is 1.
- ☐ You can change the value of variables and lists to help test the program, for example, creating a full inventory at the start. Remember to change them back afterward!
- ☐ You can hide props inside scenery, but make sure they're in the position where the scenery starts, and give players a strong hint about where it's worth searching.

## **MISSION DEBRIEF**

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

### **TRAINING MISSION #1**

The canister is in your starting room (31). The sick bay is room 41. From the starting room, go right, down all the way, left, and up.

### **TRAINING MISSION #2**

It's in the cabinet in room 47. Leave the starting room (31), and go all the way down the map.

# 10

## MAKE YOURSELF USEFUL



You've added the props to the game, so in this chapter, you'll add the code to enable astronauts to *use* objects and combine them to make new objects. These skills will be essential for your mission. You'll get a chance to rehearse them, so you're ready for any situation.

The code in this chapter is simpler than some of the listings you've seen recently and contains the answers for many of the puzzles in the *Escape* game. So I don't give away too many spoilers, I won't explain every item and solution here. For example, sometimes you might see an object number in the code, but I won't tell you the name of that object.

If you get stuck playing the game, you could read this code and use it to work out which objects are which by referring to the `objects` dictionary (see [Listing 5-6](#) and [Listing 5-8](#) in [Chapter 5](#)). That should be a last resort, though. You can solve all the puzzles by thinking like an astronaut. Ask yourself: What do you have access to that could be useful? How could you make something more useful?

## ADDING THE KEYBOARD CONTROL FOR USING OBJECTS

We'll start by adding the keyboard control in the `game_loop()` function. Open *listing9-*



10.py, your last listing in Chapter 9. We'll build on this listing.

Listing 10-1 shows the new instructions to add inside the `game_loop()` function. Add them after the keyboard controls for *drop* and *examine*, which you added in the previous chapter. These instructions start the `use_object()` function when the player presses the U key. Save the program as *listing10-1.py*. Don't try running the program yet: it won't do anything new, but it will crash if you press the U key.

*listing10-1.py*

---

--snip--

```
if keyboard.space:
 examine_object()
```

```
if keyboard.u:
 use_object()
```

--snip--

---

*Listing 10-1: Adding the keyboard control for using objects*

## ADDING STANDARD MESSAGES FOR USING OBJECTS

The function for using objects is long, so I've given it its own section in the program. Place the new `USE OBJECTS` section after the `PROP INTERACTIONS` section that you added in Chapter 9. Listing 10-2 shows the start of this new section. Add this code after the `examine_object()` function ends but before the `START` section.

*listing10-2.py*

---

--snip--

```
show_text(description, 0)
time.sleep(0.5)
```

```
#####
```

```
USE OBJECTS
```

```
#####
```

```
def use_object():
```

global room\_map, props, item\_carrying, air, selected\_item, energy  
global in\_my\_pockets, suit\_stitched, air\_fixed, game\_over

```
❶ use_message = "You fiddle around with it but don't get anywhere."
❷ standard_responses = {
 4: "Air is running out! You can't take this lying down!",
 6: "This is no time to sit around!",
 7: "This is no time to sit around!",
 32: "It shakes and rumbles, but nothing else happens.",
 34: "Ah! That's better. Now wash your hands.",
 35: "You wash your hands and shake the water off.",
 37: "The test tubes smoke slightly as you shake them.",
 54: "You chew the gum. It's sticky like glue.",
 55: "The yoyo bounces up and down, slightly slower than on Earth",
 56: "It's a bit too fiddly. Can you thread it on something?",
 59: "You need to fix the leak before you can use the canister",
 61: "You try signalling with the mirror, but nobody can see you.",
 62: "Don't throw resources away. Things might come in handy...",
 67: "To enjoy yummy space food, just add water!",
 75: "You are at Sector: " + str(current_room) + " // X: " \
 + str(player_x) + " // Y: " + str(player_y)
}

Get object number at player's location.
❸ item_player_is_on = get_item_under_player()
❹ for this_item in [item_player_is_on, item_carrying]:
❺ if this_item in standard_responses:
❻ use_message = standard_responses[this_item]
❼ show_text(use_message, 0)
 time.sleep(0.5)

#####
START
#####
--snip--
```

---

*Listing 10-2: Adding the first instructions for using objects*

Listing 10-2 shows the first part of the `use_object()` function. We'll flesh this out in

further listings in this chapter. At the end of the function, the program shows players a message to tell them what happened when they tried to use the object ⑦. That message will be in the `use_message` variable. At the start of this function, we set it up as an error message ①. Later on, it will be changed to a message of success if they used an object.

Some of the objects have no real function in the game but will reward the player with a message when they try to use them. These messages could include clues as well as add to the game story. The dictionary `standard_responses` contains messages to show players when they use certain objects ②. The dictionary key is the object number. For example, if they want to use the bed (lazy bones!), which is object 4, they see a message that says, “You can’t take this lying down!”

The variable `item_the_player_is_on` stores the object number at the player’s position in the room ③. Players can use objects they are carrying or standing on. We set up a loop that goes through a list that contains two items: the item number the player is standing on and the item number the player is carrying ④. If either of them is a key for the `standard_responses` dictionary ⑤, the `use_message` is updated to the object’s message from that dictionary ⑥. The program prioritizes items you’re carrying over items you’re standing on if they both have standard messages.

Save your file as *listing10-2.py*. Run it using `pgzrun listing10-2.py`. To test that it works, press U to use the yoyo you’re carrying.

## ADDING THE GAME PROGRESS VARIABLES

There are a few new variables we need to add to the program to store important data about the player’s progress in the game:

- `air`, which stores how much air you have available, as a percentage
- `energy`, which stores your energy, as a percentage, and will be reduced if you injure yourself
- `suit_stitched`, which stores a `True` or `False` value, depending on whether the suit has been repaired
- `air_fixed`, which stores a `True` or `False` value, depending on whether the air tank has been fixed

Add the variables to the end of the `VARIABLES` section, as shown in [Listing 10-3](#). Save your

updated program as *listing10-3.py*. This program won't do anything new if you run it: we've set up some variables but aren't doing anything with them yet.

### *listing10-3.py*

---

```
--snip--
GREEN = (0, 255, 0)
RED = (128, 0, 0)

air, energy = 100, 100
suit_stitched, air_fixed = False, False
launch_frame = 0

#####
MAP
#####
--snip--
```

---

*Listing 10-3: Adding the game progress variables*

## ADDING THE ACTIONS FOR SPECIFIC OBJECTS

The next stage in the `use_object()` function is to check particular objects to see if they have actions that can be performed with them. These checks will override any standard messages that might have been set up earlier and are shown in [Listing 10-4](#). Because these instructions are inside the `use_object()` function, they are indented by at least four spaces. Save your program as *listing10-4.py*. Run it using `pgzrun listing10-4.py`.

### *listing10-4.py*

---

```
--snip--
 if this_item in standard_responses:
 use_message = standard_responses[this_item]

❶ if item_carrying == 70 or item_player_is_on == 70:
 use_message = "Banging tunes!"
 sounds.steelmusic.play(2)
```

```

② elif item_player_is_on == 11:
③ use_message = "AIR: " + str(air) + \
 "% / ENERGY " + str(energy) + "% / "
 if not suit_stitched:
 use_message += "*ALERT* SUIT FABRIC TORN / "
 if not air_fixed:
 use_message += "*ALERT* SUIT AIR BOTTLE MISSING"
 if suit_stitched and air_fixed:
 use_message += " SUIT OK"
 show_text(use_message, 0)
 sounds.say_status_report.play()
 time.sleep(0.5)
 # If "on" the computer, player intention is clearly status update.
 # Return to stop another object use accidentally overriding this.
④ return

```

```

elif item_carrying == 60 or item_player_is_on == 60:

```

```

⑤ use_message = "You fix " + objects[60][3] + " to the suit"
 air_fixed = True
 air = 90
 air_countdown()
 remove_object(60)

```

```

elif (item_carrying == 58 or item_player_is_on == 58) \
 and not suit_stitched:

```

```

 use_message = "You use " + objects[56][3] + \
 " to repair the suit fabric"
 suit_stitched = True
 remove_object(58)

```

```

elif item_carrying == 72 or item_player_is_on == 72:

```

```

 use_message = "You radio for help. A rescue ship is coming. \
Rendezvous Sector 13, outside."
 props[40][0] = 13

```

```

elif (item_carrying == 66 or item_player_is_on == 66) \
 and current_room in outdoor_rooms:

```

```

 use_message = "You dig..."
 if (current_room == LANDER_SECTOR

```

```

 and player_x == LANDER_X
 and player_y == LANDER_Y):
 add_object(71)
 use_message = "You found the Poodle lander!"

elif item_player_is_on == 40:
 clock.unschedule(air_countdown)
 show_text("Congratulations, " + PLAYER_NAME + "!", 0)
 show_text("Mission success! You have made it to safety.", 1)
 game_over = True
 sounds.take_off.play()
 game_completion_sequence()

elif item_player_is_on == 16:
 energy += 1
 if energy > 100:
 energy = 100
 use_message = "You munch the lettuce and get a little energy back"
 draw_energy_air()

elif item_carrying == 68 or item_player_is_on == 68:
 energy = 100
 use_message = "You use the food to restore your energy"
 remove_object(68)
 draw_energy_air()

if suit_stitched and air_fixed: # open airlock access
 if current_room == 31 and props[20][0] == 31:
 open_door(20) # which includes removing the door
 sounds.say_airlock_open.play()
 show_text("The computer tells you the airlock is now open.", 1)
 elif props[20][0] == 31:
 props[20][0] = 0 # remove door from map
 sounds.say_airlock_open.play()
 show_text("The computer tells you the airlock is now open.", 1)

show_text(use_message, 0)
time.sleep(0.5)

```

```

START ##

--snip--
```

---

#### *Listing 10-4: Adding the ability to use certain objects*

Listing 10-4 includes a series of instructions that check whether the object that's being used is a particular object number. If so, the instructions for that object are carried out.

For example, if the player is carrying or standing on object 70 ❶, which is an MP3 player, they'll see a message that says "Banging tunes!" and hear some music. If the player is using the computer ❷, the message shown is made by combining information from the `air` and `energy` variables, and adding an alert if the suit or air bottle is faulty. There's also a computer speech sound effect here that says "status report!"

I've included a `return` instruction at the end of this set of instructions ❸, which prevents the player from accidentally using another object when they intended to use the computer. If we didn't include this `return` instruction, the player might end up using another prop that they're carrying instead of the computer. Keeping the controls simple means there can be some ambiguity about what the player intended to use, but the game is designed to prioritize results that help the player complete the game.

In a couple of places, I've used the short description from the `objects` dictionary instead of typing the name of the object into the string ❹. That's to prevent you from seeing any spoilers in the code!

The `\` symbol at the end of a line ❺ tells Python that the code continues on the next line. Some of the lines here are quite long, so I've used this symbol to break them up and to fit them on the book page.

Try out some of the new code by walking into one of the computer terminals and pressing the U key. You'll see a status update. If you can find the MP3 player, you can listen to that too.

#### **RED ALERT**

*Be particularly careful when you're entering the object numbers and the rest*



of the code in [Listing 10-4](#). If you make a mistake here, you might not be able to complete the puzzles in the game!

## COMBINING OBJECTS

Some of the puzzles in the game require you to use objects together. For example, you might use one object as a tool to do something to the other object, or you might join two objects together. For instance, one of the puzzles requires you to insert a GPS module into a positioning system. When you find the two parts, you need to combine them to make a working positioning system. To use two objects together, you select one in your inventory and walk on or into the other one. You might need to drop an object from your inventory onto the floor so you can work on it with another object you're carrying.

In the *Escape* game engine, combinations are called *recipes*. A single recipe contains three object numbers in a list. The first two are the items that are combined, and the third one is the object number they make when they're combined. Here's an example:

---

[73, 74, 75]

---

Object 73 (a GPS module) plus object 74 (a positioning system) makes object 75 (a working positioning system).

When you combine objects, the new object goes into your inventory. The objects you combined are removed from the game if they're props. Sometimes one will be a piece of scenery and so will remain in the game.

[Listing 10-5](#) shows you the list of recipes. Add it to the end of the `PROPS` part of your program where the information for props is set up. Save your file as *listing10-5.py*. The listing shouldn't do anything new yet if you run it, but it will check that the new data is correct.

*listing10-5.py*

---

```
--snip--
in_my_pockets = [55]
selected_item = 0 # the first item
item_carrying = in_my_pockets[selected_item]
```

```
RECIPES = [
```

```
[62, 35, 63], [76, 28, 77], [78, 38, 54], [73, 74, 75],
[59, 54, 60], [77, 55, 56], [56, 57, 58], [71, 65, 72],
[88, 58, 89], [89, 60, 90], [67, 35, 68]
]
```

```
checksum = 0
check_counter = 1
for recipe in RECIPES:
 checksum += (recipe[0] * check_counter
 + recipe[1] * (check_counter + 1)
 + recipe[2] * (check_counter + 2))
 check_counter += 3
print(len(RECIPES), "recipes")
assert len(RECIPES) == 11, "Expected 11 recipes"
assert checksum == 37296, "Error in recipes data"
print("Recipe checksum:", checksum)
```

```
#####
PROP INTERACTIONS
#####
--snip--
```

---

### *Listing 10-5: Adding recipes to the Escape game*

Now add the code to use the recipes near the end of the `use_object()` function, as shown in [Listing 10-6](#). Add it to your `use_object()` function, and save the program as *listing10-5.py*. When you run the program, using `pgzrun listing10-5.py`, you'll be able to combine objects.

### *listing10-6.py*

```
--snip--
 sounds.say_airlock_open.play()
 show_text("The computer tells you the airlock is now open.", 1)
```

- ❶ `for recipe in RECIPES:`
- ❷ `ingredient1 = recipe[0]`  
`ingredient2 = recipe[1]`

```

combination = recipe[2]
❸ if (item_carrying == ingredient1
 and item_player_is_on == ingredient2) \
❹ or (item_carrying == ingredient2
 and item_player_is_on == ingredient1):
❺ use_message = "You combine " + objects[ingredient1][3] \
 + " and " + objects[ingredient2][3] \
 + " to make " + objects[combination][3]
❻ if item_player_is_on in props.keys():
❼ props[item_player_is_on][0] = 0
❽ room_map[player_y][player_x] = get_floor_type()
❾ in_my_pockets.remove(item_carrying)
❿ add_object(combination)
 sounds.combine.play()

 show_text(use_message, 0)
 time.sleep(0.5)
--snip--

```

---

### *Listing 10-6: Combining objects in the game*

You might find you can work out what's going on in this new code: it mostly combines ideas you've seen before. We use a loop to go through all the items in the `RECIPES` list ❶, and a new recipe goes into the `recipe` list each time. We put the ingredients and combination object numbers into variables to make the function easier to understand ❷.

The program checks whether the player is carrying the first ingredient and standing on the second one ❸, or the other way around ❹. If so, the use message is updated to tell them what they combined and what they made ❺.

When the combined object is made, it usually replaces the ingredient objects. If one of the objects is scenery instead of a prop, though, it remains in the game. So the program checks whether the item the player is on is a prop ❻, and if so, its room number is set to 0, removing it from the game ❼. If it's a prop, it's also deleted from the room map for the current room ❽.

The object that was being carried is removed from the player's inventory ❾, and the newly created object is added to it ❿.

## TRAINING MISSION #1

Let's do a simple test to check that the combination code is working. We'll need to hack the code a bit for this test. In the `PROPS` section, change the line that sets up `in_my_pockets` so you're carrying items 73 and 74:

---

```
in_my_pockets = [55, 73, 74]
```

---

Now run the program: you'll be carrying the GPS module and the positioning system. Drop one of them and stand on it. Select the other one in your inventory, and press U. The items should be combined into a working GPS system! You can use it to see your location in the game. To be certain the code is working, try switching the objects so you're standing on the other one this time.

Make sure you change the code back again afterward:

---

```
in_my_pockets = [55]
```

---

## ADDING THE GAME COMPLETION SEQUENCE

There is one final function in the `USE OBJECTS` part of the program, which is a short animation that plays when the player completes the game: the astronaut takes off in the rescue ship. Add this function to the end of your `USE OBJECTS` section, as shown in [Listing 10-7](#):

*listing10-7.py*

---

```
--snip--
```

```
show_text(use_message, 0)
time.sleep(0.5)
```

```
def game_completion_sequence():
```

```
 global launch_frame #(initial value is 0, set up in VARIABLES section)
```

```
 box = Rect((0, 150), (800, 600))
```

```
 screen.draw.filled_rect(box, (128, 0, 0))
```

```
 box = Rect((0, top_left_y - 30), (800, 390))
```

```
 screen.surface.set_clip(box)
```

```

for y in range(0, 13):
 for x in range(0, 13):
 draw_image(images.soil, y, x)

launch_frame += 1
if launch_frame < 9:
 draw_image(images.rescue_ship, 8 - launch_frame, 6)
 draw_shadow(images.rescue_ship_shadow, 8 + launch_frame, 6)
 clock.schedule(game_completion_sequence, 0.25)
else:
 screen.surface.set_clip(None)
 screen.draw.text("MISSION", (200, 380), color = "white",
 fontsize = 128, shadow = (1, 1), scolor = "black")
 screen.draw.text("COMPLETE", (145, 480), color = "white",
 fontsize = 128, shadow = (1, 1), scolor = "black")
 sounds.completion.play()
 sounds.say_mission_complete.play()

#####
START
#####
--snip--

```

---

*Listing 10-7: Blast off!*

## EXPLORING THE OBJECTS

Now you can explore the objects you find in the space station and try using them to see what they do. Before you can find all the props, though, and get to work on the space station, you'll need to work out how to open the safety doors that seal off parts of the space station. In the next chapter, you'll complete the space station setup by engineering the door mechanism to open when you use the correct access pass.

You can also use what you've learned in this chapter to add your own puzzles to the *Escape* game code. The simplest approach is to use standard messages ([Listing 10-2](#)) for clues and to use recipes ([Listing 10-5](#)) to combine objects. You can also add simple instructions ([Listing 10-4](#)) to see whether the player is carrying a particular object, and then increase their `air` or `energy` variables, display a message, or do something else in the game. Happy adventuring!

## ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ The instructions for using objects go into the `use_object()` function.
- ☐ The `standard_responses` dictionary contains messages for when the player uses a particular object.
- ☐ For many objects, there are specific instructions to update different lists or variables when the player uses them.
- ☐ The `RECIPES` list stores the details of how the player can combine objects in the game.
- ☐ In a recipe, the first two items are ingredients, and the third item is what they make.

# 11

## ACTIVATING SAFETY DOORS



On the space station, doors restrict access to certain zones and ensure that astronauts can only get into areas where they're qualified to work. Many doors require personal access passes to open, and the engineering bay doors can only be opened with a button in Mission Control. The engineering bay doors also have a timer that closes them automatically to increase security.

The doors also enforce safety rules that require astronauts to have a working suit before they can enter the airlock and to have a buddy with them before the door to the planet surface can open. Footage from the security camera suggests that some astronauts have found a way to bypass the buddy requirement so they can enjoy the serenity of a solo walk on the planet's surface.

You installed the doors in the space station when you installed the props. In this chapter, you'll add the code to open and close the doors, as well as add a few other tricks and puzzles to make the game more interesting.

### PLANNING WHERE TO PUT SAFETY DOORS

Doors are clearly a vital part of the space station design, but they're also important for the game's design. Most obviously, they present a challenging puzzle: players need to

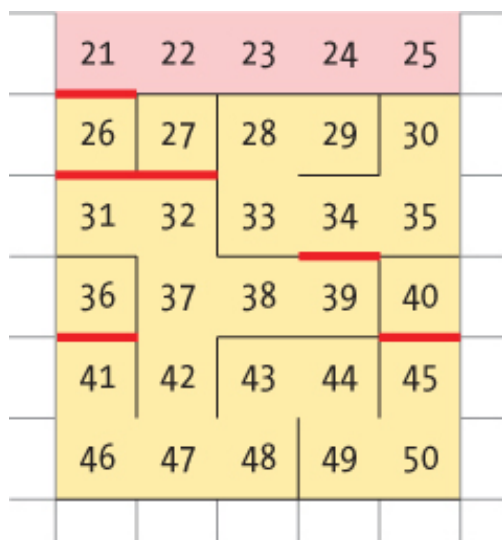


find a way to open locked doors.

The doors also help us to tell a story, in which there are obstacles that the hero must overcome using their survival training and logical thinking. The game's puzzles will only be satisfying if the player has to think about them a little bit. So it's important that we can control when players see the different puzzle elements. Imagine you enter a room and there's rampant fire blocking the other exit. If you're already carrying a fire extinguisher, you just whip it out and use it. There's no real challenge. It's more intriguing if you see the threat (or the puzzle), and then have to figure out the solution. By sealing off parts of the map, we can guide players to see a problem before they see its solution. We can't be certain they'll notice everything we put in their path, but we can give them an opportunity to experience the game at its best.

Doors also enable us to get more value from the map. Although it might not feel like it after typing it in, the game map isn't huge. We can provide a richer experience and a longer game by requiring players to cross difficult rooms more than once. For example, if we put a key at the end of a corridor, we can direct the player to retrace their steps along the corridor and use the key in a door they passed on the way.

Figure 11-1 shows the location of doors in the game. Without giving too much away, players won't be able to get into room 36 before they've gotten into the top-right section of the space station (via room 34). They won't be able to visit room 27 until they've gotten into room 40, either. By strategically placing items in the locked rooms, including access cards, we can direct the player through the game and through the story.



*Figure 11-1: The game map with doors shown in red*

When you're designing your own games, think carefully about where you put your props. It's one of the most important elements in ensuring the game presents players

with an enjoyable challenge.

## POSITIONING THE DOORS

I've positioned all the doors in *Escape* at the top or bottom exit of a room because of the game's top-down perspective. If a door was in a side exit, players would only see its top surface, and we need to make sure something as important as a door can be clearly seen.

Most of the doors are at the top of the room and remain open after the player opens them. The exception is the door between rooms 32 and 27, which has a timer mechanism that shuts it automatically. This timer provides an additional challenge: the player must rush to get to the room from the switch that opens the door, before the door closes.

The doors in *Escape* are objects 20 to 26. Their images and descriptions are set up in the `objects` dictionary (see “[Making the Space Station Objects Dictionary](#)” on page 85). The door positions are set up in the `props` dictionary (see “[Adding the Props Information](#)” on page 151). Each door has an `x` position that puts it in the room's doorway. To work out the `x` position for a door, just divide the room width by 2, round it down, and then subtract 1.

Now let's add some controls to enable players to open the doors.

## ADDING ACCESS CONTROLS

To enable the player to open the doors, we need to add some instructions to the `use_object()` function in the `USE OBJECTS` part of the program. One new code snippet will open the timed door to the engineering bay when the player presses a button in one of the rooms. You'll add this code between the instructions for handling objects 16 and 68.

The other new code addition will enable the player to use access cards to open the doors: put this after the code for using recipes.

[Listing 11-1](#) shows the new code to add. Because these instructions are part of the `use_object()` function, the first one is indented by four spaces. Your new `elif` instruction should line up with the `elif` instruction above it.

Open *listing10-7.py* from the previous chapter and add these new lines to it. Save your program as *listing11-1.py*. You can run it using `pgzrun listing11-1.py`, but we haven't added all the code necessary to make the doors work properly yet. You shouldn't see

any error messages, though.

*listing11-1.py*

---

```
--snip--
```

```
elif item_player_is_on == 16:
 energy += 1
 if energy > 100:
 energy = 100
 use_message = "You munch the lettuce and get a little energy back"
 draw_energy_air()
```

```
❶ elif item_player_is_on == 42:
❷ if current_room == 27:
❸ open_door(26)
❹ props[25][0] = 0 # Door from RM32 to engineering bay
 props[26][0] = 0 # Door inside engineering bay
❺ clock.schedule_unique(shut_engineering_door, 60)
 use_message = "You press the button"
 show_text("Door to engineering bay is open for 60 seconds", 1)
 sounds.say_doors_open.play()
 sounds.doors.play()
```

```
elif item_carrying == 68 or item_player_is_on == 68:
 energy = 100
 use_message = "You use the food to restore your energy"
 remove_object(68)
 draw_energy_air()
```

```
--snip--
```

```
for recipe in RECIPES:
 ingredient1 = recipe[0]
 ingredient2 = recipe[1]
```

```
--snip--
```

```
 add_object(combination)
 sounds.combine.play()
```

```
{key object number: door object number}
```

```

❹ ACCESS_DICTIONARY = { 79:22, 80:23, 81:24 }
❺ if item_carrying in ACCESS_DICTIONARY:
 door_number = ACCESS_DICTIONARY[item_carrying]
❻ if props[door_number][0] == current_room:
 use_message = "You unlock the door!"
❼ sounds.say_doors_open.play()
 sounds.doors.play()
 open_door(door_number)

show_text(use_message, 0)
time.sleep(0.5)

```

--snip--

---

### *Listing 11-1: Adding the ability to open doors*

The button to open the door to the engineering bay is object 42. There is one of these buttons outside the engineering bay to provide access, and another inside the engineering bay, so the player doesn't get trapped inside.

If the player is using the button ❶, the code to open the door runs. If they're using the button inside the room ❷, the `open_door()` function is used to show the door opening ❸. We'll add that function shortly.

The `props` dictionary is updated to change the room number for the door to 0, removing the door from the room (and from the game) ❹. This door works on a timer, so the program schedules the function to close the door 60 seconds later ❺. If you find it too difficult to get to the room in time, you can change the number 60 to a larger number. This number should give you just about enough time, whether you're using a PC or Raspberry Pi 3; or a Raspberry Pi 2, where the game runs a little bit more slowly.

The second chunk of code enables players to use keys to open the doors. We create a new dictionary called `ACCESS_DICTIONARY` that uses the access card number as the dictionary key and the door number as the data ❹. So object 79 (an access card) is used to open door 22, for example.

#### **TIP**

The objects used to open the doors in *Escape* are all access cards, but if

you're modifying the game, you could use any object. You could use a crowbar to pry doors open, or (if you make a game set in a fantasy world) you could use different magic spells. Just make sure players can reasonably work out what to use.

When the player presses U, the door opens if they have selected one of the items in the dictionary for unlocking doors ⑦ and if they are standing in the same room as the door it unlocks ⑧. We also play a sound effect of a computer voice saying “doors open” ⑨. This is just a recording, like any other sound in the game.

## MAKING THE DOORS OPEN AND CLOSE

We'll place the functions for opening, closing, and animating the doors into a new `DOORS` section of the program. You need to add this section after the `USE OBJECTS` section but before the `START` section at the end.

Listing 11-2 shows the first two functions you need to add to start the `DOORS` section. Add the new lines, and save your program as *listing11-2.py*. The `DOORS` section is still incomplete: you can run the program (using `pgzrun listing11-2.py`) to check for errors, but the doors won't work yet.

*listing11-2.py*

---

```
--snip--
 sounds.completion.play()
 sounds.say_mission_complete.play()

#####
DOORS
#####

❶ def open_door(opening_door_number):
 global door_frames, door_shadow_frames
 global door_frame_number, door_object_number
❷ door_frames = [images.door1, images.door2, images.door3,
 images.door4, images.floor]
 # (Final frame restores shadow ready for when door reappears).
 door_shadow_frames = [images.door1_shadow, images.door2_shadow,
```

```

 images.door3_shadow, images.door4_shadow,
 images.door_shadow]
 door_frame_number = 0
 door_object_number = opening_door_number
❸ do_door_animation()

❹ def close_door(closing_door_number):
 global door_frames, door_shadow_frames
 global door_frame_number, door_object_number, player_y
❺ door_frames = [images.door4, images.door3, images.door2,
 images.door1, images.door]
 door_shadow_frames = [images.door4_shadow, images.door3_shadow,
 images.door2_shadow, images.door1_shadow,
 images.door_shadow]
 door_frame_number = 0
 door_object_number = closing_door_number
 # If player is in same row as a door, they must be in open doorway
❻ if player_y == props[door_object_number][1]:
❼ if player_y == 0: # if in the top doorway
❽ player_y = 1 # move them down
 else:
❾ player_y = room_height - 2 # move them up
❿ do_door_animation()

#####
START
#####
--snip--

```

---

### *Listing 11-2: Setting up the door animations*

The `open_door()` and `close_door()` functions set up the door animations for opening and closing. You’ve already seen `open_door()` ❶ mentioned in [Listing 11-1](#). In [Listing 11-2](#), we define that function so it can run if the player opens a door using a key, for example.

The door animation has five frames, numbered 0 to 4, as shown in [Table 11-1](#). We store images for the animation in a list called `door_frames` ❷❺ and store the frame number in the variable `door_frame_number`. In the `open_door()` and `close_door()` functions, we set the

frame number to 0, the first frame.









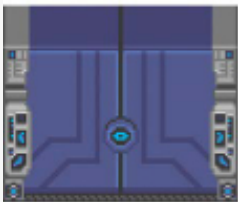
In the variable `door_object_number`, we store the object number of the door that will be opening or closing. After the variables and list have been set up, the function `do_door_animation()` is started to carry out the animation using them ③⑩. We'll add that function shortly.

The function for closing the door ④ is similar to the function for opening the door ① with two exceptions: the animation frames are different, and there is a check to stop the door from closing on top of the player.

If the player is in the same *y* position as the door ⑥, it means the player is standing in the doorway. In that case, if the player is in the top row ⑦, we set their *y* position to 1 ⑧ to move them to the next row down. If the player is not in the top row, we set their *y* position to the second row from the bottom ⑨, just inside the door.

This means the astronaut jumps out of the way of the doors of their own accord, but it's more realistic than them ending up inside the door!

**Table 11-1:** The Animation Frames for the Doors

| Frame number | 0                                                                                   | 1                                                                                   | 2                                                                                   | 3                                                                                    | 4                                                                                     |
|--------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Opening      |  |  |  |  | Final frame is a floor tile (no door).                                                |
| Closing      |  |  |  |  |  |



## ADDING THE DOOR ANIMATION

The `do_door_animation()` function will manage the animation of the doors opening and closing.

Place the `do_door_animation()` function inside the `DOORS` section of the program, after the `close_door()` function you added in [Listing 11-2](#). Add the new lines in [Listing 11-3](#), and save your program as *listing11-3.py*. You can run this version of the game using `pgzrun listing11-3.py`. The doors that are opened with a key should now be working. I'll tell you how to test them in Training Mission #1 shortly.

### *listing11-3.py*

---

```
--snip--
 player_y = room_height - 2 # move them up
 do_door_animation()

def do_door_animation():
 global door_frames, door_frame_number, door_object_number, objects
 ❶ objects[door_object_number][0] = door_frames[door_frame_number]
 objects[door_object_number][1] = door_shadow_frames[door_frame_number]
 ❷ door_frame_number += 1
 ❸ if door_frame_number == 5:
 ❹ if door_frames[-1] == images.floor:
 ❺ props[door_object_number][0] = 0 # remove door from props list
 # Regenerate room map from the props
 # to put the door in the room if required.
 ❻ generate_map()
 ❼ else:
 ❽ clock.schedule(do_door_animation, 0.15)

#####
START
#####
--snip--
```

---

### *Listing 11-3: Adding the door animation*

The `objects` dictionary contains, among other things, the image to use for a particular object. This new function starts by changing the door's image in that dictionary to the

current animation frame ❶. When the room is redrawn, it will now use that animation frame.

The function then increases the animation frame number by 1 ❷ so the next animation frame can be shown next time this function runs. If the frame is now 5, it means we've reached the end of the animation ❸. In that case, we check whether the door has opened (rather than closed) by seeing whether the final frame was a floor tile, showing no door ❹. (An index number of -1 gives you the last item in a list.)

If the door has now fully opened, the props data is updated to remove this door from the game by changing its room number to 0 ❺. If the current animation frame is the final frame, whether the door is opening or closing, a new room map is generated ❻, which ensures the door is added or removed correctly in the current room.

If the current frame isn't the final animation frame ❼, the function sets itself to run again in 0.15 seconds ❽ to show the next frame in the sequence.

You may be wondering why I didn't combine the two `if` instructions ❸❹. The reason is that the `generate_map()` function needs to run at the end of the animation, whether the door is opening or closing. If we combined the two `if` instructions, this function would only run when the door had opened.

### TRAINING MISSION #1

At this point in the program, the doors should be fully functional. Can you test that they work? Find the access card for the door in the community room and use it. Stand in the community room and use the access card for its door by selecting the access card in your inventory and pressing U. If you need a hint, look at the map in [Figure 11-1](#). The community room is number 39, and the key for it is in room 41. Remember that people sometimes tidy things away, and the key might not be lying in plain sight.

## SHUTTING THE TIMED DOOR

Next, we need to add a new function called `shut_engineering_door()` to shut the door to the engineering bay automatically. This function is set to run after a delay of 60 seconds when the door is opened (see [Listing 11-1](#)), giving the player a minute to run from the button to the door before it shuts!

Put this function in the `DOORS` section of the program after the `do_door_animation()` function you just added. Add the new lines in [Listing 11-4](#), and save the program as *listing11-4.py*. Then run this program using `pgzrun listing11-4.py`. You should see no error messages. The timed door should be working now, but I'll show you an easier way to test it shortly.

*listing11-4.py*

---

```
--snip--
 else:
 clock.schedule(do_door_animation, 0.15)

def shut_engineering_door():
 global current_room, door_room_number, props
 ❶ props[25][0] = 32 # Door from room 32 to the engineering bay.
 ❷ props[26][0] = 27 # Door inside engineering bay.
 ❸ generate_map() # Add door to room_map for if in affected room.
 ❹ if current_room == 27:
 ❺ close_door(26)
 ❻ if current_room == 32:
 ❼ close_door(25)
 show_text("The computer tells you the doors are closed.", 1)
 sounds.say_doors_closed.play()

#####
START
#####
--snip--
```

---

*Listing 11-4: Adding the code to shut the engineering door automatically*

The `shut_engineering_door()` function has two door props to work with, objects 25 and 26, because the player can see this door from either side depending on which room they're in. The first thing we do is update the `props` dictionary so these doors appear in the rooms ❶❷.

We then call the `generate_map()` function ❸. If the player is in a room with one of these doors, this function updates the room map for the current room. In other cases, the

`generate_map()` function still runs, but nothing changes.

If the player is in the engineering bay (room 27) ④, they need to see door 26 closing ⑤, so the program starts the animation. If the player is on the other side of the door, in room 32 ⑥, we need to show them door 25 closing ⑦.

### RED ALERT

*Don't mix up door numbers and room numbers. Door numbers are object numbers and aren't related to the room they're in.*

To test that the engineering bay door is working correctly, we'd have to run the game, press the button, and race to the engineering bay. So to save time, let's engineer a solution that enables us to get around the space station more quickly.

## ADDING A TELEPORTER

While you're still building the space station, you might find it helpful to be able to jump to any room in an instant. Using the latest in molecular transfer technology, we can install a *teleporter* that allows you to type in a room number and go straight there. This is a huge benefit when you're testing the game, but it's a restricted technology and isn't approved for use in a real mission on the space station. You'll need to remove it before you finish building the game. I'm trusting you with highly classified technology here.

Place the teleporter code with the other player controls in the `game_loop()` function, in the `GAME LOOP` part of the program. I recommend that you add it after the instructions for starting the `use_object` function. Because these instructions are inside a function, you need to indent the `if` instruction by four spaces and then indent the instructions under it by four more spaces.

Add the new instructions in [Listing 11-5](#), and then save your file as *listing11-5.py*. You can run this program using `pgzrun listing11-5.py`.

*listing11-5.py*

---

--snip--

```
if keyboard.u:
```

```
use_object()
```

```
Teleporter for testing
```

```
Remove this section for the real game
```

```
❶ if keyboard.x:
❷ current_room = int(input("Enter room number:"))
❸ player_x = 2
 player_y = 2
❹ generate_map()
❺ start_room()
 sounds.teleport.play()
Teleport section ends
```

```
--snip--
```

---

### *Listing 11-5: Adding a teleporter*

When you press the X key ❶, the program will ask you to type in a room number ❷. This request appears in the command line window where you type in your `pgzrun` instruction to run the program. You might need to click this window to bring it to the front and will need to click the game window to play again afterwards.

The `input()` function takes whatever you enter and puts it in a string. Because we need the input as a number, we use the `int()` function to convert it to an integer (or whole number) ❷.

The number you enter goes into the `current_room` variable. There's no error checking here, so the program might crash if you don't enter a valid room number. If you enter text instead of a number, for example, the program freezes.

You're teleported to position  $y = 2, x = 2$  ❸ inside the room you choose. This is usually a fairly safe place to be, but if the teleporter puts you inside some scenery, you can usually just walk out of it. The room map is regenerated ❹, and the room is restarted ❺, completing your teleportation to your new destination.

#### **TRAINING MISSION #2**

Use the teleporter to beam into room 27 so you can test the door in the engineering bay. Use the button at the top of the room to open the door

(press U while walking into the button), and wait in the room until the door closes. Open the door again, but this time leave the room and check that the door still closes when seen from the other side. The door animation should work correctly.

## ACTIVATING THE AIRLOCK SECURITY DOOR

As a safety feature, the airlock door to the planet's surface uses a weight sensor to open it. One astronaut must stand on the pressure pad to open the door, enabling another one to walk through it. This design ensures that astronauts cannot go out onto the planet's surface without support in the space station.

To enable this safety feature, we'll need to add a new function to the program's `DOORS` section. [Listing 11-6](#) shows the code for the new function, which animates the door. Add this code after the `shut_engineering_door()` function you added in [Listing 11-4](#). Save your updated program as *listing11-6.py*. You can run your program using `pgzrun listing11-6.py`, but the airlock door is not activated yet.

*listing11-6.py*

---

--snip--

```
show_text("The computer tells you the doors are closed.", 1)
sounds.say_doors_closed.play()
```

```
def door_in_room_26():
```

```
 global airlock_door_frame, room_map
```

```
 ❶ frames = [images.door, images.door1, images.door2,
 images.door3, images.door4, images.floor
]
```

```
 shadow_frames = [images.door_shadow, images.door1_shadow,
 images.door2_shadow, images.door3_shadow,
 images.door4_shadow, None]
```

```
 ❷ if current_room != 26:
 clock.unschedule(door_in_room_26)
 return
```

```
 # prop 21 is the door in Room 26.
```

```

❸ if ((player_y == 8 and player_x == 2) or props[63] == [26, 8, 2]) \
 and props[21][0] == 26:
❹ airlock_door_frame += 1

❺ if airlock_door_frame == 5:
 props[21][0] = 0 # Remove door from map when fully open.
 room_map[0][1] = 0
 room_map[0][2] = 0
 room_map[0][3] = 0

❻ if ((player_y != 8 or player_x != 2) and props[63] != [26, 8, 2]) \
 and airlock_door_frame > 0:
 if airlock_door_frame == 5:
 # Add door to props and map so animation is shown.
 props[21][0] = 26
 room_map[0][1] = 21
 room_map[0][2] = 255
 room_map[0][3] = 255
 airlock_door_frame -= 1

❼ objects[21][0] = frames[airlock_door_frame]
 objects[21][1] = shadow_frames[airlock_door_frame]

```

```

#####
START
#####
--snip--

```

---

### *Listing 11-6: Adding the weight-activated door in the airlock*

I've added the `door_in_room_26()` function to the game to enable a specific puzzle. To avoid telling you the solution and spoiling the puzzle, I won't cover everything that's in the code here, but I'm sure you can work it out if you want to!

We store the animation frames for the door in the list `frames`, including the first frame that shows the door shut and the final frame that shows an empty floor tile instead of the door ❶.

We store the animation frame for the airlock door in the `airlock_door_frame` variable. If



the player is standing on the pressure pad (at position  $y = 8$  and  $x = 2$ ) and the door is in the room ③, the animation frame number is increased to open the door a bit more ④. If the animation frame is now 5 ⑤, then the door is fully opened, and the `props` dictionary and room map are updated to remove the door from the room.

We add another section of code to close the door when the player is *not* standing on the pressure pad and the door is already at least partially open ⑥, so the door closes if the player moves off the pressure pad. The program only displays props that are in the room map for the current room, so the first instructions put the door (object 21) into the room map, even though the first animation frame will show the door fully open.

Finally, we change the image file for the door in the `objects` dictionary to the current animation frame ⑦. The door's shadow image is also updated. As a result, when the room is drawn, the picture for the door shows its current animation frame.

This airlock routine creates a smooth effect where the door slides open when the player steps on the pressure pad, but slides shut again the moment they walk off. If they step back onto the pad while the door is shutting, it starts to open again.

To make the airlock routine work, we also need to add the instruction to make the `door_in_room_26()` function run every 0.05 seconds when the player enters the room. When the `door_in_room_26()` function starts, it checks whether the player is still in room 26. If the player has left the room, the instructions at ② in [Listing 11-6](#) stop the function from running regularly and exit the function (using a `return` instruction) so that the door animation stops.

We'll put the code that starts the `door_in_room_26()` function into the `start_room()` function at the top of the `GAME LOOP` section. The `start_room()` function runs when the player enters a room. [Listing 11-7](#) shows the new instructions to add.

#### *listing11-7.py*

---

```
--snip--
#####
GAME LOOP
#####

def start_room():
 global airlock_door_frame
 show_text("You are here: " + room_name, 0)
```

```
if current_room == 26: # Room with self-shutting airlock door
 airlock_door_frame = 0
 clock.schedule_interval(door_in_room_26, 0.05)
```

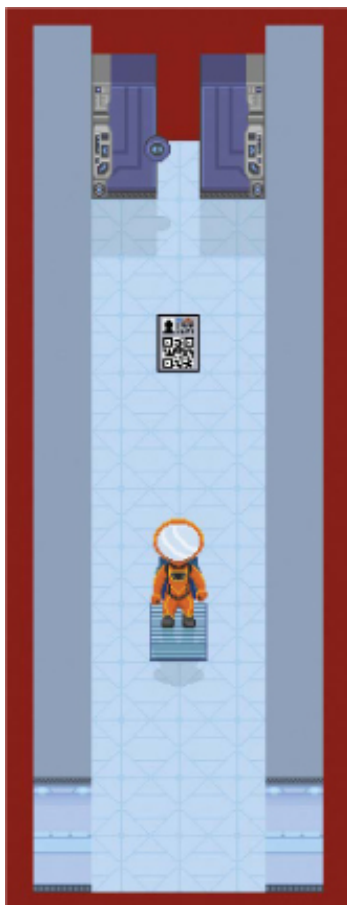
--snip--

---

### *Listing 11-7: Scheduling the door animation for the airlock*

Save your program as *listing11-7.py* and run it using `pgzrun listing11-7.py`. In the game, press X to use the teleporter and beam into room 26. Now you can test that the pressure pad works as expected (see [Figure 11-2](#)). Try walking on it, off it, and across it to see how the door behaves.

Note that if you leave through the exit at the bottom of this room, a door appears that blocks your way back again. (Normally, you would only enter the airlock by opening that door and removing it from the game.) When you teleport into rooms, strange things like this might happen. It messes with the space-time continuum.



*Figure 11-2: Standing on the pressure pad opens the door.*

## REMOVING EXITS FOR YOUR OWN GAME DESIGNS

If you're closing off exits for your own map designs, you might need to move or remove

doors in those exits as well. To remove a door from the game, change the entry for that door in the `props` dictionary so its first number is a 0, or delete its entry from the dictionary.

If you're customizing the game, you might also want to remove some of the custom code here that enables the special doors for the engineering bay and the airlock. To disable the pressure pad door, remove the new code in [Listings 11-6](#) and [11-7](#). To remove the timed door to the engineering bay, remove the code shown in [Listing 11-4](#), and additionally remove the first chunk of new code in [Listing 11-1](#) for pressing the button (using object 42).

## MISSION ACCOMPLISHED?

You've now finished building the space station, and it's fully functional. It seems you can now settle into your new life, conducting experiments and exploring the red planet.

But, wait! What's this? There could be trouble ahead.

## ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ Doors can seal off parts of the game map, so players can discover puzzle elements in the right order.
- ☐ Doors need to go at the top or the bottom of a room.
- ☐ Doors that are opened with access cards stay open.
- ☐ You can use the functions provided to add doors that close automatically, such as the door in the engineering bay.
- ☐ Doors are positioned using the `props` dictionary. Their images and descriptions are stored in the `objects` dictionary.
- ☐ To animate the door, the program changes its image in the `objects` dictionary. When the room is redrawn, the new image is used for the door.
- ☐ If a door can be seen from both sides, it needs to be represented with two door props: one in each of the rooms where it can be seen.
- ☐ `ACCESS_DICTIONARY` is used to remember which access cards unlock which doors. You

could use other objects to open doors by making changes in this dictionary.

- ☐ To adjust the difficulty of the game, you can change the delay before the engineering door slams shut.
- ☐ The teleporter enables you to beam into any room for testing purposes.
- ☐ The `input()` function in Python treats what you enter as a string. To enable players to type in a number, use the `int()` function to convert what they enter to an integer.

# 12

## DANGER! DANGER! ADDING HAZARDS

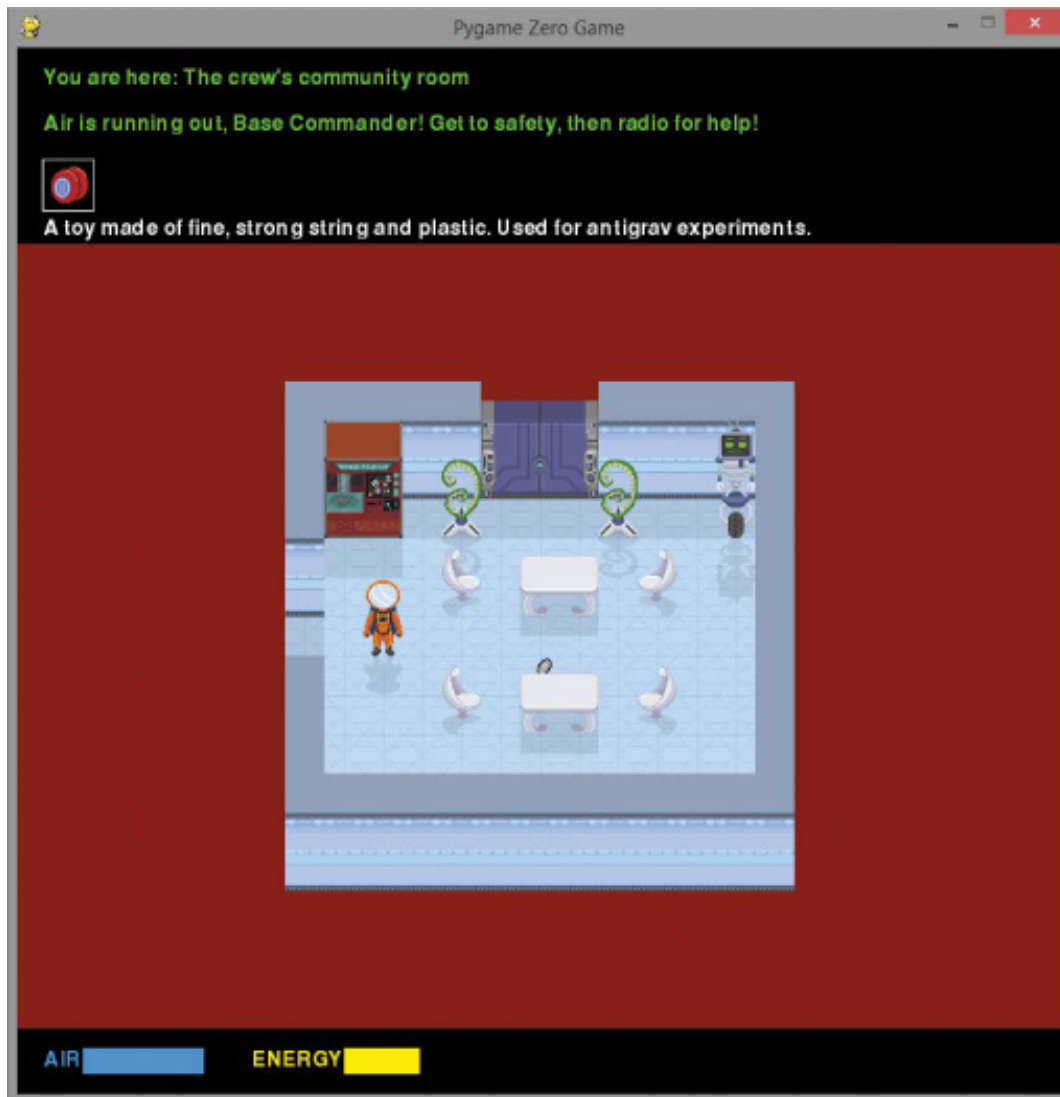


When the space station systems fail, all kinds of threats emerge. In this chapter, you'll see the air start to leak from the station and will discover moving hazards in some of the rooms, including rogue robots, balls of energy, and toxic puddles.

I've put the dangers last so you can test the game up to this point without worrying about your time or energy running out. In this chapter, we'll start the air leak and draw a timer bar to let you know how much air you have left. We'll also add hazards that can harm you and drain your energy. Finally, we'll clean up the game and get it ready to play!

### ADDING THE AIR COUNTDOWN

There are two ways for the player to fail in the game: their air can run out or their energy can run out. At the bottom of the screen, two bars show players how much air and how much energy they have remaining (see [Figure 12-1](#)).



*Figure 12-1: Two bars at the bottom of the screen show your remaining air and energy.*

You lose energy when you walk over toxic spills or are hit by moving hazards, and the air gradually runs out because of the leak in the space station wall. If you put on a space suit, you can buy more time, but the air in the suit's canister will eventually run out too. Some of your toughest decisions could be deciding when to top up your air and use food to restore your energy.

## DISPLAYING THE AIR AND ENERGY BARS

We'll create a new section of the program called `AIR`, which you need to place after the `DOORS` section but before the `START` section at the end of the program. Add the new code shown in [Listing 12-1](#) to your final listing from the previous chapter (*listing11-7.py*). Save your file as *listing12-1.py*. If you run the program, it won't do anything new yet, but this code creates the function for drawing the air and energy bars.

*listing12-1.py*

---

```

--snip--
objects[21][0] = frames[airlock_door_frame]
objects[21][1] = shadow_frames[airlock_door_frame]

#####

AIR

#####

def draw_energy_air():
 box = Rect((20, 765), (350, 20))
 ❶ screen.draw.filled_rect(box, BLACK) # Clear air bar.
 ❷ screen.draw.text("AIR", (20, 766), color=BLUE)
 ❸ screen.draw.text("ENERGY", (180, 766), color=YELLOW)

 ❹ if air > 0:
 ❺ box = Rect((50, 765), (air, 20))
 ❻ screen.draw.filled_rect(box, BLUE) # Draw new air bar.

 ❼ if energy > 0:
 box = Rect((250, 765), (energy, 20))
 screen.draw.filled_rect(box, YELLOW) # Draw new energy bar.

#####

START

#####

--snip--

```

---

### *Listing 12-1: Drawing the air and energy bars*

We begin the new `draw_energy_air()` function by drawing a black box over the status area at the bottom of the screen to clear it ❶. We then add the AIR label in blue ❷, and the ENERGY label in yellow ❸. This function will use the `air` and `energy` variables, which are already set to 100 in the `VARIABLES` part of the program.

If the player has some air left (if the variable `air` is more than 0) ❹, a box is created that uses the `air` variable for its width ❺. The box is then filled with the color blue ❻. This draws the AIR indicator bar, which starts off being 100 pixels wide and gets smaller as the `AIR` variable decreases.

We use similar instructions to draw the energy bar ❷, but the bar's start position is farther to the right (the *x* position is 250 instead of 50).

## ADDING THE AIR COUNTDOWN FUNCTIONS

We'll make three functions to enable the air countdown. The `end_the_game()` function runs when you're out of air. It displays the reason the player failed the mission, plays some sound effects, and shows a large GAME OVER message in the middle of the game window.

The `air_countdown()` function saps the air supply. We'll also add an `alarm()` function that runs shortly after the game begins to warn the player that their air is failing.

These three functions are in [Listing 12-2](#). Add the new code shown here in the `AIR` section of the program, after the `draw_energy_air()` function you just added. Save your program as *listing12-2.py*. You can run this program using `pgzrun listing12-2.py`, but you won't see anything new yet.

### *listing12-2.py*

---

--snip--

```
if energy > 0:
 box = Rect((250, 765), (energy, 20))
 screen.draw.filled_rect(box, YELLOW) # Draw new energy bar.
```

- ```
❶ def end_the_game(reason):
    global game_over
    ❷ show_text(reason, 1)
    ❸ game_over = True
    sounds.say_mission_fail.play()
    sounds.gameover.play()
    ❹ screen.draw.text("GAME OVER", (120, 400), color = "white",
        fontsize = 128, shadow = (1, 1), scolor = "black")

    ❺ def air_countdown():
        global air, game_over
        if game_over:
            return # Don't sap air when they're already dead.

    ❻ air -= 1
    ❼ if air == 20:
```



```

        sounds.say_air_low.play()
    if air == 10:
        sounds.say_act_now.play()
    ⑧ draw_energy_air()
    ⑨ if air < 1:
        end_the_game("You're out of air!")

10 def alarm():
    show_text("Air is running out, " + PLAYER_NAME
              + "! Get to safety, then radio for help!", 1)
    sounds.alarm.play(3)
    sounds.say_breach.play()

#####

##  START  ##

#####

--snip--

```

Listing 12-2: Adding the air countdown

The `air_countdown()` function ⑤ reduces the `air` variable's value by 1 each time it runs ⑥. If the value is equal to 20 ⑦ or 10, a warning sound effect plays to let the player know their air is low.

The `draw_energy_air()` function you added in [Listing 12-1](#) updates the air and energy display ⑧. If the air has run out ⑨, the `end_the_game()` function runs and displays the string "You're out of air!".

TIP

Sound files must be stored in the *sounds* folder and should be in *.wav* or *.ogg* format. To play a sound called *bang.wav*, you would use `sounds.bang.play()`. As with images, you don't need to tell Pygame Zero the file extension or where the sound is stored. Why not try recording and adding your own sound effects for various points in the game?

In the `end_the_game()` function ❶, we use the variable `reason` for the information it receives, and display that on the screen as the reason for death ❷. The `game_over` variable is set to `True` ❸. Other functions use this variable to know when the game has finished so everything can come to a stop. The `end_the_game()` function then draws the words **GAME OVER** in large text in the middle of the screen. The text is drawn at position $x = 120$, $y = 400$ in white text using a font size of 128 ❹. We also add a drop shadow under the text for effect, which is offset by 1 pixel in each direction and is colored black (see Figure 12-2).

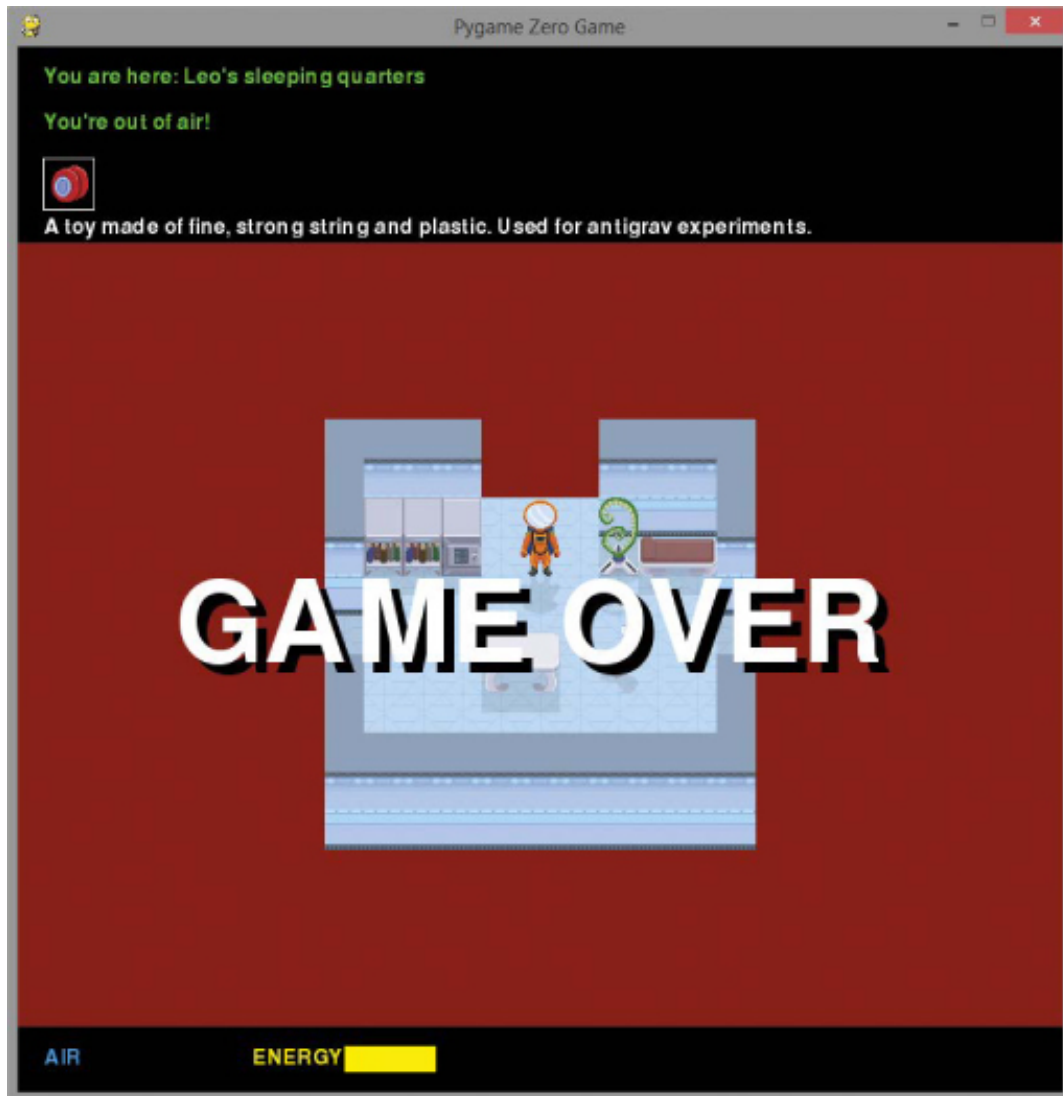


Figure 12-2: Oh no! You're out of air!

The final function in this section, `alarm()` ❿, plays the alarm sound and displays a message telling you to radio for help. It uses the player's name in the warning to personalize it.

The number in parentheses in the `sounds.alarm.play()` command is the number of times the sound should be played (in Listing 12-2, it's three).

STARTING THE AIR COUNTDOWN AND SOUNDING THE ALARM

We haven't set the three new functions to run yet. To do that, we need to add some instructions to the `START` section of the program, which is (perhaps confusingly!) at the end of the program listing. Add the new instructions shown in [Listing 12-3](#), and save it as *listing12-3.py*.

listing12-3.py

```
--snip--
#####

##  START  ##
#####

clock.schedule_interval(game_loop, 0.03)
generate_map()
clock.schedule_interval(adjust_wall_transparency, 0.05)
clock.schedule_unique(display_inventory, 1)
clock.schedule_unique(draw_energy_air, 0.5)
clock.schedule_unique(alarm, 10)
# A higher number below gives a longer time limit.
clock.schedule_interval(air_countdown, 5)
```

Listing 12-3: Starting the air countdown

Now the game has a time limit. When the air runs out, the game ends. Run the program using `pgzrun listing12-3.py`, and you should see your air supply slowly go down.

If you find the game too difficult when you're playing the final version, you can give yourself more time by changing the 5 in the final line in [Listing 12-3](#) to a higher number. This number decides how often the `air_countdown()` function saps your air supply, and is measured in seconds. In particular, if you're using a Raspberry Pi 2, the time limit might be challenging because the game runs a bit more slowly there. It's still possible to complete the game, but you can increase the number 5 to give yourself a little more, ahem, breathing space.

TRAINING MISSION #1

When your air supply reaches 0, you should see the GAME OVER message

and find that you can no longer move the astronaut. Your energy goes down by 1 percent every 5 seconds, so it'll take about 8.5 minutes (500 seconds) to run out. Can you work out how to make the air leak more often, so you can more easily test what happens when the air runs out?

After completing the training mission, make sure you change the program back again: otherwise, you'll find your mission rather hard to complete!

ADDING THE MOVING HAZARDS

There are three types of moving hazards in the game: two types of energy balls and a flying drone that's gone rogue.

Figure 12-3 shows the direction numbers the moving hazards use.

Hazards move in a straight line until they hit something, and then we add a number to change their direction. The number we add will decide the hazard's movement pattern. For example, if we add 1 to the direction number, the hazard moves in a clockwise pattern (up, right, down, left). If we add -1 to the direction number, the hazard moves in a counterclockwise pattern (left, down, right, up). If we add 2, it will bounce between going left and right (2 and 4) or up and down (1 and 3). Take a look at Figure 12-3 and check this makes sense to you. Each hazard can have its own pattern of movement.

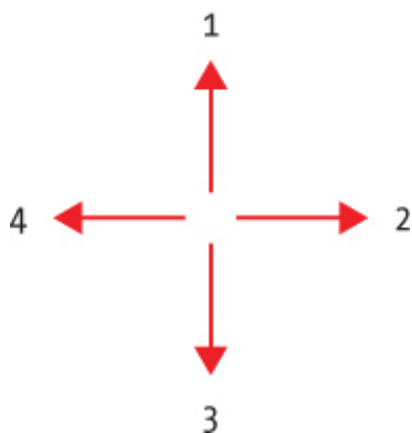






Figure 12-3: The direction numbers the moving hazards use are numbered in a clockwise order.

If the addition results in a number that's higher than 4, we subtract 4. For example, if a hazard is moving clockwise, we add 1 to its direction number each time it hits something. If it's going down (direction 3), we add 1 when it hits something, so it starts to move left (direction 4). The next time it hits something, we add 1, but that makes the direction number 5. So we subtract 4, and that gives us a direction number of 1. As

Figure 12-3 shows, that's the next direction number after 4, going around in a clockwise pattern.

Table 12-1 summarizes the numbers we can use to get different movement patterns.

Table 12-1: How to Change Direction When a Hazard Hits Something

| Movement pattern | | Number to add to the direction number |
|------------------|---|---------------------------------------|
| Clockwise |  | 1 |
| Counterclockwise |  | -1 |
| Left/right |  | 2 |
| Up/down |  | 2 |

RED ALERT

Take care not to mix up the two numbers that describe the movement. The direction number (see Figure 12-3) tells the program which direction a hazard is moving in. The number we add to the direction number (see Table 12-1) tells the program which way the hazard should bounce when it hits something.

ADDING THE HAZARD DATA

Between the `AIR` and `START` sections, we'll add a new section to the program called `HAZARDS`. Listing 12-4 shows you the hazard data. Add it to your program, and save it as *listing12-4.py*. If you run the program, it won't do anything new yet, but you can check that you don't get any error messages in the command line window.

listing12-4.py

```
--snip--
sounds.alarm.play(3)
sounds.say_breach.play()

#####

## HAZARDS ##

#####

hazard_data = {
    # room number: [[y, x, direction, bounce addition to direction]]
    ① 28: [[1, 8, 2, 1], [7, 3, 4, 1]], 32: [[1, 5, 1, 1]],
    34: [[5, 1, 1, 1], [5, 5, 1, 2]], 35: [[4, 4, 1, 2], [2, 5, 2, 2]],
    36: [[2, 1, 2, 2]], 38: [[1, 4, 3, 2], [5, 8, 1, 2]],
    40: [[3, 1, 3, 1], [6, 5, 2, 2], [7, 5, 4, 2]],
    41: [[4, 5, 2, 2], [6, 3, 4, 2], [8, 1, 2, 2]],
    42: [[2, 1, 2, 2], [4, 3, 2, 2], [6, 5, 2, 2]],
    46: [[2, 1, 2, 2]],
    48: [[1, 8, 3, 2], [8, 8, 1, 2], [3, 9, 3, 2]]
}

#####

## START ##

#####

--snip--
```

Listing 12-4: Adding the hazard data

We create a `hazard_data` dictionary that uses room numbers as dictionary keys. For each

room, there is a list that contains the data for all the hazards. The data for each hazard is in a list that contains the hazard's y position, x position, starting direction, and number to add when it hits something.

For example, room 28 ❶ has a hazard with the list data [7, 3, 4, 1]. This means the hazard starts at $y = 7$, $x = 3$. It starts moving left (direction 4), and it moves clockwise when it hits something because we add 1 to its direction number.

Room 41 contains three hazards (in three lists), which are moving from left to right and back again. We know that because they start with a direction of 2 or 4 (right or left) and add 2 to the direction when they hit something (making 4 or 6: we know that 6 becomes 2 after we subtract 4).

SAPPING THE PLAYER'S ENERGY

After the hazard data, we need to add a function called `deplete_energy()`, which reduces the player's energy when a hazard hits them. [Listing 12-5](#) shows the new function. Add it after [Listing 12-4](#) in the `HAZARDS` section of the program, and save your program as `listing12-5.py`. You can run the program to check for errors using `pgzrun listing12-5.py`, but it won't do anything new.

listing12-5.py

`--snip--`

```
46: [[2, 1, 2, 2]],
48: [[1, 8, 3, 2], [8, 8, 1, 2], [3, 9, 3, 2]]
}
```

```
❶ def deplete_energy(penalty):
    global energy, game_over
    if game_over:
        return # Don't sap energy when they're already dead.
❷ energy = energy - penalty
    draw_energy_air()
    if energy < 1:
        end_the_game("You're out of energy!")
```

```
#####
```

```
## START ##
```

```
#####
```

```
--snip--
```

Listing 12-5: Reducing the player's energy

The `deplete_energy()` function accepts a number ❶ and uses that number to reduce the player's energy variable ❷. As a result, we can use this function for hazards that drain different amounts of energy.

STARTING AND STOPPING HAZARDS

When the player enters a new room, the function `hazard_start()` puts the hazards into the room. Listing 12-6 shows this function, which you need to add after the `deplete_energy()` function in the `HAZARDS` section of the program. Save your program as *listing12-6.py*. If you run it using `pgzrun listing12-6.py`, you shouldn't notice any difference yet, because we haven't set this function to run.

listing12-6.py

```
--snip--
```

```
    if energy < 1:
        end_the_game("You're out of energy!")
```

```
def hazard_start():
```

```
    global current_room_hazards_list, hazard_map
```

```
❶ if current_room in hazard_data.keys():
```

```
❷     current_room_hazards_list = hazard_data[current_room]
```

```
❸     for hazard in current_room_hazards_list:
```

```
         hazard_y = hazard[0]
```

```
         hazard_x = hazard[1]
```

```
❹         hazard_map[hazard_y][hazard_x] = 49 + (current_room % 3)
```

```
❺     clock.schedule_interval(hazard_move, 0.15)
```

```
#####
```

```
## START ##
```

```
#####
```

```
--snip--
```

Listing 12-6: Adding the hazards to the current room

The `hazard_start()` function will run whenever the player enters a new room, so it begins by checking whether the current room has an entry in the `hazard_data` dictionary ❶. If it does, that room should have moving hazards in it, and the rest of the function runs. We put the hazard data for the room into a list called `current_room_hazards_list` ❷. The function then uses a loop ❸ to process each hazard in the list in turn.

The hazards use their own room map called `hazard_map`, so they can easily fly over objects on the floor without overwriting them in the room map. If the hazards used the same room map as the props, they would wipe out props as they flew over them, or we'd need a complicated way to remember what's underneath the hazards.

The three hazard objects have the numbers 49, 50, and 51 in the `objects` dictionary. The program uses a simple calculation to work out which one goes into a particular room. As you've seen before, Python's `%` operator gives you the remainder after doing a division. When you divide any number by 3, the remainder will be 0, 1, or 2. So the program divides the room number by 3 and adds the remainder to 49 to pick an object number ❹. So, for example, if we were in room 34, the program would work out that $34 \% 3$ is 1, and add 1 to 49 to select hazard number 50 for all the hazards in that room.

This way of selecting hazard numbers ensures the hazard is always the same type when the player enters the room. Because the map is five rooms wide, it also guarantees that two directly connected rooms cannot have the same hazard in them. That adds a sense of variety to the map, although not all rooms have hazards, so in practice, players might still encounter the same hazard twice in a row, walking through some empty rooms in between.

The function finishes by scheduling the `hazard_move()` function to run every 0.15 seconds ❺.

To start the `hazard_start()` function when the player enters a new room, add an instruction to the `start_room()` function, as shown in [Listing 12-7](#). Save your program as *listing12-7.py*. This version of the program will freeze when you leave the start room, because we haven't finished adding the code for the hazards yet.

listing12-7.py

```
--snip--
```

```
#####
```

```
## GAME LOOP ##
#####

def start_room():
    global airlock_door_frame
    show_text("You are here: " + room_name, 0)
    if current_room == 26: # Room with self-shutting airlock door
        airlock_door_frame = 0
        clock.schedule_interval(door_in_room_26, 0.05)
    hazard_start()
--snip--
```

Listing 12-7: Starting hazards when the player enters the room

Not all rooms have hazards, so we will stop the hazards from moving when the player leaves a room. We previously added instructions in the `game_loop()` function to turn off the function that makes the hazards move when the player changes room. We commented them out because we weren't ready for them yet.

We're ready for them now! Follow these steps to uncomment the instructions (you did something similar in [Chapter 8](#)):

1. Click **Edit ▸ Replace** (or press CTRL-H) in IDLE to show the Replace Text dialog box.
2. Type `#clock.unschedule(hazard_move)` into the Find box.
3. Type `clock.unschedule(hazard_move)` into the Replace With box.
4. Click **Replace All**. IDLE should replace the instruction in four places, and jump to the last one in the listing. [Listing 12-8](#) shows the new line that will be highlighted at the end of the process (you don't need to type this listing in). Above this block of code, there are three similar blocks that also now stop the hazards moving when the player leaves the room through one of the exits.

listing12-8.py

```
--snip--
if player_y == -1: # through door at TOP
    clock.unschedule(hazard_move)
```

```
current_room -= MAP_WIDTH
generate_map()
player_y = room_height - 1 # enter at bottom
player_x = int(room_width / 2) # enter at door
player_frame = 0
start_room()
return
--snip--
```

Listing 12-8: Stopping hazards when the player leaves the room

Save your updated program as *listing12-8.py*. If you run this version of the program, you'll see an error message in the console, and the game will freeze when you leave the room. The reason is that we haven't added the `hazard_move()` function yet.

SETTING UP THE HAZARD MAP

We now need to make sure that when the room map is generated for scenery and props, an empty hazard map is also generated. The `hazard_start()` function will fill it with any hazards in the room.

Add the new code shown in [Listing 12-9](#) at the end of the `generate_map()` function in the `MAKE MAP` section of the program. Place this new code just before the `GAME LOOP` section, and make sure you indent the first line by four spaces because it's inside a function.

Save your program as *listing12-9.py*. When you run it, the program won't work properly yet because it is still incomplete.

listing12-9.py

```
--snip--
    for tile_number in range(1, image_width_in_tiles):
        room_map[prop_y][prop_x + tile_number] = 255

    hazard_map = [] # empty list
    for y in range(room_height):
        hazard_map.append( [0] * room_width )

#####
## GAME LOOP ##
```

```
#####
```

```
--snip--
```

Listing 12-9: Creating the empty hazard map

These new instructions create an empty list for the hazard map and fill it with rows of 0s that are as wide as the room width.

MAKING THE HAZARDS MOVE

Now let's add the missing `hazard_move()` function to make the hazards move. Put this at the end of the `HAZARDS` section of the program after the `hazard_start()` function, as shown in [Listing 12-10](#). Save your program as *listing12-10.py*.

listing12-10.py

```
--snip--
```

```
    hazard_map[hazard_y][hazard_x] = 49 + (current_room % 3)
    clock.schedule_interval(hazard_move, 0.15)
```

```
def hazard_move():
```

```
    global current_room_hazards_list, hazard_data, hazard_map
```

```
    global old_player_x, old_player_y
```

```
    if game_over:
```

```
        return
```

```
    for hazard in current_room_hazards_list:
```

```
        hazard_y = hazard[0]
```

```
        hazard_x = hazard[1]
```

```
        hazard_direction = hazard[2]
```

```
❶    old_hazard_x = hazard_x
    old_hazard_y = hazard_y
    hazard_map[old_hazard_y][old_hazard_x] = 0
```

```
❷    if hazard_direction == 1: # up
        hazard_y -= 1
    if hazard_direction == 2: # right
        hazard_x += 1
    if hazard_direction == 3: # down
```

```
    hazard_y += 1
    if hazard_direction == 4: # left
        hazard_x -= 1
```

```
hazard_should_bounce = False
```

```
③    if (hazard_y == player_y and hazard_x == player_x) or \
        (hazard_y == from_player_y and hazard_x == from_player_x
         and player_frame > 0):
        sounds.ouch.play()
        deplete_energy(10)
        hazard_should_bounce = True
```

```
④    # Stop hazard going out of the doors
```

```
    if hazard_x == room_width:
        hazard_should_bounce = True
        hazard_x = room_width - 1
```

```
    if hazard_x == -1:
        hazard_should_bounce = True
        hazard_x = 0
```

```
    if hazard_y == room_height:
        hazard_should_bounce = True
        hazard_y = room_height - 1
```

```
    if hazard_y == -1:
        hazard_should_bounce = True
        hazard_y = 0
```

```
⑤    # Stop when hazard hits scenery or another hazard.
```

```
    if room_map[hazard_y][hazard_x] not in items_player_may_stand_on \
        or hazard_map[hazard_y][hazard_x] != 0:
        hazard_should_bounce = True
```

```
⑥    if hazard_should_bounce:
        hazard_y = old_hazard_y # Move back to last valid position.
        hazard_x = old_hazard_x
```

```
⑦    hazard_direction += hazard[3]
```

```
⑧    if hazard_direction > 4:
        hazard_direction -= 4
```

```
    if hazard_direction < 1:
```

```

        hazard_direction += 4
    9      hazard[2] = hazard_direction
    10     hazard_map[hazard_y][hazard_x] = 49 + (current_room % 3)
        hazard[0] = hazard_y
        hazard[1] = hazard_x

#####

##  START  ##

#####

--snip--

```

Listing 12-10: Adding the hazard movement function

The `hazard_move()` function uses an idea similar to the player movement. The hazard's position is stored in the `old_hazard_x` and `old_hazard_y` variables ❶. The hazard is then moved ❷.

Then we check whether the hazard has hit the player ❸, gone out the door ❹, or hit the scenery or another hazard ❺. If it has ❻, then its position is reset to its old values, and we change its direction by adding the last number in its list of data to the direction number ❼. If adding this number increases the direction number to more than 4 ❽, the function subtracts 4, as we discussed earlier in this chapter, because 4 is the highest valid direction number. On the other hand, if adding this number decreases the direction number to less than 1, the function adds 4. Finally, the new direction is saved in the hazard data ❾.

At the end of the function ❿, the hazard is put into the hazard map.

You can run this program using `pgzrun listing12-10.py`. The first room with a hazard is the one to the right of your starting room. When you enter it, your energy will mysteriously go down sometimes, even though you can't see anything dangerous. This is because we haven't added code to draw the hazards yet.

TIP

When a hazard hits you ❸, the `deplete_energy()` function reduces your energy by 10 percent. If you find the game too difficult, you can change this number to 5. If you complete the game and want a tougher challenge the next time

around, you could change it to 20!

DISPLAYING HAZARDS IN THE ROOM

It doesn't seem fair to have invisible dangers, so let's add a few lines to show the hazards in the room. [Listing 12-11](#) shows three new lines to add to the `draw()` function in the `DISPLAY` section of the program. Put these near the end of the function, before the code to draw the player.

Indent these instructions by a total of 12 spaces because they're inside the `draw()` function (4 spaces), inside the `y` loop (another 4), and inside the `x` loop (another 4-space indentation). Save your program as *listing12-11.py*.

listing12-11.py

```
--snip--
        # Use shadow across width of object.
        for z in range(0, shadow_width):
            draw_shadow(shadow_image, y, x+z)
        else:
            draw_shadow(shadow_image, y, x)

        hazard_here = hazard_map[y][x]
        if hazard_here != 0: # If there's a hazard at this position
            draw_image(objects[hazard_here][0], y, x)

    if (player_y == y):
        draw_player()
--snip--
```

Listing 12-11: Displaying the moving hazards

This listing completes the moving hazards. Run your program using `pgzrun listing12-11.py`. Then start running for your life! You should now be able to see the moving hazards, such as the energy ball shown in [Figure 12-4](#).

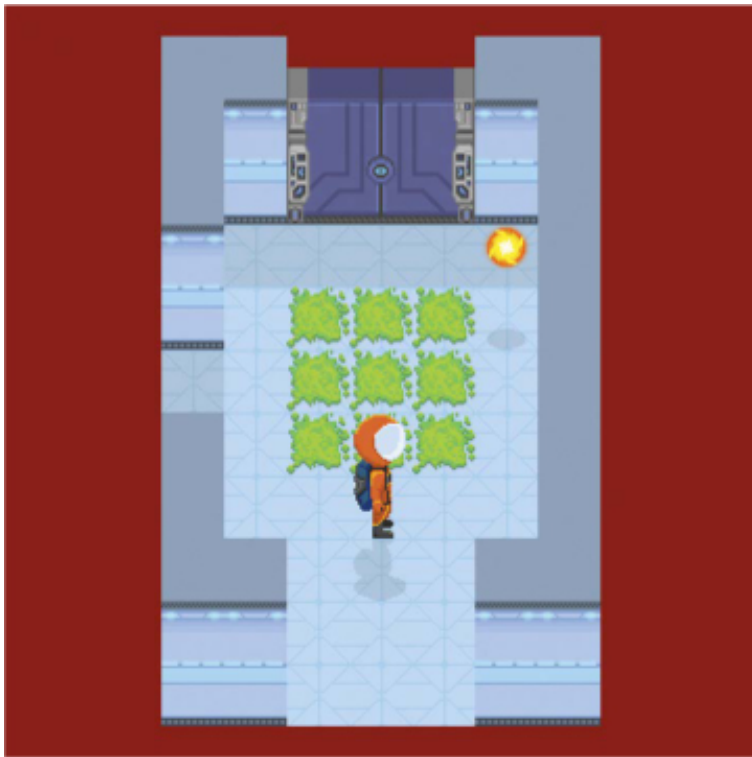


Figure 12-4: This energy ball bounces around the room in a counterclockwise pattern.

TRAINING MISSION #2

Test that the moving hazards work correctly. Enter the room to the right of your start room (or teleport into room 32 if necessary). When the energy ball hits you, does your energy decrease? Does the energy ball bounce off you? Can you bounce the energy ball into both doorways to check that it stays in the room? Does the game end when you run out of energy?

STOPPING THE PLAYER FROM WALKING THROUGH HAZARDS

We also need to add a line to stop the player from walking into or through hazards. In practice, the hazard will usually bounce off the player, but without making the fix shown in [Listing 12-12](#), it's sometimes possible to pass through the hazard.

We already added the code we need in the `game_loop()` function, but we commented it out. Now it's time to uncomment it by deleting the `#` symbol before the `\` at the end of one line, and removing the `#` at the start of the next line.

We also need to delete the colon after `items_player_may_stand_on`. A quick way to find the right part of the program is to press CTRL-F to open the search box, and then enter `#\`. [Listing 12-12](#) shows you the lines to modify.


```
--snip--
# If the player is standing somewhere they shouldn't, move them back.
if room_map[player_y][player_x] not in items_player_may_stand_on \
    or hazard_map[player_y][player_x] != 0:
    player_x = old_player_x
    player_y = old_player_y
    player_frame = 0
--snip--
```

Listing 12-12: Stopping the player from passing through hazards

Save your program as *listing12-12.py* and run it with `pgzrun listing12-12.py`. Can you track down all three types of flying hazards in the space station?

ADDING THE TOXIC SPILLS

You might have noticed the green splash on the floor in [Figure 12-4](#). It's a toxic spill, and it saps your energy when you walk on it. You'll have to think strategically. Should you run through it to get somewhere faster? Or should you walk carefully around it, saving your energy for later but maybe slowing you down?

[Listing 12-13](#) shows the instructions to add to sap your energy when you're walking on the toxic floor. These instructions go in the `game_loop()` function, just after the instructions you fixed in [Listing 12-12](#).

Save your program as *listing12-13.py*. You can test that it works by running the program using `pgzrun listing12-13.py` and then walking on the toxic floor. The toxic floor is object 48 and is positioned as scenery in the room.

```
--snip--
# If the player is standing somewhere they shouldn't, move them back.
if room_map[player_y][player_x] not in items_player_may_stand_on \
    or hazard_map[player_y][player_x] != 0:
    player_x = old_player_x
    player_y = old_player_y
    player_frame = 0
```

```
if room_map[player_y][player_x] == 48: # toxic floor
    deplete_energy(1)

if player_direction == "right" and player_frame > 0:
    player_offset_x = -1 + (0.25 * player_frame)
--snip--
```

Listing 12-13: Reducing the player's energy when they walk on the toxic floor

MAKING THE FINISHING TOUCHES

The game is now nearly complete. Before you embark on your exploration of the space station, we need to remove some of the instructions we used while building and testing the game.

DISABLING THE TELEPORTER

Mission rules forbid the use of the teleporter once your work on the space station begins. Find its instructions in the `game_loop()` function, highlight them using your mouse, and then click **Format** ▶ **Comment Out Region** to disable them. Your code should now look like [Listing 12-14](#).

listing12-14.py

```
--snip--
#### Teleporter for testing
#### Remove this section for the real game
## if keyboard.x:
##     current_room = int(input("Enter room number:"))
##     player_x = 2
##     player_y = 2
##     generate_map()
##     start_room()
##     sounds.teleport.play()
#### Teleport section ends
--snip--
```

Listing 12-14: The teleporter is turned off.

CLEANING UP THE DATA

While testing the game, you might have changed the contents of some of the variables and lists. The game should look like [Figure 12-5](#) when it begins. If it doesn't, look at the `VARIABLES` section of the program and make sure the `current_room` variable is set to 31.

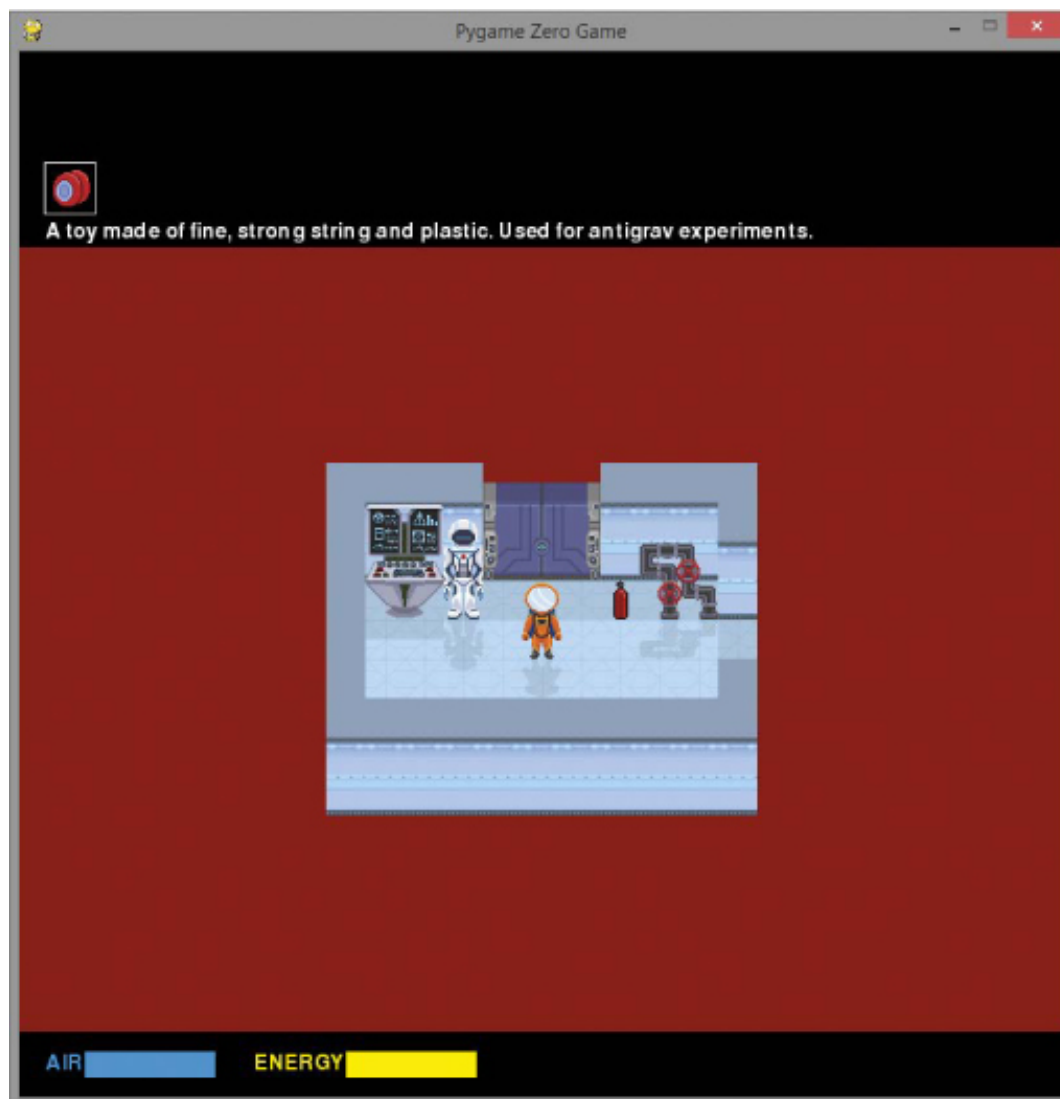


Figure 12-5: The start of your mission

If you're carrying more than your yoyo, look at the `PROPS` part of the program and check that this line is correct:

```
in_my_pockets = [55]
```

YOUR ADVENTURE BEGINS

It's an exciting moment: your training is complete; the space station is ready; and your mission on Mars is about to begin. Let's set a sci-fi fanfare to play when the game starts. [Listing 12-15](#) shows the final instruction you'll add to *Escape*.

```
--snip--
clock.schedule_unique(alarm, 10)
clock.schedule_interval(air_countdown, 11) # A higher number gives a longer
time limit.
sounds.mission.play() # Intro music
```

Listing 12-15: A sci-fi fanfare plays when the game begins.

Save your final program as *escape.py*. You can now play the game using `pgzrun escape.py`. See “[Playing the Game](#)” on [page 11](#) for instructions.

Congratulations on completing the space station construction. You’ve truly earned your place on this mission. It’s time to begin your work on the planet’s surface!

YOUR NEXT MISSION: CUSTOMIZING THE GAME

Did you make it to safety in the *Escape* game? That was a close shave! For your next mission, try customizing the game. There are different ways to use this book, so you might already have made some customizations as you built the game. Here are some suggestions for modifying the game, starting with the easiest:

- Change the names of the characters in the game to those of your friends. See [Listing 4-1 on page 63 in Chapter 4](#).
- Customize the images. You can edit our images, or create your own. The game includes a whiteboard image that you can edit using your favorite art package. If you make your images the same size as ours, use the same filenames, and store them in the *images* folder, they should just drop into the game world with no problem.
- Redesign the room layouts. [Chapter 6](#) explains how scenery is positioned in a room.
- Add your own objects to the game. Start by creating their images. Props should be 30 pixels square. Scenery items can be bigger and should touch the left and right sides of their tile spaces so that it doesn’t look odd when the player can’t get closer to the scenery than the tile next door. (For example, if your image is 30, 60, or 90 pixels wide and touches the ground at both sides, it should look fine.) You need to add the new items in the `objects` dictionary (see [Chapter 5](#)). For help positioning scenery, see [Chapter 6](#). For advice on positioning props, see [Chapter 9](#).

- Create your own space station map (see [Chapter 4](#)).
- Use the game engine to make your own game. You can replace the images and maps, and code your own puzzles to make a new game based on the *Escape* code. The `USE OBJECTS` section is where the game puzzles are programmed. It details what happens when objects are used, individually or in combination with other objects. It might be useful to keep the code for combining objects (recipes) and just update it (see [Chapter 10](#)); keep the code for displaying standard responses (see [Chapter 10](#)); and keep the code for opening doors (see [Chapter 11](#)).

If you make any changes that affect room 26, you'll need to disable the code for its pressure pad (see [Chapter 11](#)).

Bear in mind that any changes you make might break the puzzles in the original *Escape* game, making it impossible to complete. For example, it might become impossible to find important tools. I recommend saving any changes you make separately, so you can always come back to the original code.

SHARING YOUR CUSTOMIZATIONS

I'd love to hear about your customizations! You can find me on Twitter at [@musicandwords](#) or visit my website at www.sean.co.uk, which includes bonus content for the book. If you share your modified *Escape* game with others or share your own games built using its code, sounds, or images, please credit this book and its author, and make it clear that you've modified the code. Thank you!

ARE YOU FIT TO FLY?

Check the following boxes to confirm that you've learned the key lessons in this chapter.

- ☐ You can use Pygame Zero to draw text with a shadow underneath it and can adjust the size of the text displayed.
- ☐ You can play a sound multiple times by putting the number of times in parentheses in its `sounds.sound_name.play()` instruction.
- ☐ The moving hazards' directions are numbered from 1 at the top, moving clockwise.

To create a movement pattern for a hazard, you provide the number you want to add to its direction number when it hits something.

- ☐ The `deplete_energy()` function reduces the player's energy.
- ☐ Hazards use their own room map called `hazard_map`. This enables them to more easily move over objects on the floor.
- ☐ Before playing the game, check that the starting variables are correct.

MISSION DEBRIEF

Here is the answer for the first training mission in this chapter.

TRAINING MISSION #1

In the final line of Listing 12-3, change the value 5 to 1. That will make the function that reduces the air run every second instead of every 5 seconds. You could make it go even faster by changing the value to 0.5 or another decimal number.

A

ESCAPE: THE COMPLETE GAME LISTING



This appendix shows the final listing for the *Escape* game. You can use it as a reference to see where to place particular functions and sections, or read through it if you want to see the whole listing in one place. This listing doesn't include the temporary sections you wrote while building the game, such as the `EXPLORER` section. It just contains the code that is in the final game.

Remember that you can also download the *escape.py* listing and read it in IDLE, which lets you search it by pressing CTRL-F.

I've changed `PLAYER_NAME` to "Captain" in this listing. When you're building or customizing the game, you can use your own name (see ❶ in Listing 4-1 on page 63).

To test this project, I rebuilt the game using the instructions in this book. This game listing has been copied from game code that has been tested to completion on Windows, the Raspberry Pi 3 Model B+, and the Raspberry Pi 2 Model B.

```
# Escape - A Python Adventure
# by Sean McManus / www.sean.co.uk
# Art by Rafael Pimenta
# Typed in by PUT YOUR NAME HERE
```

```
import time, random, math
```

```
#####
```

```
## VARIABLES ##
```

```
#####
```

```
WIDTH = 800 #window size
```

```
HEIGHT = 800
```

```
#PLAYER variables
```

```
PLAYER_NAME = "Captain" # change this to your name!
```

```
FRIEND1_NAME = "Karen" # change this to a friend's name!
```

```
FRIEND2_NAME = "Leo" # change this to another friend's name!
```

```
current_room = 31 # start room = 31
```

```
top_left_x = 100
```

```
top_left_y = 150
```

```
DEMO_OBJECTS = [images.floor, images.pillar, images.soil]
```

```
LANDER_SECTOR = random.randint(1, 24)
```

```
LANDER_X = random.randint(2, 11)
```

```
LANDER_Y = random.randint(2, 11)
```

```
TILE_SIZE = 30
```

```
player_y, player_x = 2, 5
```

```
game_over = False
```

```
PLAYER = {
```

```
    "left": [images.spacesuit_left, images.spacesuit_left_1,
             images.spacesuit_left_2, images.spacesuit_left_3,
             images.spacesuit_left_4
            ],
```

```
    "right": [images.spacesuit_right, images.spacesuit_right_1,
              images.spacesuit_right_2, images.spacesuit_right_3,
              images.spacesuit_right_4
             ],
```



```

"up": [images.spacesuit_back, images.spacesuit_back_1,
        images.spacesuit_back_2, images.spacesuit_back_3,
        images.spacesuit_back_4
    ],
"down": [images.spacesuit_front, images.spacesuit_front_1,
          images.spacesuit_front_2, images.spacesuit_front_3,
          images.spacesuit_front_4
    ]
}

```

```

player_direction = "down"
player_frame = 0
player_image = PLAYER[player_direction][player_frame]
player_offset_x, player_offset_y = 0, 0

```

```

PLAYER_SHADOW = {
    "left": [images.spacesuit_left_shadow, images.spacesuit_left_1_shadow,
             images.spacesuit_left_2_shadow, images.spacesuit_left_3_shadow,
             images.spacesuit_left_3_shadow
    ],
    "right": [images.spacesuit_right_shadow, images.spacesuit_right_1_shadow,
              images.spacesuit_right_2_shadow,
              images.spacesuit_right_3_shadow, images.spacesuit_right_3_shadow
    ],
    "up": [images.spacesuit_back_shadow, images.spacesuit_back_1_shadow,
           images.spacesuit_back_2_shadow, images.spacesuit_back_3_shadow,
           images.spacesuit_back_3_shadow
    ],
    "down": [images.spacesuit_front_shadow, images.spacesuit_front_1_shadow,
             images.spacesuit_front_2_shadow, images.spacesuit_front_3_shadow,
             images.spacesuit_front_3_shadow
    ]
}

```

```

player_image_shadow = PLAYER_SHADOW["down"][0]

```

```

PILLARS = [
    images.pillar, images.pillar_95, images.pillar_80,
    images.pillar_60, images.pillar_50
]

```

```
]
```

```
wall_transparency_frame = 0
```

```
BLACK = (0, 0, 0)
```

```
BLUE = (0, 155, 255)
```

```
YELLOW = (255, 255, 0)
```

```
WHITE = (255, 255, 255)
```

```
GREEN = (0, 255, 0)
```

```
RED = (128, 0, 0)
```

```
air, energy = 100, 100
```

```
suit_stitched, air_fixed = False, False
```

```
launch_frame = 0
```

```
#####
```

```
##  MAP  ##
```

```
#####
```

```
MAP_WIDTH = 5
```

```
MAP_HEIGHT = 10
```

```
MAP_SIZE = MAP_WIDTH * MAP_HEIGHT
```

```
GAME_MAP = [ ["Room 0 - where unused objects are kept", 0, 0, False, False] ]
```

```
outdoor_rooms = range(1, 26)
```

```
for planetsectors in range(1, 26): #rooms 1 to 25 are generated here
```

```
    GAME_MAP.append( ["The dusty planet surface", 13, 13, True, True] )
```

```
GAME_MAP += [
```

```
    #["Room name", height, width, Top exit?, Right exit?]
```

```
    ["The airlock", 13, 5, True, False], # room 26
```

```
    ["The engineering lab", 13, 13, False, False], # room 27
```

```
    ["Poodle Mission Control", 9, 13, False, True], # room 28
```

```
    ["The viewing gallery", 9, 15, False, False], # room 29
```

```
    ["The crew's bathroom", 5, 5, False, False], # room 30
```

```
    ["The airlock entry bay", 7, 11, True, True], # room 31
```

```
    ["Left elbow room", 9, 7, True, False], # room 32
```

```

["Right elbow room", 7, 13, True, True], # room 33
["The science lab", 13, 13, False, True], # room 34
["The greenhouse", 13, 13, True, False], # room 35
[PLAYER_NAME + "'s sleeping quarters", 9, 11, False, False], # room 36
["West corridor", 15, 5, True, True], # room 37
["The briefing room", 7, 13, False, True], # room 38
["The crew's community room", 11, 13, True, False], # room 39
["Main Mission Control", 14, 14, False, False], # room 40
["The sick bay", 12, 7, True, False], # room 41
["West corridor", 9, 7, True, False], # room 42
["Utilities control room", 9, 9, False, True], # room 43
["Systems engineering bay", 9, 11, False, False], # room 44
["Security portal to Mission Control", 7, 7, True, False], # room 45
[FRIEND1_NAME + "'s sleeping quarters", 9, 11, True, True], # room 46
[FRIEND2_NAME + "'s sleeping quarters", 9, 11, True, True], # room 47
["The pipeworks", 13, 11, True, False], # room 48
["The chief scientist's office", 9, 7, True, True], # room 49
["The robot workshop", 9, 11, True, False] # room 50
]

```

#simple sanity check on map above to check data entry

```
assert len(GAME_MAP)-1 == MAP_SIZE, "Map size and GAME_MAP don't match"
```

```
#####
```

```
## OBJECTS ##
```

```
#####
```

```

objects = {
    0: [images.floor, None, "The floor is shiny and clean"],
    1: [images.pillar, images.full_shadow, "The wall is smooth and cold"],
    2: [images.soil, None, "It's like a desert. Or should that be dessert?"],
    3: [images.pillar_low, images.half_shadow, "The wall is smooth and cold"],
    4: [images.bed, images.half_shadow, "A tidy and comfortable bed"],
    5: [images.table, images.half_shadow, "It's made from strong plastic."],
    6: [images.chair_left, None, "A chair with a soft cushion"],
    7: [images.chair_right, None, "A chair with a soft cushion"],
    8: [images.bookcase_tall, images.full_shadow,
        "Bookshelves, stacked with reference books"],

```

9: [images.bookcase_small, images.half_shadow,
"Bookshelves, stacked with reference books"],

10: [images.cabinet, images.half_shadow,
"A small locker, for storing personal items"],

11: [images.desk_computer, images.half_shadow,
"A computer. Use it to run life support diagnostics"],

12: [images.plant, images.plant_shadow, "A spaceberry plant, grown here"],

13: [images.electrical1, images.half_shadow,
"Electrical systems used for powering the space station"],

14: [images.electrical2, images.half_shadow,
"Electrical systems used for powering the space station"],

15: [images.cactus, images.cactus_shadow, "Ouch! Careful on the cactus!"],

16: [images.shrub, images.shrub_shadow,
"A space lettuce. A bit limp, but amazing it's growing here!"],

17: [images.pipes1, images.pipes1_shadow, "Water purification pipes"],

18: [images.pipes2, images.pipes2_shadow,
"Pipes for the life support systems"],

19: [images.pipes3, images.pipes3_shadow,
"Pipes for the life support systems"],

20: [images.door, images.door_shadow, "Safety door. Opens automatically \\
for astronauts in functioning spacesuits."],

21: [images.door, images.door_shadow, "The airlock door. \\
For safety reasons, it requires two person operation."],

22: [images.door, images.door_shadow, "A locked door. It needs " \\
+ PLAYER_NAME + "'s access card"],

23: [images.door, images.door_shadow, "A locked door. It needs " \\
+ FRIEND1_NAME + "'s access card"],

24: [images.door, images.door_shadow, "A locked door. It needs " \\
+ FRIEND2_NAME + "'s access card"],

25: [images.door, images.door_shadow,
"A locked door. It is opened from Main Mission Control"],

26: [images.door, images.door_shadow,
"A locked door in the engineering bay."],

27: [images.map, images.full_shadow,
"The screen says the crash site was Sector: " \\
+ str(LANDER_SECTOR) + " // X: " + str(LANDER_X) + \\
" // Y: " + str(LANDER_Y)],

28: [images.rock_large, images.rock_large_shadow,
"A rock. Its coarse surface feels like a whetstone", "the rock"],

29: [images.rock_small, images.rock_small_shadow,
"A small but heavy piece of Martian rock"],

30: [images.crater, None, "A crater in the planet surface"],

31: [images.fence, None,
"A fine gauze fence. It helps protect the station from dust storms"],

32: [images.contraption, images.contraption_shadow,
"One of the scientific experiments. It gently vibrates"],

33: [images.robot_arm, images.robot_arm_shadow,
"A robot arm, used for heavy lifting"],

34: [images.toilet, images.half_shadow, "A sparkling clean toilet"],

35: [images.sink, None, "A sink with running water", "the taps"],

36: [images.globe, images.globe_shadow,
"A giant globe of the planet. It gently glows from inside"],

37: [images.science_lab_table, None,
"A table of experiments, analyzing the planet soil and dust"],

38: [images.vending_machine, images.full_shadow,
"A vending machine. It requires a credit.", "the vending machine"],

39: [images.floor_pad, None,
"A pressure sensor to make sure nobody goes out alone."],

40: [images.rescue_ship, images.rescue_ship_shadow, "A rescue ship!"],

41: [images.mission_control_desk, images.mission_control_desk_shadow, \
"Mission Control stations."],

42: [images.button, images.button_shadow,
"The button for opening the time-locked door in engineering."],

43: [images.whiteboard, images.full_shadow,
"The whiteboard is used in brainstorming and planning meetings."],

44: [images.window, images.full_shadow,
"The window provides a view out onto the planet surface."],

45: [images.robot, images.robot_shadow, "A cleaning robot, turned off."],

46: [images.robot2, images.robot2_shadow,
"A planet surface exploration robot, awaiting set-up."],

47: [images.rocket, images.rocket_shadow, "A 1-person craft in repair."],

48: [images.toxic_floor, None, "Toxic floor - do not walk on!"],

49: [images.drone, None, "A delivery drone"],

50: [images.energy_ball, None, "An energy ball - dangerous!"],

51: [images.energy_ball2, None, "An energy ball - dangerous!"],

52: [images.computer, images.computer_shadow,
"A computer workstation, for managing space station systems."],

53: [images.clipboard, None,

"A clipboard. Someone has doodled on it.", "the clipboard"],
54: [images.bubble_gum, None,
"A piece of sticky bubble gum. Spaceberry flavour.", "bubble gum"],
55: [images.yoyo, None, "A toy made of fine, strong string and plastic. \
Used for antigrav experiments.", PLAYER_NAME + "'s yoyo"],
56: [images.thread, None,
"A piece of fine, strong string", "a piece of string"],
57: [images.needle, None,
"A sharp needle from a cactus plant", "a cactus needle"],
58: [images.threaded_needle, None,
"A cactus needle, spearing a length of string", "needle and string"],
59: [images.canister, None,
"The air canister has a leak.", "a leaky air canister"],
60: [images.canister, None,
"It looks like the seal will hold!", "a sealed air canister"],
61: [images.mirror, None,
"The mirror throws a circle of light on the walls.", "a mirror"],
62: [images.bin_empty, None,
"A rarely used bin, made of light plastic", "a bin"],
63: [images.bin_full, None,
"A heavy bin full of water", "a bin full of water"],
64: [images.rags, None,
"An oily rag. Pick it up by a corner if you must!", "an oily rag"],
65: [images.hammer, None,
"A hammer. Maybe good for cracking things open...", "a hammer"],
66: [images.spoon, None, "A large serving spoon", "a spoon"],
67: [images.food_pouch, None,
"A dehydrated food pouch. It needs water.", "a dry food pack"],
68: [images.food, None,
"A food pouch. Use it to get 100% energy.", "ready-to-eat food"],
69: [images.book, None, "The book has the words 'Don't Panic' on the \
cover in large, friendly letters", "a book"],
70: [images.mp3_player, None,
"An MP3 player, with all the latest tunes", "an MP3 player"],
71: [images.lander, None, "The Poodle, a small space exploration craft. \
Its black box has a radio sealed inside.", "the Poodle lander"],
72: [images.radio, None, "A radio communications system, from the \
Poodle", "a communications radio"],
73: [images.gps_module, None, "A GPS Module", "a GPS module"],

```

74: [images.positioning_system, None, "Part of a positioning system. \
Needs a GPS module.", "a positioning interface"],
75: [images.positioning_system, None,
     "A working positioning system", "a positioning computer"],
76: [images.scissors, None, "Scissors. They're too blunt to cut \
anything. Can you sharpen them?", "blunt scissors"],
77: [images.scissors, None,
     "Razor-sharp scissors. Careful!", "sharpened scissors"],
78: [images.credit, None,
     "A small coin for the station's vending systems",
     "a station credit"],
79: [images.access_card, None,
     "This access card belongs to " + PLAYER_NAME, "an access card"],
80: [images.access_card, None,
     "This access card belongs to " + FRIEND1_NAME, "an access card"],
81: [images.access_card, None,
     "This access card belongs to " + FRIEND2_NAME, "an access card"]
}

```

```

items_player_may_carry = list(range(53, 82))

```

```

# Numbers below are for floor, pressure pad, soil, toxic floor.

```

```

items_player_may_stand_on = items_player_may_carry + [0, 39, 2, 48]

```

```

#####

```

```

## SCENERY ##

```

```

#####

```

```

# Scenery describes objects that cannot move between rooms.

```

```

# room number: [[object number, y position, x position]...]

```

```

scenery = {

```

```

    26: [[39,8,2]],

```

```

    27: [[33,5,5], [33,1,1], [33,1,8], [47,5,2],
         [47,3,10], [47,9,8], [42,1,6]],

```

```

    28: [[27,0,3], [41,4,3], [41,4,7]],

```

```

    29: [[7,2,6], [6,2,8], [12,1,13], [44,0,1],
         [36,4,10], [10,1,1], [19,4,2], [17,4,4]],

```

```

    30: [[34,1,1], [35,1,3]],

```

```

    31: [[11,1,1], [19,1,8], [46,1,3]],

```

```

32: [[48,2,2], [48,2,3], [48,2,4], [48,3,2], [48,3,3],
    [48,3,4], [48,4,2], [48,4,3], [48,4,4]],
33: [[13,1,1], [13,1,3], [13,1,8], [13,1,10], [48,2,1],
    [48,2,7], [48,3,6], [48,3,3]],
34: [[37,2,2], [32,6,7], [37,10,4], [28,5,3]],
35: [[16,2,9], [16,2,2], [16,3,3], [16,3,8], [16,8,9], [16,8,2], [16,1,8],
    [16,1,3], [12,8,6], [12,9,4], [12,9,8],
    [15,4,6], [12,7,1], [12,7,11]],
36: [[4,3,1], [9,1,7], [8,1,8], [8,1,9],
    [5,5,4], [6,5,7], [10,1,1], [12,1,2]],
37: [[48,3,1], [48,3,2], [48,7,1], [48,5,2], [48,5,3],
    [48,7,2], [48,9,2], [48,9,3], [48,11,1], [48,11,2]],
38: [[43,0,2], [6,2,2], [6,3,5], [6,4,7], [6,2,9], [45,1,10]],
39: [[38,1,1], [7,3,4], [7,6,4], [5,3,6], [5,6,6],
    [6,3,9], [6,6,9], [45,1,11], [12,1,8], [12,1,4]],
40: [[41,5,3], [41,5,7], [41,9,3], [41,9,7],
    [13,1,1], [13,1,3], [42,1,12]],
41: [[4,3,1], [10,3,5], [4,5,1], [10,5,5], [4,7,1],
    [10,7,5], [12,1,1], [12,1,5]],
44: [[46,4,3], [46,4,5], [18,1,1], [19,1,3],
    [19,1,5], [52,4,7], [14,1,8]],
45: [[48,2,1], [48,2,2], [48,3,3], [48,3,4], [48,1,4], [48,1,1]],
46: [[10,1,1], [4,1,2], [8,1,7], [9,1,8], [8,1,9], [5,4,3], [7,3,2]],
47: [[9,1,1], [9,1,2], [10,1,3], [12,1,7], [5,4,4], [6,4,7], [4,1,8]],
48: [[17,4,1], [17,4,2], [17,4,3], [17,4,4], [17,4,5], [17,4,6], [17,4,7],
    [17,8,1], [17,8,2], [17,8,3], [17,8,4],
    [17,8,5], [17,8,6], [17,8,7], [14,1,1]],
49: [[14,2,2], [14,2,4], [7,5,1], [5,5,3], [48,3,3], [48,3,4]],
50: [[45,4,8], [11,1,1], [13,1,8], [33,2,1], [46,4,6]]
}

```

```
checksum = 0
```

```
check_counter = 0
```

```
for key, room_scenery_list in scenery.items():
```

```
    for scenery_item_list in room_scenery_list:
```

```
        checksum += (scenery_item_list[0] * key
                     + scenery_item_list[1] * (key + 1)
                     + scenery_item_list[2] * (key + 2))
```

```
    check_counter += 1
```



```

print(check_counter, "scenery items")
assert check_counter == 161, "Expected 161 scenery items"
assert checksum == 200095, "Error in scenery data"
print("Scenery checksum: " + str(checksum))

for room in range(1, 26): # Add random scenery in planet locations.
    if room != 13: # Skip room 13.
        scenery_item = random.choice([16, 28, 29, 30])
        scenery[room] = [[scenery_item, random.randint(2, 10),
                           random.randint(2, 10)]]

# Use loops to add fences to the planet surface rooms.
for room_coordinate in range(0, 13):
    for room_number in [1, 2, 3, 4, 5]: # Add top fence
        scenery[room_number] += [[31, 0, room_coordinate]]
    for room_number in [1, 6, 11, 16, 21]: # Add left fence
        scenery[room_number] += [[31, room_coordinate, 0]]
    for room_number in [5, 10, 15, 20, 25]: # Add right fence
        scenery[room_number] += [[31, room_coordinate, 12]]

del scenery[21][-1] # Delete last fence panel in Room 21
del scenery[25][-1] # Delete last fence panel in Room 25

#####
## MAKE MAP ##
#####

def get_floor_type():
    if current_room in outdoor_rooms:
        return 2 # soil
    else:
        return 0 # tiled floor

def generate_map():
    # This function makes the map for the current room,
    # using room data, scenery data and prop data.
    global room_map, room_width, room_height, room_name, hazard_map
    global top_left_x, top_left_y, wall_transparency_frame

```

```

room_data = GAME_MAP[current_room]
room_name = room_data[0]
room_height = room_data[1]
room_width = room_data[2]

floor_type = get_floor_type()
if current_room in range(1, 21):
    bottom_edge = 2 #soil
    side_edge = 2 #soil
if current_room in range(21, 26):
    bottom_edge = 1 #wall
    side_edge = 2 #soil
if current_room > 25:
    bottom_edge = 1 #wall
    side_edge = 1 #wall

# Create top line of room map.
room_map=[[side_edge] * room_width]
# Add middle lines of room map (wall, floor to fill width, wall).
for y in range(room_height - 2):
    room_map.append([side_edge]
                    + [floor_type]*(room_width - 2) + [side_edge])
# Add bottom line of room map.
room_map.append([bottom_edge] * room_width)

# Add doorways.
middle_row = int(room_height / 2)
middle_column = int(room_width / 2)

if room_data[4]: # If exit at right of this room
    room_map[middle_row][room_width - 1] = floor_type
    room_map[middle_row+1][room_width - 1] = floor_type
    room_map[middle_row-1][room_width - 1] = floor_type

if current_room % MAP_WIDTH != 1: # If room is not on left of map
    room_to_left = GAME_MAP[current_room - 1]
    # If room on the left has a right exit, add left exit in this room
    if room_to_left[4]:
        room_map[middle_row][0] = floor_type

```

```

room_map[middle_row + 1][0] = floor_type
room_map[middle_row - 1][0] = floor_type

if room_data[3]: # If exit at top of this room
    room_map[0][middle_column] = floor_type
    room_map[0][middle_column + 1] = floor_type
    room_map[0][middle_column - 1] = floor_type

if current_room <= MAP_SIZE - MAP_WIDTH: # If room is not on bottom row
    room_below = GAME_MAP[current_room+MAP_WIDTH]
    # If room below has a top exit, add exit at bottom of this one
    if room_below[3]:
        room_map[room_height-1][middle_column] = floor_type
        room_map[room_height-1][middle_column + 1] = floor_type
        room_map[room_height-1][middle_column - 1] = floor_type

if current_room in scenery:
    for this_scenery in scenery[current_room]:
        scenery_number = this_scenery[0]
        scenery_y = this_scenery[1]
        scenery_x = this_scenery[2]
        room_map[scenery_y][scenery_x] = scenery_number

        image_here = objects[scenery_number][0]
        image_width = image_here.get_width()
        image_width_in_tiles = int(image_width / TILE_SIZE)

        for tile_number in range(1, image_width_in_tiles):
            room_map[scenery_y][scenery_x + tile_number] = 255

center_y = int(HEIGHT / 2) # Center of game window
center_x = int(WIDTH / 2)
room_pixel_width = room_width * TILE_SIZE # Size of room in pixels
room_pixel_height = room_height * TILE_SIZE
top_left_x = center_x - 0.5 * room_pixel_width
top_left_y = (center_y - 0.5 * room_pixel_height) + 110

for prop_number, prop_info in props.items():
    prop_room = prop_info[0]

```

```

prop_y = prop_info[1]
prop_x = prop_info[2]
if (prop_room == current_room and
    room_map[prop_y][prop_x] in [0, 39, 2]):
    room_map[prop_y][prop_x] = prop_number
    image_here = objects[prop_number][0]
    image_width = image_here.get_width()
    image_width_in_tiles = int(image_width / TILE_SIZE)
    for tile_number in range(1, image_width_in_tiles):
        room_map[prop_y][prop_x + tile_number] = 255

hazard_map = [] # empty list
for y in range(room_height):
    hazard_map.append( [0] * room_width )

#####
## GAME LOOP ##
#####

def start_room():
    global airlock_door_frame
    show_text("You are here: " + room_name, 0)
    if current_room == 26: # Room with self-shutting airlock door
        airlock_door_frame = 0
        clock.schedule_interval(door_in_room_26, 0.05)
    hazard_start()

def game_loop():
    global player_x, player_y, current_room
    global from_player_x, from_player_y
    global player_image, player_image_shadow
    global selected_item, item_carrying, energy
    global player_offset_x, player_offset_y
    global player_frame, player_direction

    if game_over:
        return

```

```
if player_frame > 0:
    player_frame += 1
    time.sleep(0.05)
if player_frame == 5:
    player_frame = 0
    player_offset_x = 0
    player_offset_y = 0
```

save player's current position

```
old_player_x = player_x
old_player_y = player_y
```

move if key is pressed

```
if player_frame == 0:
    if keyboard.right:
        from_player_x = player_x
        from_player_y = player_y
        player_x += 1
        player_direction = "right"
        player_frame = 1
    elif keyboard.left: #elif stops player making diagonal movements
        from_player_x = player_x
        from_player_y = player_y
        player_x -= 1
        player_direction = "left"
        player_frame = 1
    elif keyboard.up:
        from_player_x = player_x
        from_player_y = player_y
        player_y -= 1
        player_direction = "up"
        player_frame = 1
    elif keyboard.down:
        from_player_x = player_x
        from_player_y = player_y
        player_y += 1
        player_direction = "down"
        player_frame = 1
```

check for exiting the room

```
if player_x == room_width: # through door on RIGHT
    clock.unschedule(hazard_move)
    current_room += 1
    generate_map()
    player_x = 0 # enter at left
    player_y = int(room_height / 2) # enter at door
    player_frame = 0
    start_room()
    return
```

```
if player_x == -1: # through door on LEFT
    clock.unschedule(hazard_move)
    current_room -= 1
    generate_map()
    player_x = room_width - 1 # enter at right
    player_y = int(room_height / 2) # enter at door
    player_frame = 0
    start_room()
    return
```

```
if player_y == room_height: # through door at BOTTOM
    clock.unschedule(hazard_move)
    current_room += MAP_WIDTH
    generate_map()
    player_y = 0 # enter at top
    player_x = int(room_width / 2) # enter at door
    player_frame = 0
    start_room()
    return
```

```
if player_y == -1: # through door at TOP
    clock.unschedule(hazard_move)
    current_room -= MAP_WIDTH
    generate_map()
    player_y = room_height - 1 # enter at bottom
    player_x = int(room_width / 2) # enter at door
    player_frame = 0
    start_room()
```

```
return
```

```
if keyboard.g:  
    pick_up_object()
```

```
if keyboard.tab and len(in_my_pockets) > 0:  
    selected_item += 1  
    if selected_item > len(in_my_pockets) - 1:  
        selected_item = 0  
    item_carrying = in_my_pockets[selected_item]  
    display_inventory()
```

```
if keyboard.d and item_carrying:  
    drop_object(old_player_y, old_player_x)
```

```
if keyboard.space:  
    examine_object()
```

```
if keyboard.u:  
    use_object()
```

```
#### Teleporter for testing
```

```
#### Remove this section for the real game
```

```
## if keyboard.x:  
##     current_room = int(input("Enter room number:"))  
##     player_x = 2  
##     player_y = 2  
##     generate_map()  
##     start_room()  
##     sounds.teleport.play()  
#### Teleport section ends
```

```
# If the player is standing somewhere they shouldn't, move them back.
```

```
if room_map[player_y][player_x] not in items_player_may_stand_on \  
    or hazard_map[player_y][player_x] != 0:  
    player_x = old_player_x  
    player_y = old_player_y  
    player_frame = 0
```

```

if room_map[player_y][player_x] == 48: # toxic floor
    deplete_energy(1)

if player_direction == "right" and player_frame > 0:
    player_offset_x = -1 + (0.25 * player_frame)
if player_direction == "left" and player_frame > 0:
    player_offset_x = 1 - (0.25 * player_frame)
if player_direction == "up" and player_frame > 0:
    player_offset_y = 1 - (0.25 * player_frame)
if player_direction == "down" and player_frame > 0:
    player_offset_y = -1 + (0.25 * player_frame)

#####

## DISPLAY ##

#####

def draw_image(image, y, x):
    screen.blit(
        image,
        (top_left_x + (x * TILE_SIZE),
         top_left_y + (y * TILE_SIZE) - image.get_height())
    )

def draw_shadow(image, y, x):
    screen.blit(
        image,
        (top_left_x + (x * TILE_SIZE),
         top_left_y + (y * TILE_SIZE))
    )

def draw_player():
    player_image = PLAYER[player_direction][player_frame]
    draw_image(player_image, player_y + player_offset_y,
               player_x + player_offset_x)
    player_image_shadow = PLAYER_SHADOW[player_direction][player_frame]
    draw_shadow(player_image_shadow, player_y + player_offset_y,
               player_x + player_offset_x)

```



```

def draw():
    if game_over:
        return

    # Clear the game arena area.
    box = Rect((0, 150), (800, 600))
    screen.draw.filled_rect(box, RED)
    box = Rect((0, 0), (800, top_left_y + (room_height - 1)*30))
    screen.surface.set_clip(box)
    floor_type = get_floor_type()

    for y in range(room_height): # Lay down floor tiles, then items on floor.
        for x in range(room_width):
            draw_image(objects[floor_type][0], y, x)
            # Next line enables shadows to fall on top of objects on floor
            if room_map[y][x] in items_player_may_stand_on:
                draw_image(objects[room_map[y][x]][0], y, x)

    # Pressure pad in room 26 is added here, so props can go on top of it.
    if current_room == 26:
        draw_image(objects[39][0], 8, 2)
        image_on_pad = room_map[8][2]
        if image_on_pad > 0:
            draw_image(objects[image_on_pad][0], 8, 2)

    for y in range(room_height):
        for x in range(room_width):
            item_here = room_map[y][x]
            # Player cannot walk on 255: it marks spaces used by wide objects.
            if item_here not in items_player_may_stand_on + [255]:
                image = objects[item_here][0]
                if (current_room in outdoor_rooms
                    and y == room_height - 1
                    and room_map[y][x] == 1) or \
                    (current_room not in outdoor_rooms
                     and y == room_height - 1
                     and room_map[y][x] == 1
                     and x > 0
                     and x < room_width - 1):

```

```

# Add transparent wall image in the front row.
image = PILLARS[wall_transparency_frame]

draw_image(image, y, x)

if objects[item_here][1] is not None: # If object has a shadow
    shadow_image = objects[item_here][1]
    # if shadow might need horizontal tiling
    if shadow_image in [images.half_shadow,
                        images.full_shadow]:
        shadow_width = int(image.get_width() / TILE_SIZE)
        # Use shadow across width of object.
        for z in range(0, shadow_width):
            draw_shadow(shadow_image, y, x+z)
    else:
        draw_shadow(shadow_image, y, x)

hazard_here = hazard_map[y][x]
if hazard_here != 0: # If there's a hazard at this position
    draw_image(objects[hazard_here][0], y, x)

if (player_y == y):
    draw_player()

screen.surface.set_clip(None)

def adjust_wall_transparency():
    global wall_transparency_frame

    if (player_y == room_height - 2
        and room_map[room_height - 1][player_x] == 1
        and wall_transparency_frame < 4):
        wall_transparency_frame += 1 # Fade wall out.

    if ((player_y < room_height - 2
        or room_map[room_height - 1][player_x] != 1)
        and wall_transparency_frame > 0):
        wall_transparency_frame -= 1 # Fade wall in.

```

```

def show_text(text_to_show, line_number):
    if game_over:
        return
    text_lines = [15, 50]
    box = Rect((0, text_lines[line_number]), (800, 35))
    screen.draw.filled_rect(box, BLACK)
    screen.draw.text(text_to_show,
                      (20, text_lines[line_number]), color=GREEN)

#####

##  PROPS  ##

#####

# Props are objects that may move between rooms, appear or disappear.
# All props must be set up here. Props not yet in the game go into room 0.
# object number : [room, y, x]
props = {
    20: [31, 0, 4], 21: [26, 0, 1], 22: [41, 0, 2], 23: [39, 0, 5],
    24: [45, 0, 2],
    25: [32, 0, 2], 26: [27, 12, 5], # two sides of same door
    40: [0, 8, 6], 53: [45, 1, 5], 54: [0, 0, 0], 55: [0, 0, 0],
    56: [0, 0, 0], 57: [35, 4, 6], 58: [0, 0, 0], 59: [31, 1, 7],
    60: [0, 0, 0], 61: [36, 1, 1], 62: [36, 1, 6], 63: [0, 0, 0],
    64: [27, 8, 3], 65: [50, 1, 7], 66: [39, 5, 6], 67: [46, 1, 1],
    68: [0, 0, 0], 69: [30, 3, 3], 70: [47, 1, 3],
    71: [0, LANDER_Y, LANDER_X], 72: [0, 0, 0], 73: [27, 4, 6],
    74: [28, 1, 11], 75: [0, 0, 0], 76: [41, 3, 5], 77: [0, 0, 0],
    78: [35, 9, 11], 79: [26, 3, 2], 80: [41, 7, 5], 81: [29, 1, 1]
}

checksum = 0
for key, prop in props.items():
    if key != 71: # 71 is skipped because it's different each game.
        checksum += (prop[0] * key
                     + prop[1] * (key + 1)
                     + prop[2] * (key + 2))
print(len(props), "props")

```

```
assert len(props) == 37, "Expected 37 prop items"
print("Prop checksum:", checksum)
assert checksum == 61414, "Error in props data"
```

```
in_my_pockets = [55]
selected_item = 0 # the first item
item_carrying = in_my_pockets[selected_item]
```

```
RECIPES = [
    [62, 35, 63], [76, 28, 77], [78, 38, 54], [73, 74, 75],
    [59, 54, 60], [77, 55, 56], [56, 57, 58], [71, 65, 72],
    [88, 58, 89], [89, 60, 90], [67, 35, 68]
]
```

```
checksum = 0
check_counter = 1
for recipe in RECIPES:
    checksum += (recipe[0] * check_counter
                 + recipe[1] * (check_counter + 1)
                 + recipe[2] * (check_counter + 2))
    check_counter += 3
print(len(RECIPES), "recipes")
assert len(RECIPES) == 11, "Expected 11 recipes"
assert checksum == 37296, "Error in recipes data"
print("Recipe checksum:", checksum)
```

```
#####
## PROP INTERACTIONS ##
#####
```

```
def find_object_start_x():
    checker_x = player_x
    while room_map[player_y][checker_x] == 255:
        checker_x -= 1
    return checker_x
```

```
def get_item_under_player():
    item_x = find_object_start_x()
```

```
item_player_is_on = room_map[player_y][item_x]
return item_player_is_on
```

```
def pick_up_object():
    global room_map
    # Get object number at player's location.
    item_player_is_on = get_item_under_player()
    if item_player_is_on in items_player_may_carry:
        # Clear the floor space.
        room_map[player_y][player_x] = get_floor_type()
        add_object(item_player_is_on)
        show_text("Now carrying " + objects[item_player_is_on][3], 0)
        sounds.pickup.play()
        time.sleep(0.5)
    else:
        show_text("You can't carry that!", 0)
```

```
def add_object(item): # Adds item to inventory.
    global selected_item, item_carrying
    in_my_pockets.append(item)
    item_carrying = item
    # Minus one because indexes start at 0.
    selected_item = len(in_my_pockets) - 1
    display_inventory()
    props[item][0] = 0 # Carried objects go into room 0 (off the map).
```

```
def display_inventory():
    box = Rect((0, 45), (800, 105))
    screen.draw.filled_rect(box, BLACK)

    if len(in_my_pockets) == 0:
        return

    start_display = (selected_item // 16) * 16
    list_to_show = in_my_pockets[start_display : start_display + 16]
    selected_marker = selected_item % 16

    for item_counter in range(len(list_to_show)):
        item_number = list_to_show[item_counter]
```

```

image = objects[item_number][0]
screen.blit(image, (25 + (46 * item_counter), 90))
box_left = (selected_marker * 46) - 3
box = Rect((22 + box_left, 85), (40, 40))
screen.draw.rect(box, WHITE)
item_highlighted = in_my_pockets[selected_item]
description = objects[item_highlighted][2]
screen.draw.text(description, (20, 130), color="white")

```

```

def drop_object(old_y, old_x):
    global room_map, props
    if room_map[old_y][old_x] in [0, 2, 39]: # places you can drop things
        props[item_carrying][0] = current_room
        props[item_carrying][1] = old_y
        props[item_carrying][2] = old_x
        room_map[old_y][old_x] = item_carrying
        show_text("You have dropped " + objects[item_carrying][3], 0)
        sounds.drop.play()
        remove_object(item_carrying)
        time.sleep(0.5)
    else: # This only happens if there is already a prop here
        show_text("You can't drop that there.", 0)
        time.sleep(0.5)

```

```

def remove_object(item): # Takes item out of inventory
    global selected_item, in_my_pockets, item_carrying
    in_my_pockets.remove(item)
    selected_item = selected_item - 1
    if selected_item < 0:
        selected_item = 0
    if len(in_my_pockets) == 0: # If they're not carrying anything
        item_carrying = False # Set item_carrying to False
    else: # Otherwise set it to the new selected item
        item_carrying = in_my_pockets[selected_item]
    display_inventory()

```

```

def examine_object():
    item_player_is_on = get_item_under_player()
    left_tile_of_item = find_object_start_x()

```

```

if item_player_is_on in [0, 2]: # don't describe the floor
    return
description = "You see: " + objects[item_player_is_on][2]
for prop_number, details in props.items():
    # props = object number: [room number, y, x]
    if details[0] == current_room: # if prop is in the room
        # If prop is hidden (= at player's location but not on map)
        if (details[1] == player_y
            and details[2] == left_tile_of_item
            and room_map[details[1]][details[2]] != prop_number):
            add_object(prop_number)
            description = "You found " + objects[prop_number][3]
            sounds.combine.play()
show_text(description, 0)
time.sleep(0.5)

```

```
#####
```

```
## USE OBJECTS ##
```

```
#####
```

```
def use_object():
```

```

    global room_map, props, item_carrying, air, selected_item, energy
    global in_my_pockets, suit_stitched, air_fixed, game_over

```

```
use_message = "You fiddle around with it but don't get anywhere."
```

```
standard_responses = {
```

```
    4: "Air is running out! You can't take this lying down!",
```

```
    6: "This is no time to sit around!",
```

```
    7: "This is no time to sit around!",
```

```
    32: "It shakes and rumbles, but nothing else happens.",
```

```
    34: "Ah! That's better. Now wash your hands.",
```

```
    35: "You wash your hands and shake the water off.",
```

```
    37: "The test tubes smoke slightly as you shake them.",
```

```
    54: "You chew the gum. It's sticky like glue.",
```

```
    55: "The yoyo bounces up and down, slightly slower than on Earth",
```

```
    56: "It's a bit too fiddly. Can you thread it on something?",
```

```
    59: "You need to fix the leak before you can use the canister",
```

```
    61: "You try signalling with the mirror, but nobody can see you.",
```

```

62: "Don't throw resources away. Things might come in handy...",
67: "To enjoy yummy space food, just add water!",
75: "You are at Sector: " + str(current_room) + " // X: " \
    + str(player_x) + " // Y: " + str(player_y)
}

```

Get object number at player's location.

```

item_player_is_on = get_item_under_player()
for this_item in [item_player_is_on, item_carrying]:
    if this_item in standard_responses:
        use_message = standard_responses[this_item]

```

```

if item_carrying == 70 or item_player_is_on == 70:
    use_message = "Banging tunes!"
    sounds.steelmusic.play(2)

```

```

elif item_player_is_on == 11:
    use_message = "AIR: " + str(air) + \
        "% / ENERGY " + str(energy) + "% / "
    if not suit_stitched:
        use_message += "*ALERT* SUIT FABRIC TORN / "
    if not air_fixed:
        use_message += "*ALERT* SUIT AIR BOTTLE MISSING"
    if suit_stitched and air_fixed:
        use_message += " SUIT OK"
    show_text(use_message, 0)
    sounds.say_status_report.play()
    time.sleep(0.5)
    # If "on" the computer, player intention is clearly status update.
    # Return to stop another object use accidentally overriding this.
    return

```

```

elif item_carrying == 60 or item_player_is_on == 60:
    use_message = "You fix " + objects[60][3] + " to the suit"
    air_fixed = True
    air = 90
    air_countdown()
    remove_object(60)

```



```

elif (item_carrying == 58 or item_player_is_on == 58) \
    and not suit_stitched:
    use_message = "You use " + objects[56][3] + \
        " to repair the suit fabric"
    suit_stitched = True
    remove_object(58)

elif item_carrying == 72 or item_player_is_on == 72:
    use_message = "You radio for help. A rescue ship is coming. \
Rendezvous Sector 13, outside."
    props[40][0] = 13

elif (item_carrying == 66 or item_player_is_on == 66) \
    and current_room in outdoor_rooms:
    use_message = "You dig..."
    if (current_room == LANDER_SECTOR
        and player_x == LANDER_X
        and player_y == LANDER_Y):
        add_object(71)
        use_message = "You found the Poodle lander!"

elif item_player_is_on == 40:
    clock.unschedule(air_countdown)
    show_text("Congratulations, " + PLAYER_NAME + "!", 0)
    show_text("Mission success! You have made it to safety.", 1)
    game_over = True
    sounds.take_off.play()
    game_completion_sequence()

elif item_player_is_on == 16:
    energy += 1
    if energy > 100:
        energy = 100
    use_message = "You munch the lettuce and get a little energy back"
    draw_energy_air()

elif item_player_is_on == 42:
    if current_room == 27:
        open_door(26)

```

```

props[25][0] = 0 # Door from RM32 to engineering bay
props[26][0] = 0 # Door inside engineering bay
clock.schedule_unique(shut_engineering_door, 60)
use_message = "You press the button"
show_text("Door to engineering bay is open for 60 seconds", 1)
sounds.say_doors_open.play()
sounds.doors.play()

elif item_carrying == 68 or item_player_is_on == 68:
    energy = 100
    use_message = "You use the food to restore your energy"
    remove_object(68)
    draw_energy_air()

if suit_stitched and air_fixed: # open airlock access
    if current_room == 31 and props[20][0] == 31:
        open_door(20) # which includes removing the door
        sounds.say_airlock_open.play()
        show_text("The computer tells you the airlock is now open.", 1)
    elif props[20][0] == 31:
        props[20][0] = 0 # remove door from map
        sounds.say_airlock_open.play()
        show_text("The computer tells you the airlock is now open.", 1)

for recipe in RECIPES:
    ingredient1 = recipe[0]
    ingredient2 = recipe[1]
    combination = recipe[2]
    if (item_carrying == ingredient1
        and item_player_is_on == ingredient2) \
        or (item_carrying == ingredient2
            and item_player_is_on == ingredient1):
        use_message = "You combine " + objects[ingredient1][3] \
            + " and " + objects[ingredient2][3] \
            + " to make " + objects[combination][3]
        if item_player_is_on in props.keys():
            props[item_player_is_on][0] = 0
            room_map[player_y][player_x] = get_floor_type()
            in_my_pockets.remove(item_carrying)

```

```
add_object(combination)
sounds.combine.play()
```

```
# {key object number: door object number}
```

```
ACCESS_DICTIONARY = { 79:22, 80:23, 81:24 }
```

```
if item_carrying in ACCESS_DICTIONARY:
```

```
    door_number = ACCESS_DICTIONARY[item_carrying]
```

```
    if props[door_number][0] == current_room:
```

```
        use_message = "You unlock the door!"
```

```
        sounds.say_doors_open.play()
```

```
        sounds.doors.play()
```

```
        open_door(door_number)
```

```
show_text(use_message, 0)
```

```
time.sleep(0.5)
```

```
def game_completion_sequence():
```

```
    global launch_frame #(initial value is 0, set up in VARIABLES section)
```

```
    box = Rect((0, 150), (800, 600))
```

```
    screen.draw.filled_rect(box, (128, 0, 0))
```

```
    box = Rect ((0, top_left_y - 30), (800, 390))
```

```
    screen.surface.set_clip(box)
```

```
    for y in range(0, 13):
```

```
        for x in range(0, 13):
```

```
            draw_image(images.soil, y, x)
```

```
launch_frame += 1
```

```
if launch_frame < 9:
```

```
    draw_image(images.rescue_ship, 8 - launch_frame, 6)
```

```
    draw_shadow(images.rescue_ship_shadow, 8 + launch_frame, 6)
```

```
    clock.schedule(game_completion_sequence, 0.25)
```

```
else:
```

```
    screen.surface.set_clip(None)
```

```
    screen.draw.text("MISSION", (200, 380), color = "white",
```

```
        fontsize = 128, shadow = (1, 1), scolor = "black")
```

```
    screen.draw.text("COMPLETE", (145, 480), color = "white",
```

```
        fontsize = 128, shadow = (1, 1), scolor = "black")
```

```
    sounds.completion.play()
```

```
    sounds.say_mission_complete.play()
```

```
#####  
## DOORS ##  
#####
```

```
def open_door(opening_door_number):  
    global door_frames, door_shadow_frames  
    global door_frame_number, door_object_number  
    door_frames = [images.door1, images.door2, images.door3,  
                   images.door4, images.floor]  
    # (Final frame restores shadow ready for when door reappears).  
    door_shadow_frames = [images.door1_shadow, images.door2_shadow,  
                          images.door3_shadow, images.door4_shadow,  
                          images.door_shadow]  
    door_frame_number = 0  
    door_object_number = opening_door_number  
    do_door_animation()
```

```
def close_door(closing_door_number):  
    global door_frames, door_shadow_frames  
    global door_frame_number, door_object_number, player_y  
    door_frames = [images.door4, images.door3, images.door2,  
                   images.door1, images.door]  
    door_shadow_frames = [images.door4_shadow, images.door3_shadow,  
                          images.door2_shadow, images.door1_shadow,  
                          images.door_shadow]  
    door_frame_number = 0  
    door_object_number = closing_door_number  
    # If player is in same row as a door, they must be in open doorway  
    if player_y == props[door_object_number][1]:  
        if player_y == 0: # if in the top doorway  
            player_y = 1 # move them down  
        else:  
            player_y = room_height - 2 # move them up  
    do_door_animation()
```

```
def do_door_animation():  
    global door_frames, door_frame_number, door_object_number, objects
```

```

objects[door_object_number][0] = door_frames[door_frame_number]
objects[door_object_number][1] = door_shadow_frames[door_frame_number]
door_frame_number += 1
if door_frame_number == 5:
    if door_frames[-1] == images.floor:
        props[door_object_number][0] = 0 # remove door from props list
        # Regenerate room map from the props
        # to put the door in the room if required.
        generate_map()
    else:
        clock.schedule(do_door_animation, 0.15)

def shut_engineering_door():
    global current_room, door_room_number, props
    props[25][0] = 32 # Door from room 32 to the engineering bay.
    props[26][0] = 27 # Door inside engineering bay.
    generate_map() # Add door to room_map for if in affected room.
    if current_room == 27:
        close_door(26)
    if current_room == 32:
        close_door(25)
    show_text("The computer tells you the doors are closed.", 1)
    sounds.say_doors_closed.play()

def door_in_room_26():
    global airlock_door_frame, room_map
    frames = [images.door, images.door1, images.door2,
              images.door3, images.door4, images.floor
              ]

    shadow_frames = [images.door_shadow, images.door1_shadow,
                     images.door2_shadow, images.door3_shadow,
                     images.door4_shadow, None]

    if current_room != 26:
        clock.unschedule(door_in_room_26)
        return

    # prop 21 is the door in Room 26.

```

```

if ((player_y == 8 and player_x == 2) or props[63] == [26, 8, 2]) \
    and props[21][0] == 26:
    airlock_door_frame += 1
    if airlock_door_frame == 5:
        props[21][0] = 0 # Remove door from map when fully open.
        room_map[0][1] = 0
        room_map[0][2] = 0
        room_map[0][3] = 0

```

```

if ((player_y != 8 or player_x != 2) and props[63] != [26, 8, 2]) \
    and airlock_door_frame > 0:
    if airlock_door_frame == 5:
        # Add door to props and map so animation is shown.
        props[21][0] = 26
        room_map[0][1] = 21
        room_map[0][2] = 255
        room_map[0][3] = 255
    airlock_door_frame -= 1

```

```

objects[21][0] = frames[airlock_door_frame]
objects[21][1] = shadow_frames[airlock_door_frame]

```

```
#####
```

```
##  AIR  ##
```

```
#####
```

```
def draw_energy_air():
```

```

    box = Rect((20, 765), (350, 20))
    screen.draw.filled_rect(box, BLACK)
    screen.draw.text("AIR", (20, 766), color=BLUE)
    screen.draw.text("ENERGY", (180, 766), color=YELLOW)

```

```

if air > 0:
    box = Rect((50, 765), (air, 20))
    screen.draw.filled_rect(box, BLUE) # Draw new air bar.

```

```

if energy > 0:
    box = Rect((250, 765), (energy, 20))

```

```
screen.draw.filled_rect(box, YELLOW) # Draw new energy bar.
```

```
def end_the_game(reason):  
    global game_over  
    show_text(reason, 1)  
    game_over = True  
    sounds.say_mission_fail.play()  
    sounds.gameover.play()  
    screen.draw.text("GAME OVER", (120, 400), color = "white",  
                    fontsize = 128, shadow = (1, 1), scolor = "black")
```

```
def air_countdown():  
    global air, game_over  
    if game_over:  
        return # Don't sap air when they're already dead.  
    air -= 1  
    if air == 20:  
        sounds.say_air_low.play()  
    if air == 10:  
        sounds.say_act_now.play()  
    draw_energy_air()  
    if air < 1:  
        end_the_game("You're out of air!")
```

```
def alarm():  
    show_text("Air is running out, " + PLAYER_NAME  
            + "! Get to safety, then radio for help!", 1)  
    sounds.alarm.play(3)  
    sounds.say_breach.play()
```

```
#####  
## HAZARDS ##  
#####
```

```
hazard_data = {  
    # room number: [[y, x, direction, bounce addition to direction]]  
    28: [[1, 8, 2, 1], [7, 3, 4, 1]], 32: [[1, 5, 4, -1]],
```

```

34: [[5, 1, 1, 1], [5, 5, 1, 2]], 35: [[4, 4, 1, 2], [2, 5, 2, 2]],
36: [[2, 1, 2, 2]], 38: [[1, 4, 3, 2], [5, 8, 1, 2]],
40: [[3, 1, 3, -1], [6, 5, 2, 2], [7, 5, 4, 2]],
41: [[4, 5, 2, 2], [6, 3, 4, 2], [8, 1, 2, 2]],
42: [[2, 1, 2, 2], [4, 3, 2, 2], [6, 5, 2, 2]],
46: [[2, 1, 2, 2]],
48: [[1, 8, 3, 2], [8, 8, 1, 2], [3, 9, 3, 2]]
}

```

```

def deplete_energy(penalty):
    global energy, game_over
    if game_over:
        return # Don't sap energy when they're already dead.
    energy = energy - penalty
    draw_energy_air()
    if energy < 1:
        end_the_game("You're out of energy!")

def hazard_start():
    global current_room_hazards_list, hazard_map
    if current_room in hazard_data.keys():
        current_room_hazards_list = hazard_data[current_room]
        for hazard in current_room_hazards_list:
            hazard_y = hazard[0]
            hazard_x = hazard[1]
            hazard_map[hazard_y][hazard_x] = 49 + (current_room % 3)
        clock.schedule_interval(hazard_move, 0.15)

def hazard_move():
    global current_room_hazards_list, hazard_data, hazard_map
    global old_player_x, old_player_y

    if game_over:
        return

    for hazard in current_room_hazards_list:
        hazard_y = hazard[0]
        hazard_x = hazard[1]
        hazard_direction = hazard[2]

```



```
old_hazard_x = hazard_x
old_hazard_y = hazard_y
hazard_map[old_hazard_y][old_hazard_x] = 0
```

```
if hazard_direction == 1: # up
    hazard_y -= 1
if hazard_direction == 2: # right
    hazard_x += 1
if hazard_direction == 3: # down
    hazard_y += 1
if hazard_direction == 4: # left
    hazard_x -= 1
```

```
hazard_should_bounce = False
```

```
if (hazard_y == player_y and hazard_x == player_x) or \
    (hazard_y == from_player_y and hazard_x == from_player_x
    and player_frame > 0):
    sounds.ouch.play()
    deplete_energy(10)
    hazard_should_bounce = True
```

```
# Stop hazard going out of the doors
```

```
if hazard_x == room_width:
    hazard_should_bounce = True
    hazard_x = room_width - 1
if hazard_x == -1:
    hazard_should_bounce = True
    hazard_x = 0
if hazard_y == room_height:
    hazard_should_bounce = True
    hazard_y = room_height - 1
if hazard_y == -1:
    hazard_should_bounce = True
    hazard_y = 0
```

```
# Stop when hazard hits scenery or another hazard.
```

```
if room_map[hazard_y][hazard_x] not in items_player_may_stand_on \
    or hazard_map[hazard_y][hazard_x] != 0:
```

```
hazard_should_bounce = True
```

```
if hazard_should_bounce:  
    hazard_y = old_hazard_y # Move back to last valid position.  
    hazard_x = old_hazard_x  
    hazard_direction += hazard[3]  
    if hazard_direction > 4:  
        hazard_direction -= 4  
    if hazard_direction < 1:  
        hazard_direction += 4  
    hazard[2] = hazard_direction
```

```
hazard_map[hazard_y][hazard_x] = 49 + (current_room % 3)  
hazard[0] = hazard_y  
hazard[1] = hazard_x
```

```
#####
```

```
## START ##
```

```
#####
```

```
clock.schedule_interval(game_loop, 0.03)  
generate_map()  
clock.schedule_interval(adjust_wall_transparency, 0.05)  
clock.schedule_unique(display_inventory, 1)  
clock.schedule_unique(draw_energy_air, 0.5)  
clock.schedule_unique(alarm, 10)  
# A higher number below gives a longer time limit.  
clock.schedule_interval(air_countdown, 5)  
sounds.mission.play() # Intro music
```

B

TABLE OF VARIABLES, LISTS, AND DICTIONARIES



To help you to understand the Escape listing, I’ve provided the following table, which contains some of the variables, lists, and dictionaries used in the game. I’ve included those that I think will be most useful for customizing the game. You can also use the book’s index to find references to specific variables, lists, and dictionaries.

If the name of a variable, list, or dictionary is capitalized, it means its contents are not intended to be changed after they’re set up.

| Variable, list, or dictionary | Description |
|-------------------------------|---|
| ACCESS_DICTIONARY | Dictionary that pairs keys with doors. See “ Adding Access Controls ” on page 185 (Chapter 11). |
| air | Air remaining for player. Set to starting value in VARIABLES section. |

air_fixed

Set to `True` when the player has fitted the air canister to the suit. Otherwise, `False`.

checksum

Used to check data has been entered correctly when typing in the game listing. If you modify the game data, you will need to modify or disable checksum code. Put a `#` before the `assert` instructions to disable them.

current_room

Number of the room the player is now in. Set it as the starting room in the `VARIABLES` section.

energy

Energy remaining for player. Set to starting value in `VARIABLES` section.

FRIEND1_NAME

A friend's name, used in descriptions of rooms and objects.

FRIEND2_NAME

A friend's name, used in descriptions of rooms and objects.

GAME_MAP

Stores the map of how rooms connect to each other. See [“Creating the Map Data” on page 60 \(Chapter 4\)](#).

game_over

Set to `True` when the game has finished. Otherwise, it should be `False`.

hazard_data

Dictionary containing position and movement information for the moving hazards. See [“Adding the Moving Hazards” on page 203 \(Chapter 12\)](#).

| | |
|------------|--|
| hazard_map | Used to keep track of where moving hazards are in the room the player is now in. Automatically generated. You don't need to modify this. |
|------------|--|

| | |
|--------|--------------------------------------|
| HEIGHT | Height of the game window in pixels. |
|--------|--------------------------------------|

| | |
|---------------|--|
| in_my_pockets | List of object numbers for items player is carrying. Set up in the PROPS section to contain the items the player begins the game with. |
|---------------|--|

| | |
|---------------|---|
| item_carrying | Object number of the item the player has selected in their inventory. |
|---------------|---|

| | |
|-------------------|--|
| item_player_is_on | Object number of the item the player is standing on. |
|-------------------|--|

| | |
|------------------------|---|
| items_player_may_carry | List containing the object numbers of items the player can pick up. |
|------------------------|---|

| | |
|---------------------------|---|
| items_player_may_stand_on | List containing the object numbers of items the player can walk on. |
|---------------------------|---|

| | |
|---------------|--|
| LANDER_SECTOR | Room number where the Poodle lander is hidden. |
|---------------|--|

| | |
|----------|--|
| LANDER_X | The x-coordinate of where the Poodle lander is hidden. |
|----------|--|

| | |
|----------|--|
| LANDER_Y | The y-coordinate of where the Poodle lander is hidden. |
|----------|--|

MAP_HEIGHT

How many rooms tall the map is (see [Chapter 4, Figure 4-1 on page 60](#)).

MAP_WIDTH

How many rooms wide the map is (see [Chapter 4, Figure 4-1 on page 60](#)).

objects

Dictionary containing images and descriptions for all objects in the game. See “[Making the Space Station Objects Dictionary](#)” on page 85 ([Chapter 5](#)).

outdoor_rooms

A range of the planet surface room numbers (see [Chapter 4, Figure 4-1 on page 60](#)).

PILLARS

Dictionary containing animation frames for front wall transparency.

PLAYER

Dictionary containing player animation frames.

player_direction

Direction player is facing. Should be left, right, up, or down.

player_frame

Used for the player’s animation frame.

PLAYER_NAME

Used in descriptions of objects and messages to the player. Set it as your name in the `VARIABLES` section.

PLAYER_SHADOW

Dictionary containing shadows for player animation.

`player_x` Player's x position in the room, measured in tiles. Set it as the starting position in the `VARIABLES` section.

`player_y` Player's y position in the room, measured in tiles. Set it as the starting position in the `VARIABLES` section.

`props` Dictionary containing location of all the moveable objects in the game. See “[Adding the Props Information](#)” on page 151 (Chapter 9).

`RECIPES` List containing ways that objects can be combined to make new objects. See “[Combining Objects](#)” on page 177 (Chapter 10).

`room_map` Used to remember what's at each position in the room the player is now in. Automatically generated. You don't need to modify this.

`scenery` Dictionary containing data for positioning fixed objects in rooms. See “[Understanding the Dictionary for the Scenery Data](#)” on page 97 (Chapter 6).

`standard_responses` Dictionary of messages to display when the player uses items that serve no other purpose.

`suit_stitched` Set to `True` when the player has fixed the suit. Otherwise, `False`.

`use_message` Text shown to player when they use or try to use an object.

WIDTH

Width of the game window in pixels.

C

DEBUGGING YOUR LISTINGS



Some of the listings in this book might not work for you the first time. Don't be put off! This is normal when programming, even for experienced coders. It's easy to overlook details that will make a huge difference to the program. Fixing errors in a program is called *debugging*.

To minimize problems, I've kept the listings as short as possible, so if something doesn't work in a listing, you won't have to check many instructions. I've also included warnings in the text when there's anything particularly tricky that you should look out for.

Remember that if you can't work out how to fix a program, you can use my version of that listing that you downloaded in the book's resources (see [“What's in the ZIP File” on page 8](#)). If you've modified the program, try copying and pasting the new bits from my listing into your program.

In this appendix, I've compiled some tips to help you fix any programs that aren't working for you. When Python spots an error, it usually shows you the line in the program where it first noticed something was wrong. That isn't always the line where the mistake actually is: it's just how far Python got before it noticed a problem. If the line shown looks okay, check the previous line first and then check the other new

instructions in the listing for mistakes.

INDENTATION

Indentation is used to tell Python which bits of the program belong together. For example, all the instructions that belong to a function need to be indented underneath the `def` instruction that defines the function. Instructions that belong to a `while`, `for`, `if`, or `else` command need to be indented too. Listing C-1 provides an example, part of the `get_floor_type()` function.

```
--snip--
❶ def get_floor_type():
❷     if current_room in outdoor_rooms:
❸         return 2 # soil
❹     else:
❺         return 0 # tiled floor
--snip--
```

Listing C-1: An excerpt from the game listing, showing indentation levels

All the instructions belong to the function `get_floor_type()` ❶, so they're all indented by at least four spaces (see ❷ and ❹). The `return` instructions (❸ and ❺) also belong to the `if` ❷ and `else` ❹ commands above them, so they're indented by another four spaces, making eight spaces in total. When you add the colon at the end of the line when typing in the `def`, `if`, and `else` instructions, the indentation on the next line is added automatically in IDLE. Use the `DELETE` key to remove indentation you don't need.

If you get the indentation level wrong for some instructions, the program might behave strangely or simply run slower, even if Python doesn't report any errors. So it's worth double-checking your indentation levels.

If Python does give an error that shows it expected an indented block, it means you haven't indented something that you should have. If Python tells you there's an unexpected indent, you've added too many spaces at the start of the instruction, or you might have instructions indented at different levels that should be lined up. In this book, I've used four spaces for each level of indentation.

CASE SENSITIVITY

Python is case sensitive, which means it matters whether you use uppercase (ABC...) or

lowercase (abc...) letters. Most of the time, you should use lowercase when writing Python code. Here are the exceptions:

- The values `True`, `False`, and `None` have a capital letter at the start. When you type them correctly, they'll be orange in IDLE.
- Some of the variable, dictionary, and list names in the program are uppercase, such as `TILE_SIZE` and `PLAYER`. If your capitalization is inconsistent, you might get an error message saying that a particular name is not defined. Python doesn't recognize two names with different capitalization as the same name. (Check for spelling errors in the name too.)
- Anything inside quotation marks may vary in case. This is text the program uses to do something and is often written so it looks correct when people read it.
- Python ignores anything after a `#` symbol on the same line, so you can use whatever capitalization you like there.

PARENTHESES AND BRACKETS

Check that you're using the correct bracket shapes in the correct order, especially if Python tells you there's a problem with something in a list or dictionary:

- Parentheses `()` are used for tuples and for giving information to a function. For example, the `range()`, `print()` and `len()` functions use parentheses. So do our own functions in the *Escape* game, such as `remove_object()` and `draw_image()`.
- Square brackets `[]` mark the start and the end of a list. Sometimes, you might have a list inside another list, so you'll have several pairs.
- Curly brackets `{}` mark the start and the end of a dictionary.

COLONS

When the code line begins with `for`, `while`, `if`, `else` or `def`, it needs a colon `:` at the end of it. A colon also separates the key from the data in a dictionary. The *Escape* listing doesn't use semicolons `;`, so if there's one in your code, change it to a colon.

COMMAS

Items in a list or tuple need commas between them. When adding new lines to a list,

make sure you include a comma after the last item before adding new items. Look for patterns in the data to help you spot any mistakes involving commas. For example, each list in the `props` dictionary and `recipes` list has three numbers in it.

IMAGES AND SOUNDS

If Python tells you that no images or sounds directory was found, check that you've downloaded the files and are saving your files in the right place. See “[Downloading the Game Files](#)” on [page 7](#) and [Listing 1-1](#) on [page 19](#).

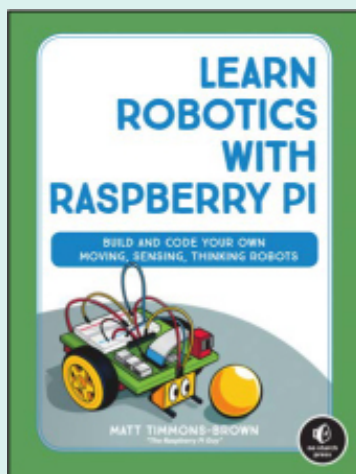
SPELLING

IDLE's color coding can help you spot spelling errors in some instructions. Check that the colors on your screen match the colors in the book. Be careful when you're spelling variables and lists: any mistakes might cause the program to stop or behave strangely.

RESOURCES

Visit <https://nostarch.com/missionpython/> for updates, errata, program files, and other information.

MORE SMART BOOKS FOR CURIOUS KIDS!



LEARN ROBOTICS WITH RASPBERRY PI

Build and Code Your Own Moving, Sensing, Thinking Robots

by MATT TIMMONS-BROWN

FALL 2018, 200 PP., \$24.95

ISBN 978-1-59327-920-2

full color



PYTHON FLASH CARDS

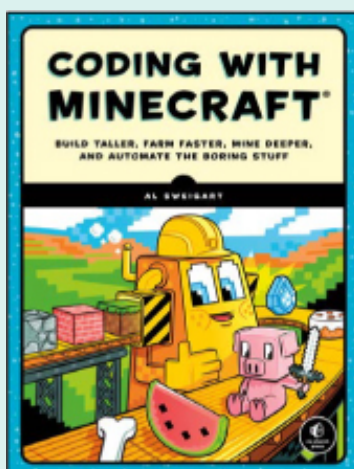
Syntax, Concepts, and Examples

by ERIC MATTHES

FALL 2018, 101 CARDS, \$27.95

ISBN 978-1-59327-896-0

full color



CODING WITH MINECRAFT

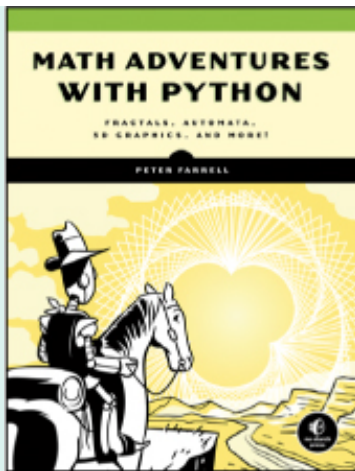
Build Taller, Farm Faster, Mine Deeper, and Automate the Boring Stuff

by AL SWEIGART

MAY 2018, 256 PP., \$29.95

ISBN 978-1-59327-853-3

full color



MATH ADVENTURES WITH PYTHON

Fractals, Automata, 3D Graphics, and More!

by PETER FARRELL

FALL 2018, 304 PP., \$29.95

ISBN 978-1-59327-867-0

full color



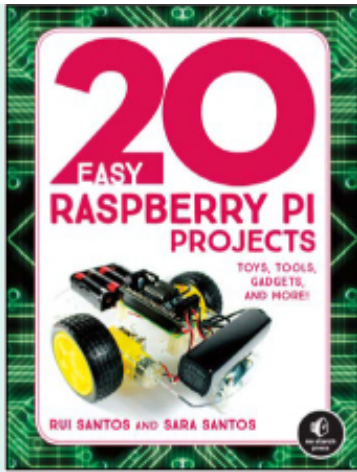
CRACKING CODES WITH PYTHON

An Introduction to Building and Breaking Ciphers

by AL SWEIGART

JANUARY 2018, 416 PP., \$29.95

ISBN 978-1-59327-822-9



20 EASY RASPBERRY PI PROJECTS

Toys, Tools, Gadgets, and More!

by RUI SANTOS *and* SARA SANTOS

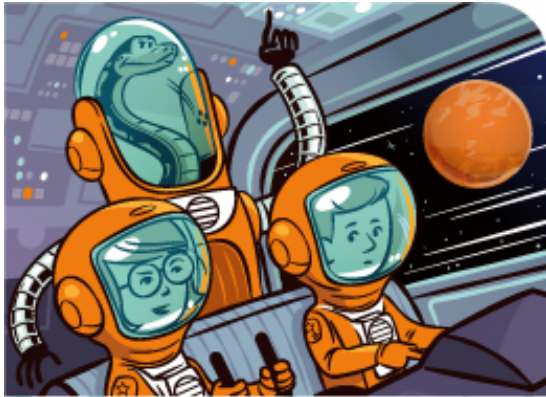
APRIL 2018, 288 PP., \$24.95

ISBN 978-1-59327-843-4

full color

1.800.420.7240 or 1.415.863.9900 | sales@nostarch.com | www.nostarch.com

CODE YOUR OWN SPACE STATION ADVENTURE GAME!



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

In *Mission Python*, you'll code a puzzlebased 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.



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



THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com