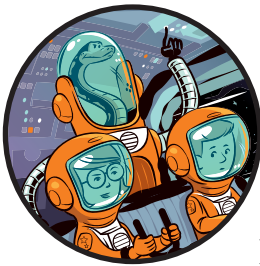




YOUR FIRST SPACEWALK



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.

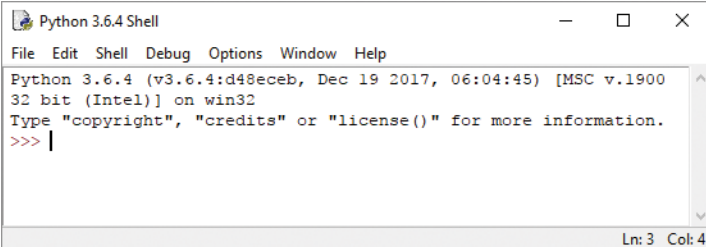
STARTING IDLE ON THE RASPBERRY PI

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.



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> |
Ln: 3 Col: 4
```

Figure 1-2: The Python shell

So let's give Python something to do!

DISPLAYING TEXT

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

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

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

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

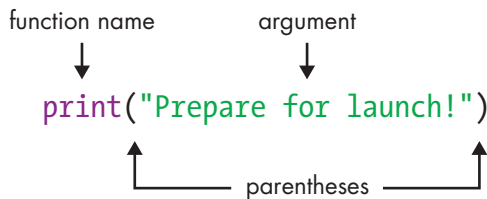


Figure 1-3: The different parts of your first instruction

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

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

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

```
Prepare for launch!
```

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

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

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

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

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

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

TRAINING MISSION #1

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

OUTPUTTING AND USING NUMBERS

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

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

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

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

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

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

INTRODUCING SCRIPT MODE

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

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

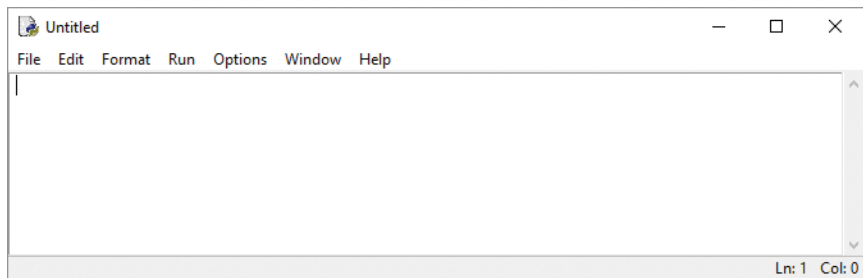


Figure 1-4: Python script mode

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

TIP

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

CREATING THE STARFIELD

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

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

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

```
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).

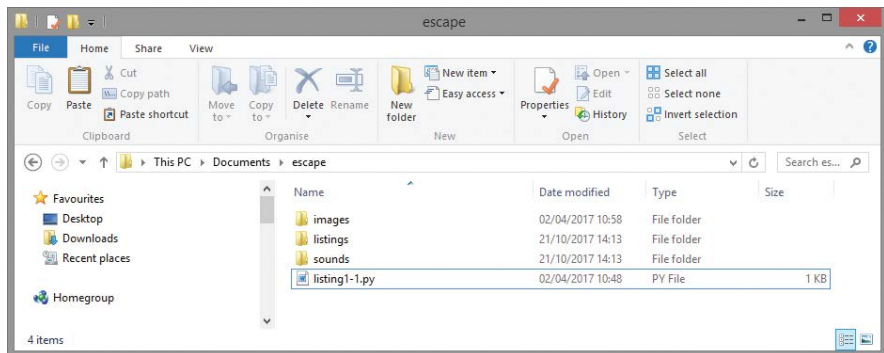


Figure 1-5: Your new Python program and the images folder should be stored in the same place.

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.

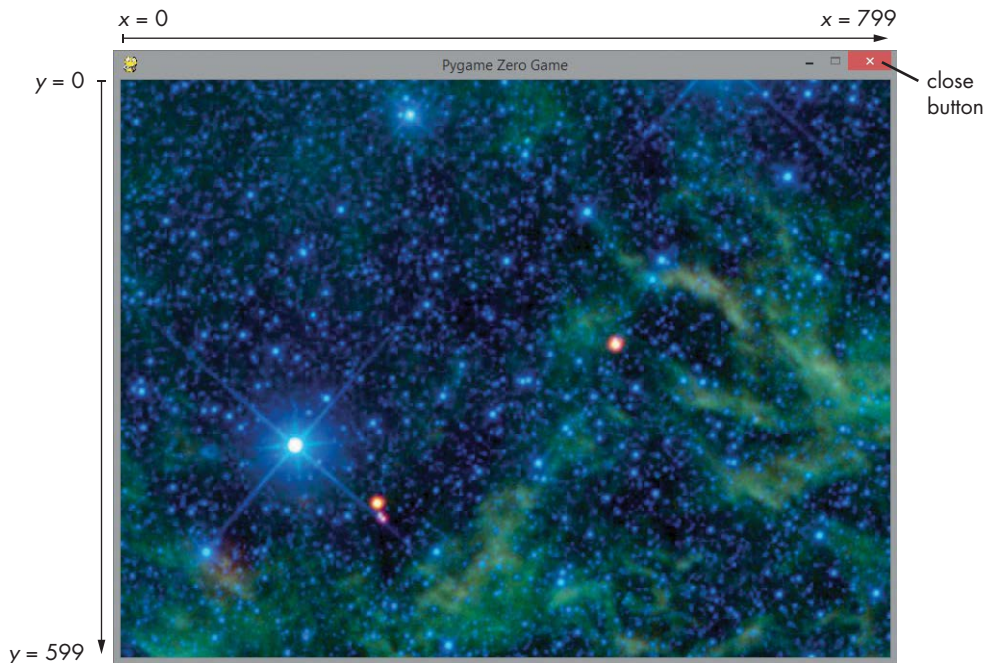


Figure 1-6: The starfield. The starfield image is courtesy of NASA/JPL-Caltech/UCLA and shows star cluster NGC 2259.

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

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

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

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

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

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

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

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

STOPPING YOUR PYGAME ZERO PROGRAM

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

RED ALERT

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

ADDING THE PLANET AND SPACESHIP

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

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

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

listing1-2.py

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

Listing 1-2: Adding Mars and the ship

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

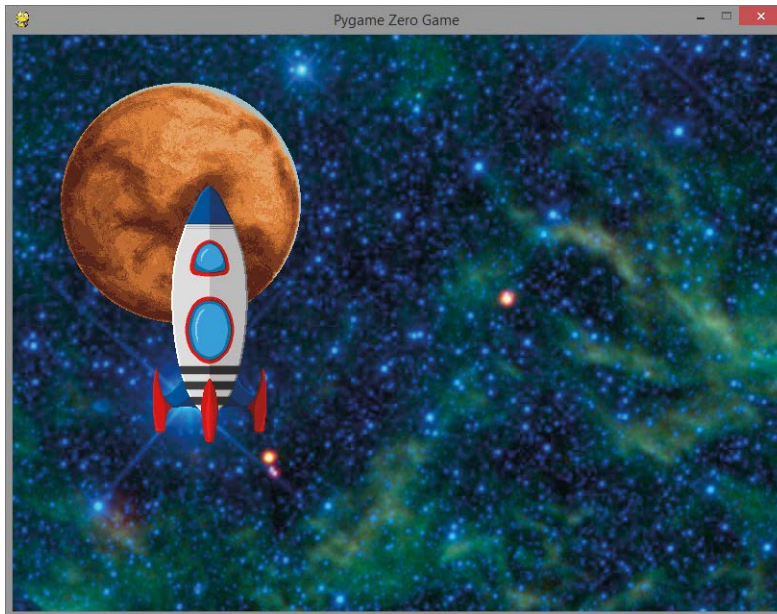


Figure 1-7: Mars and the spaceship. The Mars image was taken by the Hubble Space Telescope in 1991.

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

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

CHANGING PERSPECTIVE: FLYING BEHIND THE PLANET

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

`listing1-3.py`

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

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

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

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

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

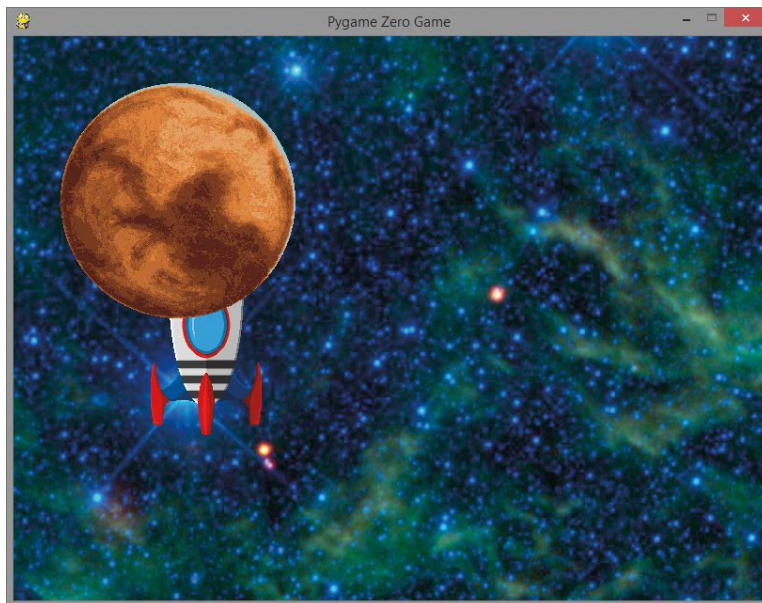


Figure 1-8: The spaceship is now behind the planet.

TRAINING MISSION #2

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

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

SPACEWALKING!

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

listing1-4.py

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

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

❿ 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.

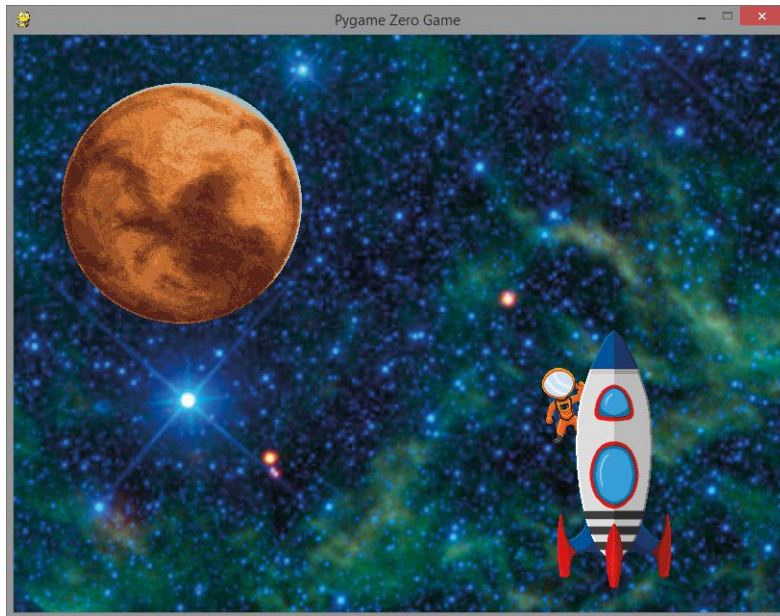


Figure 1-9: You emerge from the ship for your spacewalk.

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 ⑤. If it is, we change the value of `player_x` by adding 5 ⑥, 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 ⑦, which is short for "else if." You can think of *else* as meaning *otherwise* here. In plain English, this part of our program means, "If the right arrow key is pressed, add 5 to the *x* position. Otherwise, if the left key is pressed, subtract 5 from the *x* position." We then use `elif` to check for up and down keypresses in the same way, and change the *y* position to move the astronaut up or down. The `draw()` function uses the `player_x` and `player_y` variables for the astronaut's position, so changing the numbers in these variables makes the astronaut move on the screen.

TIP

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

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

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

TRAINING MISSION #4

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

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

ARE YOU FIT TO FLY?

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

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

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

MISSION DEBRIEF

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

TRAINING MISSION #1

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

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

TRAINING MISSION #2

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

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

TRAINING MISSION #3

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

TRAINING MISSION #4

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

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

