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.

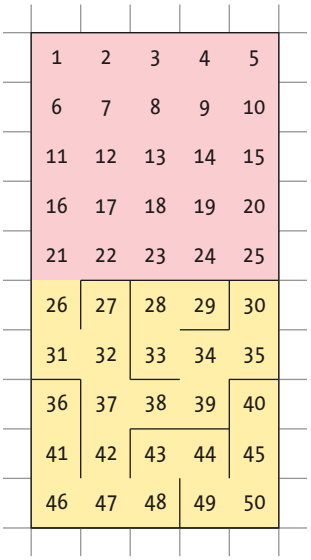


Figure 4-1: The space station map

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*. Remember to save it (and 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).

> **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.

listing4-1.py

```
# 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] )

❼ 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
❽         [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
❾ 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` ❼. 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 ❽. 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 ❾. 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 ❹ 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

From the command line, navigate to your *escape* folder and run the program from the command line using **`pgzrun listing4-1.py`**. An empty game window should open. 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 see an error message in the command line window. If so, 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 use **`pgzrun listing4-2.py`** to run it and check for any error messages in the command line window.

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

❿   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:

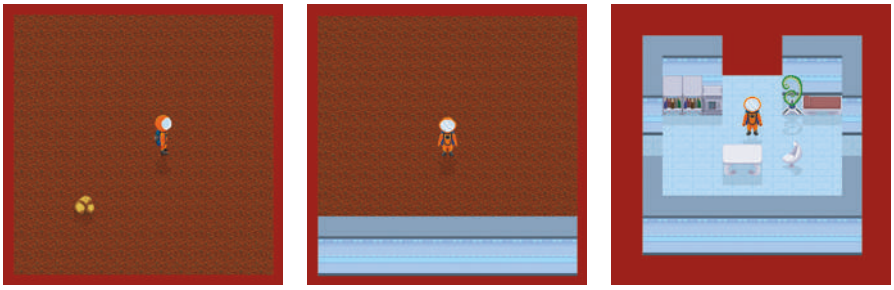
```
["The airlock", 13, 5, True, False]
```

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).



A planet surface room | A planet surface room bordering the space station | An inside room

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 ❼. The `int()` function converts a decimal number into a whole number (an *integer*).

We check for a right exit first ❽. 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 ❾. 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 ❿. 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 command line window. You can also check that the program is working by adding these two lines to the end of your program and running it again:

```
generate_map()
print(room_map)
```

In the command line window you should see 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]]
```

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` at the end of your program to print a different room. Check the output against the map and the `GAME_MAP` code to make sure the results match what you expect. Here is one to try:

```
current_room = 45
generate_map()
print(room_map)
```

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

Make sure you delete all the test instructions from the end of your program when you've finished experimenting.

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*.

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 ❻. The `str()` function converts the room number to a string ❼, 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 ❽, 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.

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.

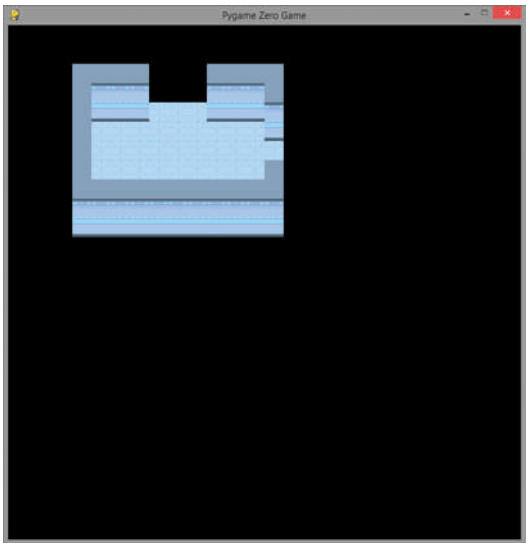


Figure 4-4: The Explorer *shows your starting room in 3D.*

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.

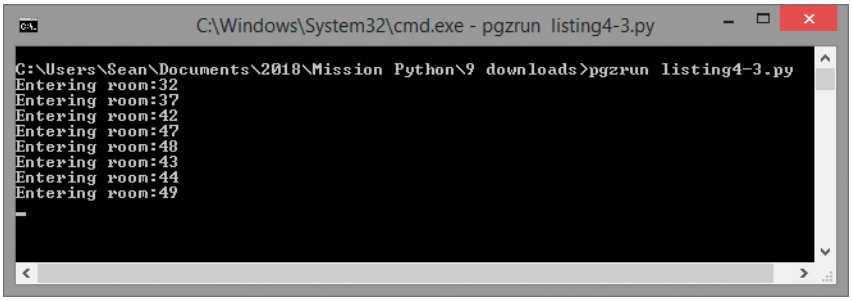


Figure 4-5: The command line window tells you which room you're entering.

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
        ]
--snip--
```

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--
```