MARK VANSTONE

Educational software author from the nineties, author of the ArcVenture series, disappeared into the corporate software wasteland. Rescued by the Raspberry Pi!
technovisualeducation.co.uk
twitter.com/mindexplorers

GET STARTED WITH PYGAME ZERO

You'll Need

- Raspbian Jessie or newer
- An image manipulation program such as GIMP
- A little imagination
- A keyboard

Pygame Zero is a great choice for anyone who wants to start writing computer games on the Raspberry Pi

If you've done some Python coding and wanted to write a game, you may have come across Pygame. The Pygame module adds many functions that help you to write games in Python. Pygame Zero goes one step further to let you skip over the cumbersome process of making all those game loops and setting up your program structure. You don't need to worry about functions to load graphics or keeping data structures for all the game elements. If you just want to get stuck in and start making things happen on the screen without all the fluff, then Pygame Zero is what you need.

>STEP-01

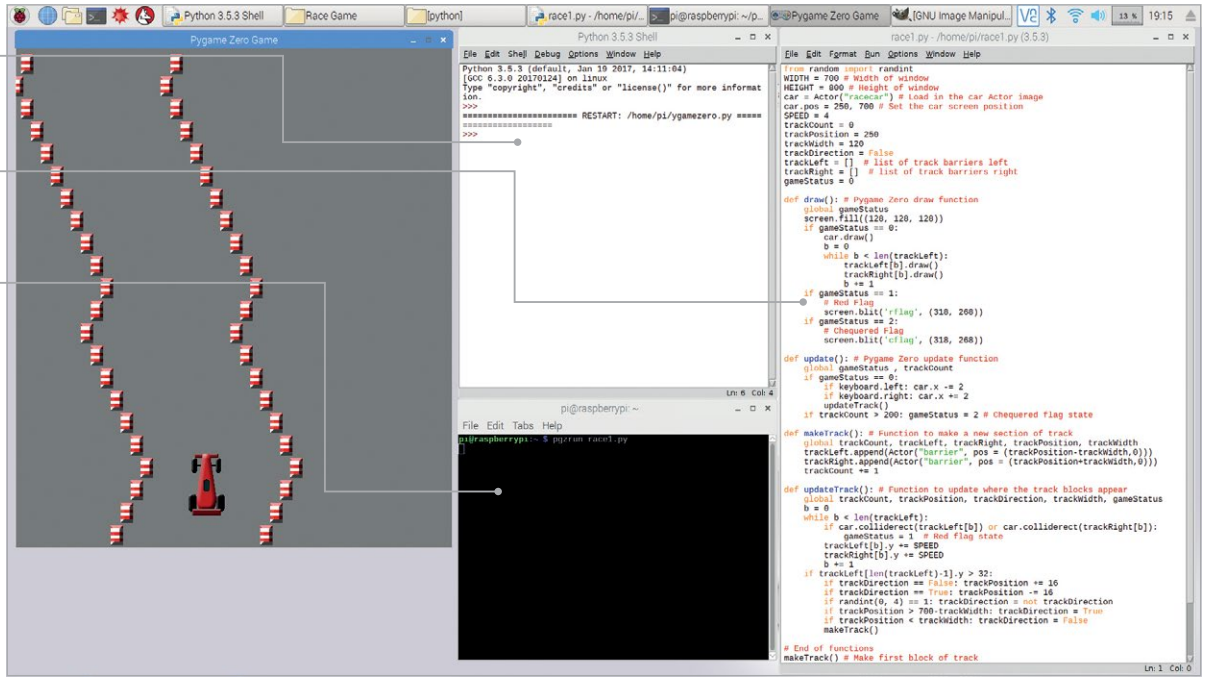
Loading a suitable program editor

The first really labour-saving thing about Pygame Zero is that you can write a program in a simple text editor. For the easiest route we suggest using the IDLE Python 3 editor, as Pygame Zero needs to be formatted like Python with its indents and you'll get the benefit of syntax highlighting to help you along the way.
So the first step in your journey will be to open the Python 3 IDLE editor from the Raspbian main menu, under Programming. You'll be presented with the Python Shell window.

The Python Shell window that appears when we open IDLE

Our program listing. This is a file window from the IDLE application

The Terminal window – enter the command to run our program



TERMINAL SHORTCUTS

Instead of retyping `pgzrun race1.py` in the Terminal window, you can use the up arrow to repeat the last command.

>STEP-02

Writing a Pygame Zero program

To start writing your first Pygame Zero program, go to the File menu of the IDLE Python Shell window and select 'New File' to open up a new editor window – and that's it! You have written your first Pygame Zero program! The Pygame Zero framework assumes that you will want to open a new window to run your game inside, so even a blank file will create a running game environment. Of course at this stage your game doesn't do very much, but you can test it to make sure that you can get a program running.

>STEP-03

Running your first Pygame Zero program

With other Python programs, you can run them directly from the Python file window. Currently IDLE does not support running Pygame Zero programs directly, but the alternative is very simple. First of all, you need to save your blank program file. We suggest saving it as **pygame1.py** in your default user folder (just save the file without changing directory). All you need to do then is open a Terminal window from the main Raspbian menu and type **`pgzrun pygame1.py`** (assuming you called your program **pygame1.py**) and hit **RETURN**. After a few seconds, a window titled 'Pygame Zero Game' should appear.

>STEP-04

Setting up the basics

By default, the Pygame Zero window opens at the size of 800 pixels wide by 600 pixels high. If you want to change the size of your window, there are two predefined variables you can set. If you include **`WIDTH = 700`** in your program, then the window will be set at 700 pixels wide. If you include **`HEIGHT = 800`**, then the window will be set to 800 pixels high. In this tutorial we'll be writing a simple racing game, so we want our window to be a bit taller than it is wide. When you have set the **`WIDTH`** and **`HEIGHT`** variables, you could save your file as **race1.py** and test it like before by typing **`pgzrun race1.py`** into the Terminal window.

>STEP-05

Look! No game loop!

When writing a Python game, normally you would have a game loop – that's a piece of code that is run over and over while the game is running. Pygame Zero does away with this idea and provides predefined functions to handle each of the tasks that the game loop normally performs. The first of these we will look at is the function **`draw()`**. We can write this function into our program the same as we would normally define a function in Python, which is **`def draw():`**. Then, so that you can see the draw function doing something, add a line underneath indented by one tab: **`screen.fill((128, 128, 128))`**. This is shown in the **figure1.py** listing overleaf.

race1.py

Language
>PYTHON

DOWNLOAD:
magpi.cc/VcqutR

```
01. from random import randint
02. WIDTH = 700 # Width of window
03. HEIGHT = 800 # Height of window
04. car = Actor("racecar") # Load in the
    car Actor image
05. car.pos = 250, 700 # Set the car screen position
06. SPEED = 4
07. trackCount = 0
08. trackPosition = 250
09. trackWidth = 120
10. trackDirection = False
11. trackLeft = []  # list of track barriers left
12. trackRight = []  # list of track barriers right
13. gameStatus = 0
14.
15. def draw(): # Pygame Zero draw function
16.     global gameStatus
17.     screen.fill((128, 128, 128))
18.     if gameStatus == 0:
19.         car.draw()
20.         b = 0
21.         while b < len(trackLeft):
22.             trackLeft[b].draw()
23.             trackRight[b].draw()
24.             b += 1
25.     if gameStatus == 1:
26.         # Red Flag
27.         screen.blit('rflag', (318, 268))
28.     if gameStatus == 2:
29.         # Chequered Flag
30.         screen.blit('cflag', (318, 268))
31.
32. def update(): # Pygame Zero update function
33.     global gameStatus , trackCount
34.     if gameStatus == 0:
35.         if keyboard.left: car.x -= 2
36.         if keyboard.right: car.x += 2
37.         updateTrack()
38.     if trackCount > 200: gameStatus = 2 # Chequered flag
    state
39.
40. def makeTrack(): # Function to make a new section of track
41.     global trackCount, trackLeft, trackRight, trackPosition,
    trackWidth
42.     trackLeft.append(Actor("barrier", pos = (trackPosition-
    trackWidth,0)))
43.     trackRight.append(Actor("barrier", pos =
    (trackPosition+trackWidth,0)))
44.     trackCount += 1
45.
46. def updateTrack(): # Function to update where the track
    blocks appear
47.     global trackCount, trackPosition, trackDirection,
    trackWidth, gameStatus
48.     b = 0
49.     while b < len(trackLeft):
50.         if car.colliderect(trackLeft[b]) or
    car.colliderect(trackRight[b]):
51.             gameStatus = 1  # Red flag state
52.         trackLeft[b].y += SPEED
53.         trackRight[b].y += SPEED
54.         b += 1
55.     if trackLeft[len(trackLeft)-1].y > 32:
56.         if trackDirection == False: trackPosition += 16
57.         if trackDirection == True: trackPosition -= 16
58.         if randint(0, 4) == 1: trackDirection = not
    trackDirection
59.         if trackPosition > 700-trackWidth: trackDirection =
    True
60.         if trackPosition < trackWidth: trackDirection =
    False
61.         makeTrack()
62.
63. # End of functions
64. makeTrack() # Make first block of track
```

figure1.py

```
01. WIDTH = 700
02. HEIGHT = 800
03.
04. def draw():
05.     screen.fill((128, 128, 128))
```

Figure 1 To set the height and width of a Pygame Zero window, just set the variables `HEIGHT` and `WIDTH`. Then you can fill the screen with a colour

>STEP-06

The Python format

You may have noticed that in the previous step we said to indent the **`screen.fill`** line by one tab. Pygame Zero follows the same formatting rules as Python, so you will need to take care to indent your code correctly. The indents in Python show that the code is inside a structure. So if you define a function, all the code inside it will be indented by one tab. If you then have a condition or a loop, for example an **`if`** statement, then the contents of that condition will be indented by another tab (so two in total).

> Actors in Pygame Zero are dynamic graphic objects, much the same as sprites

>STEP-07

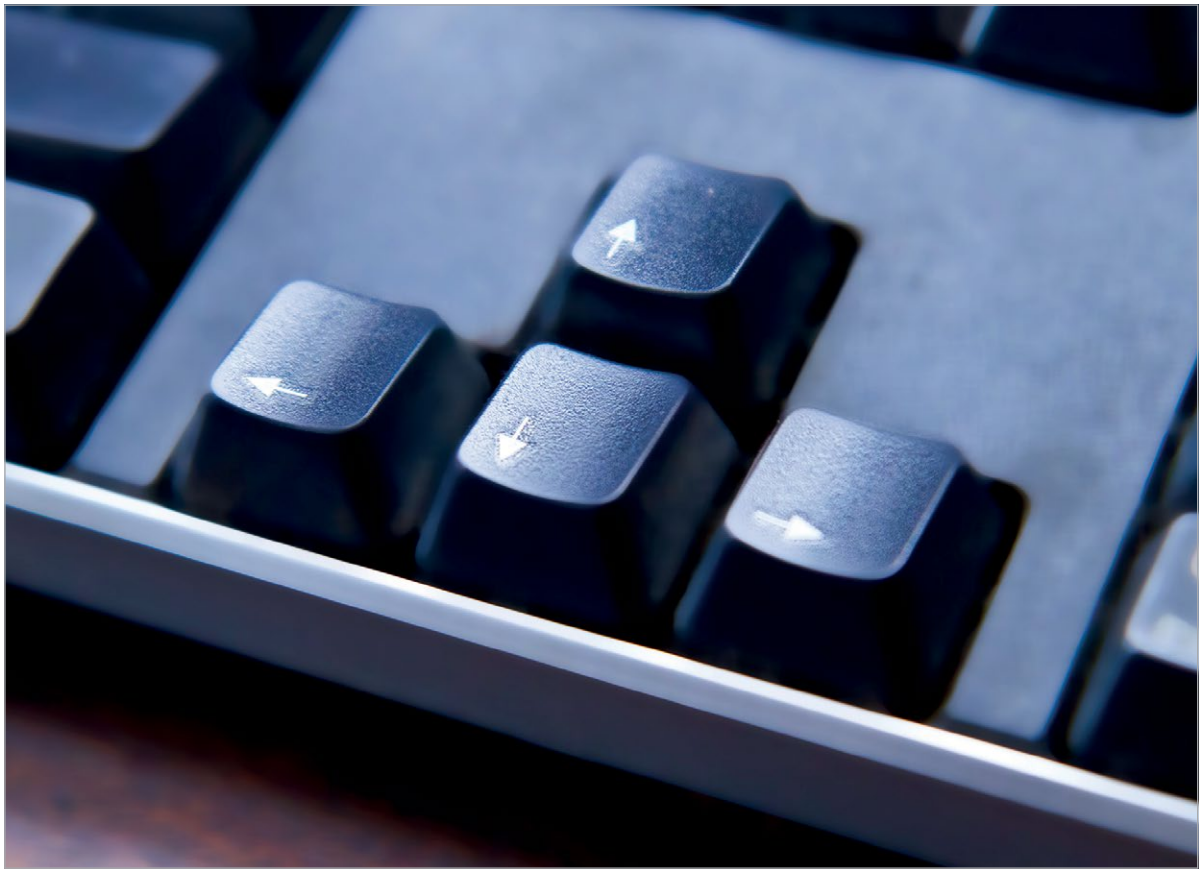
All the world's a stage

The screen object used in Step 5 is a predefined object that refers to the window we've opened for our game. The **`fill`** function fills the window with the RGB value (a tuple value) provided – in this case, a shade of grey. Now that we have our stage set, we can create our Actors. Actors in Pygame Zero are dynamic graphic objects, much the same as sprites in other programming systems. We can load an Actor by typing **`car = Actor("racecar")`**. This is best placed near the top of your program, before the **`draw()`** function.

>STEP-08

It's all about image

When we define an Actor in our program, what we are actually doing is saying 'go and get this image'. In Pygame Zero our images need to be stored in a directory called **images**, next to our program file. So our Actor would be looking for an image file in the **images** folder called **racecar.png**. It could be a GIF or a JPG file, but it is recommended that your images are PNG files as that file type provides good-quality images with transparencies. You can get a full free image creation program called GIMP by typing **`sudo apt-get install gimp`** in your Terminal window. If you want to use our images, you can download them from **magpi.cc/srHWWH**.



Right To respond to key presses, Pygame Zero has a built-in object called `keyboard`. The arrow key states can be read with `keyboard.up`, `keyboard.down`, and so on

THE GRAPHICS

If you use PNG files for your graphics rather than JPGs, you can keep part of the image transparent.

>STEP-09

Drawing your Actor

Once you have loaded in your image by defining your Actor, you can set its position on the screen. You can do this straight after loading the Actor by typing **`car.pos = 250, 500`** to set it at position 250, 500 on the screen. Now, when the **`draw()`** function runs, we want to display our race car at the co-ordinates that we have set. So, in our **`draw()`** function, after the **`screen.fill`** command we can type **`car.draw()`**. This will draw our race car at its defined position. Test your program to make sure this is working, by saving it and running **`pgzrun race1.py`** as before.

>STEP-10

I'm a control freak!

Once we have our car drawing on the screen, the next stage is to enable the player to move it backwards and forwards. We can do this with key presses; in this case we are going to use the left and right arrow keys. We can read the state of these keys inside another predefined function called **`update()`**. We can type in the definition of this function by adding **`def update():`** to our program. This function is continually checked while the game is running. We can now add an indented **`if`** statement to check the state of a key; e.g., **`if keyboard.left:`**.

>STEP-11

Steering the car

We need to write some code to detect key presses of both arrow keys and also to do something about it if we detect that either has been pressed. Continuing from our **`if keyboard.left:`** line, we can write **`car.x -= 2`**. This means subtract 2 from the car's x co-ordinate. It could also be written in long-hand as **`car.x = car.x - 2`**. Then, on the next line and with the same indent as the first **`if`** statement, we can do the same for the right arrow; i.e., **`if keyboard.right: car.x += 2`**. These lines of code will move the car Actor left and right.

>STEP-12

The long and winding road

Now that we have a car that we can steer, we need a track for it to drive on. We are going to build our track out of Actors, one row at a time. We will need to make some lists to keep track of the Actors we create. To create our lists, we can write the following near the top of our program: **`trackLeft = []`** (note the square brackets) and then, on the next line, **`trackRight = []`**. This creates two empty lists: one to hold the data about the left side of the track, and one to hold the data about the right-hand side.

figure2.py

```
01. def makeTrack(): # Function to make a new section of track
02.     global trackCount, trackLeft, trackRight,
    trackPosition, trackWidth
03.     trackLeft.append(Actor("barrier", pos =
    (trackPosition-trackWidth,0)))
04.     trackRight.append(Actor("barrier", pos =
    (trackPosition+trackWidth,0)))
05.     trackCount += 1
```

>STEP-13

Building the track

We will need to set up a few more variables for the track. After your two lists, declare the following variables: **`trackCount = 0`** and then **`trackPosition = 250`**, then **`trackWidth = 120`**, and finally **`trackDirection = false`**. Then let's make a new function called **`makeTrack()`**. Define this function after your **`update()`** function. See the **figure2.py** listing for the code to put inside **`makeTrack()`**. The function will add one track Actor on the left and one on the right, both using the image **barrier.png**. Each time we call this function, it will add a section of track at the top of the screen.

Figure 2 The `makeTrack()` function. This creates two new Actors with the barrier image at the top of the screen

>STEP-14

On the move

The next thing that we need to do is to move the sections of track down the screen towards the car. Let's write a new function called **`updateTrack()`**. We will call this function in our **`update()`** function after

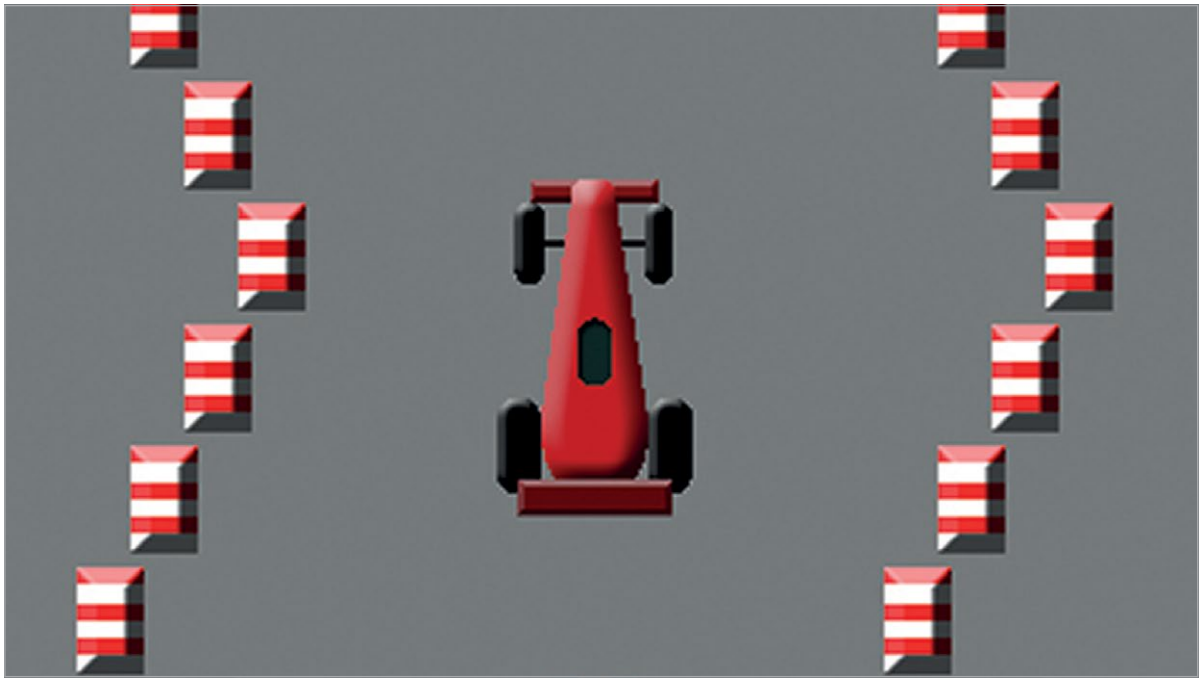
Figure 3 The `updateTrack()` function. Notice the constant SPEED – we need to define this at the top of our program, perhaps starting with the value 4

figure3.py

```
01. def updateTrack(): # Function to update where the track
    blocks appear
02.     global trackCount, trackPosition, trackDirection,
    trackWidth
03.     b = 0
04.     while b < len(trackLeft):
05.         trackLeft[b].y += SPEED
06.         trackRight[b].y += SPEED
07.         b += 1
08.     if trackLeft[len(trackLeft)-1].y > 32:
09.         if trackDirection == False: trackPosition += 16
10.         if trackDirection == True: trackPosition -= 16
11.         if randint(0, 4) == 1: trackDirection = not
    trackDirection
12.         if trackPosition > 700-trackWidth:
    trackDirection = True
13.         if trackPosition < trackWidth: trackDirection =
    False
14.         makeTrack()
```

CHANGING THE TRACK WIDTH

You can make the game easier or harder by changing the `trackWidth` variable to make the track a different width.



Right The race car with barriers making up a track to stay within. The track pieces are created by random numbers so each play is different

Figure 4 The `draw()` function and the `update()` function with conditions (`if` statements) to do different things depending on the value of `gameStatus`

we do the keyboard checks. See the **figure3.py** listing for the code for our **`updateTrack()`** function. In this function we are using **`randint()`**. This is a function that we must load from an external module, so at the top of our code we write **`from random import randint`**. We use this function to make the track curve backwards and forwards.

figure4.py

```
01. def draw(): # Pygame Zero draw function
02.     global gameStatus
03.     screen.fill((128, 128, 128))
04.     if gameStatus == 0:
05.         car.draw()
06.         b = 0
07.         while b < len(trackLeft):
08.             trackLeft[b].draw()
09.             trackRight[b].draw()
10.             b += 1
11.     if gameStatus == 1:
12.         # Red Flag
13.
14.     if gameStatus == 2:
15.         # Chequered Flag
16.
17. def update(): # Pygame Zero update function
18.     global gameStatus , trackCount
19.     if gameStatus == 0:
20.         if keyboard.left: car.x -= 2
21.         if keyboard.right: car.x += 2
22.         updateTrack()
23.     if trackCount > 200: gameStatus = 2 # Chequered
    flag state
```

>STEP-15

Making more track

Notice at the bottom of the **`updateTrack()`** function, there is a call to our **`makeTrack()`** function. This means that for each update when the track sections move down, a new track section is created at the top of the screen. We will need to start this process off, so we will put a call to **`makeTrack()`** at the bottom of our code. If we run our code at the moment, we should see a track snaking down towards the car. The only problem is that we can move the car over the track barriers and we want to keep the car inside them with some collision detection.

>STEP-16

A bit of a car crash

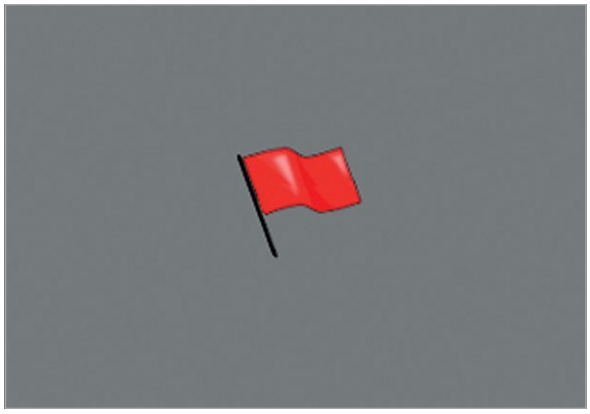
We need to make sure that our car doesn't touch the track Actors. As we are looking through the existing barrier Actors in our **`updateTrack()`** function, we may as well test for collisions at the same time. We can write **`if car.colliderect(trackLeft[b]) or car.colliderect(trackRight[b]):`** and then, indented on the next line, **`gameStatus = 1`**. We have not covered **`gameStatus`** yet – we are going to use this variable to show if the game is running, the car has crashed, or we have reached the end of the race. Define your **`gameStatus`** variable near the top of the program as **`gameStatus = 0`**. You will also need to add it to the global variables in the **`updateTrack()`** function.

>STEP-17

Changing state

In this game we will have three different states to the game stored in our variable **`gameStatus`**. The first or default state will be that the game is running and will be represented by the number 0. The next state will

Below Each of the barrier blocks is checked against the car to detect collisions. If the car hits a barrier, the red flag graphic is displayed



be set if the car crashes, which will be the number 1. The third state will be if we have finished the race, which we'll set as the number 2 in **gameStatus**. We will need to reorganise our **draw()** function and our **update()** function to respond to the **gameStatus** variable. See the **figure4.py** listing for how we do that.

>STEP-18

Finishing touches

All we need to do now is to display something if **gameStatus** is set to 1 or 2. If **gameStatus** is 1 then it means that the car has crashed and we should display a red flag. We can do that with the code:

screen.blit('rflag', (318, 268)). To see if the car has reached the finish, we should count how many track sections have been created and then perhaps when we get to 200, set **gameStatus** to 2. We can do this in the **update()** function as in **Figure 4**. Then, in the **draw()** function, if the **gameStatus** is 2, then we can write **screen.blit('cflag', (318, 268))**. Have a look at the full code listing to see how this all fits together.

> The next thing that we need to do is to move the sections of track down the screen

>STEP-19

Did you win?

If you didn't get the program working first time, you are not alone – it's quite rare to have everything exactly right first time. Check that you have written all the variables and functions correctly and that the capital letters are in the right places. Python also insists on having code properly formatted with indents. When it's all in place, test your program as before and you should have a racing game with a chequered flag at the end!

CHANGING THE SPEED

If you want to make the track move faster or slower, try changing the value of **SPEED** at the start of the program.

The official Pygame Zero documentation can be found at magpi.cc/fBqznh