**Python Basics**

Whitespace matters! Your code will not run correctly if you use improper indentation.

# this is a comment

**Python Logic**

if

if test:

#do stuff if test is true elif test 2:

#do stuff if test2 is true else:

#do stuff if both tests are false

while

while test:

#keep doing stuff until

#test is false

**Python Strings**

for

for x in aSequence:

# do stuff for each member of

# aSequence, for example: each

# item in a list, each character

# in a string, etc.

for x in range(10):

# do stuff 10 times (0 through 9)

for x in range(5,10):

# do stuff 5 times (5 through 9)

A string is a sequence of characters, usually used to store text.

Creation: the\_string = “Hello World!”

the\_string = ‘Hello World!’ Accessing: the\_string[4] returns ‘o’

Splitting: the\_string.split(‘ ‘) returns [‘Hello’, ‘World!’]

the\_string.split(‘r”) returns [‘Hello Wo’, ‘ld!’]

To join a list of strings together, call join() as a method of the string you want to separate the values in the list (‘’ if none), and pass the list as an argument. Yes, it’s weird.

words = [“this”, ‘is’, ‘a’, ‘list’, ‘of’, “strings”]

‘ ‘.join(words) returns “This is a list of strings”

‘ZOOL’.join(words) returns “ThisZOOLisZOOLaZOOLlistZOOLofZOOLstrings”

‘’.join(words) returns “Thisisalistofstrings”

String Formatting: similar to printf() in C, uses the % operator to add elements of a tuple into a string

this\_string = “there”

print “Hello %s!” % this\_string Returns “Hello there!”

**Python Tuples**

A tuple consists of a number of values separated by commas. They are useful for ordered pairs and returning several values from a function.

Creation: emptyTuple = ()

singleItemTuple = (“spam”,) # note the comma!

thistuple = 12, 89, ‘a’

thistuple = (12, 89, ‘a’)

accessing: thistuple[0] returns 12

**Python Dictionaries**

A dictionary is a set of key:value pairs. All keys in a dictionary must be unique.

Creation: emptyDict = {}

thisdict = {‘a’:1, ‘b’:23, ‘c’:”eggs”}

accessing: thisdict[‘a’] returns 1 deleting: del thisdict[‘b’]

finding: thisdict.has\_key(‘e’) returns False thisdict.keys() returns [‘a’, ‘c’] thisdict.items() returns [(‘a’, 1), (‘c’, ‘eggs’)]

‘c’ in thisdict returns True

‘thisisnotthere’ in thisdict returns False

**Python List Manipulation**

One of the most important data structures in Python is the list. Lists are very flexible and have many built-in control functions.

**Operation Syntax Return New Value**

Create thelist = [5,3,‘p’,9,‘e’] No return value [5,3,‘p’,9,‘e’]

Accessing thelist[0] 5 Unchanged

Slicing thelist[1:3] thelist[2:] thelist[:2] thelist[2:-1]

[3, ‘p’]

[‘p’, 9, ‘e’] [5, 3]

[‘p’, 9]

Unchanged Unchanged Unchanged Unchanged

Length len(thelist) 5 Unchanged

Sort thelist.sort() No return value [3,5,9,’e’,’p’]

Add thelist.append(37) No return value [3,5,9,’e’,’p’,37]

Return and Remove thelist.pop() 37 thelist.pop(1) 5

[3,’z’,9,’p’] [‘z’,9,’p’]

Insert thelist.insert(2, ‘z’) No return value [3,’z’,9,’e’,’p’]

Remove thelist.remove(‘e’)

del thelist[0]

No return value

No return value

[3,’z’,9,’p’] [’z’,9,’p’]

Concatenate thelist + [0] [’z’,9,’p’,0] [’z’,9,’p’]

Finding 9 in thelist True Unchanged

**List Comprehension**

A special expression enclosed in square brackets that returns a new list. The expression is of the form: [expression for expr in sequence *if condition*]. The condition is optional.

>>> [x\*5 for x in range(5)] [0, 5, 10, 15, 20]

>>> [x for x in range(5) if x%2 == 0] [0, 2, 4]

**Python Function Definition**

Function:

def myFunc(param1, param2):

“””By putting this initial sentence in triple quotes, you can

access it by calling myFunc. doc ”””

#indented code block goes here

spam = param1 + param2 return spam

Python - Sticker Collection

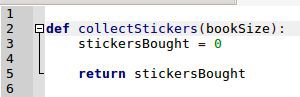
Have you ever collected stickers? I remember collecting Premier Leagure 94/95 stickers and swapping them with friends. We would invariably end up with a great big wad of ‘swaps’ and spend lunch and break time hunting for ones we did not have in our friends swap piles.

After the sticker collection got so far, I would invariably end up with 4 or 5 copies of the same people. Do the sticker companies do this deliberately? Are the stickers truly random? Let us experiment using code to find out.

Defining the function

We will define a function to simulate a single person collecting an entire sticker book 1 sticker at a time. This is obviously not as efficient as multiple people collecting and swapping, but it should illustrate that getting several copies of the same sticker is actually very likely.

This function will accept 1 argument called **bookSize** and return the number of stickers bought. The skeleton of the function will look like this.



Create a Dictionary

In order to track how many copies of each sticker we have, I will create a **dictionary**.

The documentation for a dictionary can be found here <https://docs.python.org/3.7/tutorial/datastructures.html#dictionaries>

Dictionaries (also called maps and associative arrays) are used to store key value pairs.

Our stickers dictionary will start out empty, then as each numbered sticker is collected, the sticker number will be used as a key, and the count will be used as the value. For example if I buy a packet of stickers with the numbers 1, 45, 60 and 20 then my dictionary will look like this.

|  |  |
| --- | --- |
| Sticker Number (key) | Count (value) |
| 1 | 1 |
| 45 | 1 |
| 60 | 1 |
| 20 | 1 |

If I buy a second packet, which contains the numbers 2, 45, 78 and 100 then my dictionary will

look like this.

|  |  |
| --- | --- |
| Sticker Number (key) | Count (value) |
| 1 | 1 |
| 2 | 1 |
| 45 | 2 |
| 60 | 1 |
| 78 | 1 |
| 20 | 1 |
| 100 | 1 |

Note that I have my first swap! I have collected sticker 45 twice and so the count is pushed up to

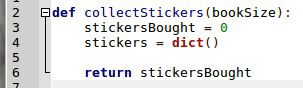
2.

The collection will be considered complete when the number of keys in the dictionary is equal to the number of stickers available. As an example for a 5 sticker book (very small book!), the dictionary might look like this.

|  |  |
| --- | --- |
| Sticker Number (key) | Count (value) |
| 1 | 5 |
| 2 | 10 |
| 3 | 1 |
| 4 | 7 |
| 5 | 3 |

So I have a lot of swaps (particularly sticker 2) and only one of sticker 3. Presumably sticker 3 was the last sticker I was waiting for. So our algorithm will complete when the dictionary size equals the sticker book size.

Add a dictionary called **stickers** to our function as shown in line 4.



Loop Collecting Stickers

We will use a **while** loop to check if the dictionary size is equal to the book size. The loop will continue *buying stickers* until the collection is complete. We should count each sticker bought by incrementing the **stickersBought** variable.



Generate Random Stickers

We are going to make the sticker selection completely random, so we will need the **randint**

function from the **random** library in Python. Add the following import to the top of your code.

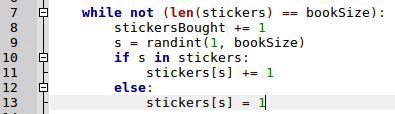


Now that we have access to the random number library, we should use it to *buy a sticker*. Line 9 will choose a random number from 1 to **bookSize** and put it into the variable **s**.



We must now use this randomly selected sticker to mark our dictionary. If we haven’t collected this sticker before, we should create a new entry in our dictionary with a count of 1. If we **have** seen this sticker before then we should increment the count already there.

Lines 10 through to 13 show how this works. Notice the use of the **if <> in <>** statement.



That should be enough, let us have a look at the completed function.

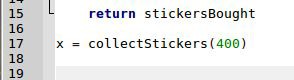


Call our Function

If we run this source code now, nothing will happen. We have written our function to simulate

the collection of stickers, but we still need to **invoke** it. Since we are now adding code outside of the function, the indentation should go back to the left hand edge of the text file.

Add line 17 as shown



If you run this code, it will call our function, but not print anything. So we will now add a print to see what happened!



Run your code again and hopefully you will see something like this. This screenshot is of me running the code from the terminal in Ubuntu Linux.



Let’s zoom in



Put sticker size in variable

Those last two lines of code use the value 400 twice. I would rather put that into a variable to ease maintenance.

● Create a new variable called **myBookSize** and set it to 400.

● Use the variable in the call to **collectStickers**

● Edit the format string and format arguments to pass in the **myBookSize** variable. The code should end up like this.



Run it again and check it still works.

Try Yourself

Running this simulation once gives you a very small sample of how many stickers you are likely to need to complete a single collection. It would be better to call the function many times and take an average of the number of stickers bought.

Call the **collectStickers** function in a **for x in range(y)** type of loop and add up the total number of stickers bought. Then divide that number by the number of attempts (shown as **y** here) to get the average. Then print the average.

I will let you figure out the code for this step. You should have all the tools you need, good luck.

Python Sticker Collection

Object Oriented Version

In this exercise we will be using classes to encapsulate functions and their data. We will be writing a simulation of sticker collecting. My interest in this simulation came from the following question:

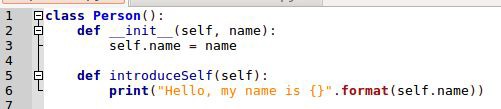
“Do they deliberately print a smaller number of some stickers to keep you buying them?”

By simulating the sticker collection on a computer, we can see how the collection stacks up with a truly random sticker printing process. I believe this program will demonstrate that there is no need for the sticker printers to make some ‘rarer’ than others, because even with a completely fair and even distribution, you will end up with many swaps, and many swaps of the same old faces.

Object Oriented Programming

We will be using the **class** keyword in Python to define our program. Classes are a means by which a programmer can group up a number of variables in a structure, and give functions to that structure which operate on a particular instance. If that is not clear (and who could blame you) let us look at an example.

If we wanted to simulate people introducing themselves, we could write a class called **Person**. The **Person** class might look like this.



Let us break that down line by line.

Line 1 is the *class declaration*. It tells Python that we want to create a new type named **Person**. Any code indented within the Person is assumed to belong to that class.

Line 2 is the start of the *constructor*. This is the function that is called when we create *instances*

of the **Person** class. This function has a special name that Python recognises **init** . The first

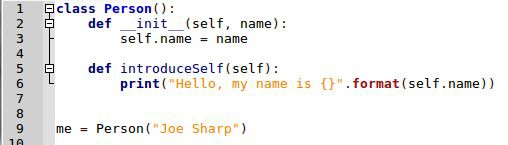
parameter is **self** which is a reference to the *instance* of the class. We will see that in action later. The second parameter is going to be the **name** of the person.

Line 3 shows us setting the variable **name** on our instance of the **Person**. Each **Person** will have a **name**. This belonging of variables is part of what makes a class powerful.

Line 5 is our own custom function called **introduceSelf**. It requires no actual parameters, but because it is a *class member function* it has to have the parameter **self**.

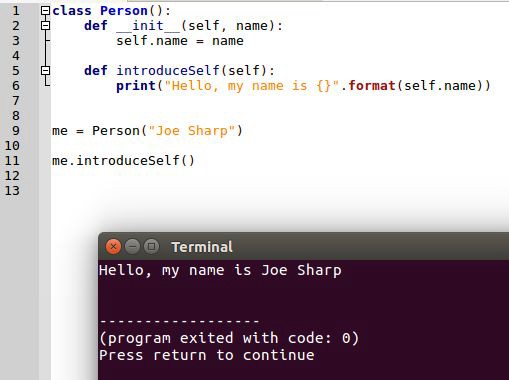
Line 6 is the implementation of our **introduceSelf** function. In this function we will simply print a greeting from the Person. Note the use of the format function on the string.

Now that we have defined a class, let us create *instances*. Here is how we create a single instance of Person.



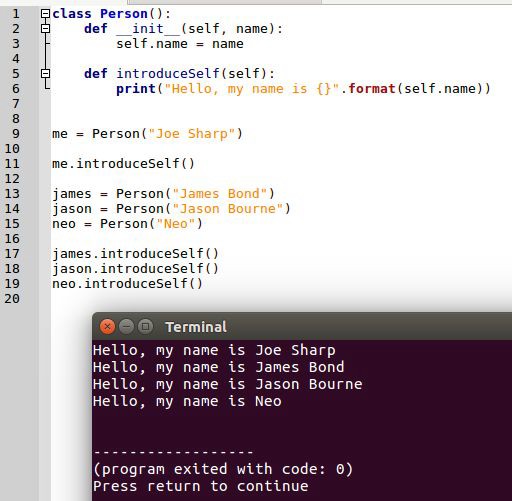
Line 9 shows the creation of a single instance of person and assigns it to a variable called **me**. Notice that we only passed in the **name** of the Person. We do not have to pass in **self** because Python understands this is a class and does that for us.

Now that we have created an instance we can call the **introduceSelf** function on the instance. How do we do this? See Line 11 below:



You can now run this code and you should see the print out. Again note that we do not have to pass **self** into the **introduceSelf** function, this is done by Python because we have called the *method* on our *instance*.

Now that we have created a single instance let’s create some more and call upon them to introduce themselves.



I have created 4 people now and called the **introduceSelf** function on each one. Notice that each instance uses its own name in the print out.

Sticker Book Design

Now that we have had a brief introduction to classes, we shall design our sticker book code. The first thing we will address is deciding which classes we want to create. I have decided on the following classes.

● Sticker Book - This will contain the following member variables

○ Name - the name of the collection (such as ‘Pokemon’ or ‘World Cup 2014’)

○ Collection Size - The total number of sticker available in the book

○ Packet Size - The number of stickers in a single packet.

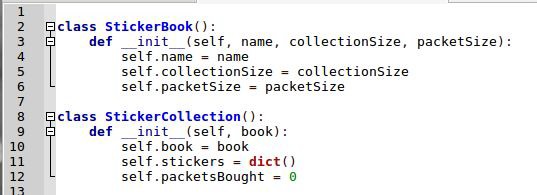
● Sticker Collection

○ Book - The collection will be of a single book. So the book shall be passed in so that the collection can refer to it.

○ Stickers - The list of stickers collected so far

○ Packets Bought - The number of packets bought for this collection

That should do for now. The following code declares these classes.



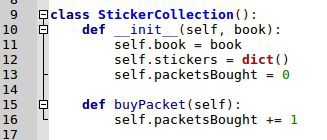
The **StickerCollection** class uses the **book** passed into the constructor. It then creates an empty **dictionary** and initialises the count of **packetsBought** to zero.

Now that we have our classes, let us write a *function* for the **StickerCollection** class. We will write a function called **buyPacket**. This will randomly choose a number of stickers to receive in

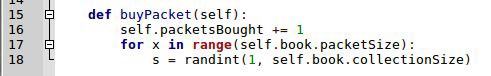
a packet (according to the packet size) and put that data into our dictionary. First we will need to import the **randint** function from the **random** library. Do this as shown in Line 1:



Now add the member function to the StickerCollection class as shown in lines 15-16.



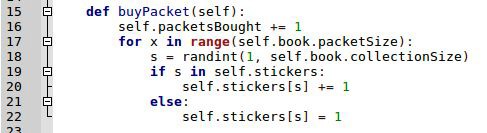
So far our function just increments the number of packets bought. Now we need to buy as many stickers as there are in a packet. We will use a **for loop** and the **packetSize** variable from the **book** tied to the collection. The code looks like this.



Line 17 shows the loop, and how we are accessing **packetSize** on our **book** *member variable.*

Line 18 shows us choosing a random sticker between 1 and the collection size of our book.

Now that we have chosen a sticker, we will record it in our **stickers** dictionary that we created earlier.



At the start of our collection, the **self.stickers** dictionary is empty. So there are no keys ‘in it’. As we collect more stickers we will count how many we have of each number so we can track our swaps. Once we have a sticker for every number in the collection we will assume the collection

is complete.

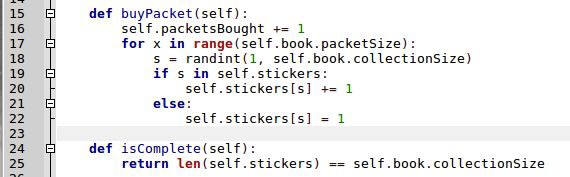
Line 19 - This shows how we determine if a key is in a dictionary. Line 20 - If the value exists in the dictionary we increment it by 1

Line 21 - The else, this means the sticker we have just bought it not already in our collection

Line 22 - If it is a new sticker, we simply count the first 1 of that number in our dictionary.

If we ran the **buyPacket** function repeatedly. Eventually the dictionary would contain every sticker in the collection and we would be finished.

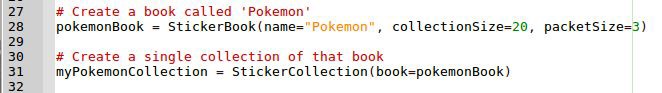
We will now write a *member function* for the **StickerBook** class that we can use to check if the collection is complete. This function is shown in lines 24 and 25 below.



I am using the length of the stickers dictionary and the collection size to check if the collection is complete. Since we are using a dictionary, when we get swaps it does not increase the number of keys in the dictionary. The keys are unique for each sticker number.

We now have enough code to complete a collection. We will write some code to create a sticker book, then create a collection, and cycle through purchasing stickers until the collection is complete.

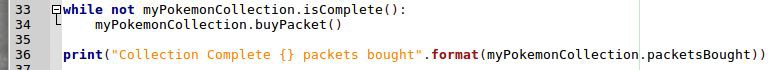
Note the indentation is back against the margin, this code exists outside either of our class definitions.



Line 28 creates the Sticker Book, I am using named parameters here so a developer can see the purpose of the various numbers.

Line 31 creates a single collection against that book. The cool thing about the encapsulation is that we could create multiple collections of the same book to simulate a group of friends collecting the same stickers.

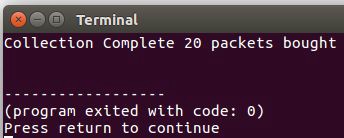
Let us now create a loop for buying the stickers.



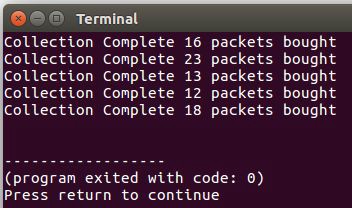
Note that I am calling **isComplete** on our instance of the sticker book called

**myPokemonCollection**. While the collection is incomplete, keep buying stickers.

Line 36 shows us printing how many packets were bought by the end of the collection. Here is an example of that running.



If you run this code several times you will get different numbers each time, that is the luck of the draw. Try adding a loop so it runs the entire collection simulation 5 times. Your print should look like this.



Python - Sticker Collection

Continued from Procedural

This exercise assumes that you have completed the Python Sticker Collection - Procedural Version. If you have not yet done that, then go back and complete it first. This tutorial literally picks up from where you left off.

Create New Source File

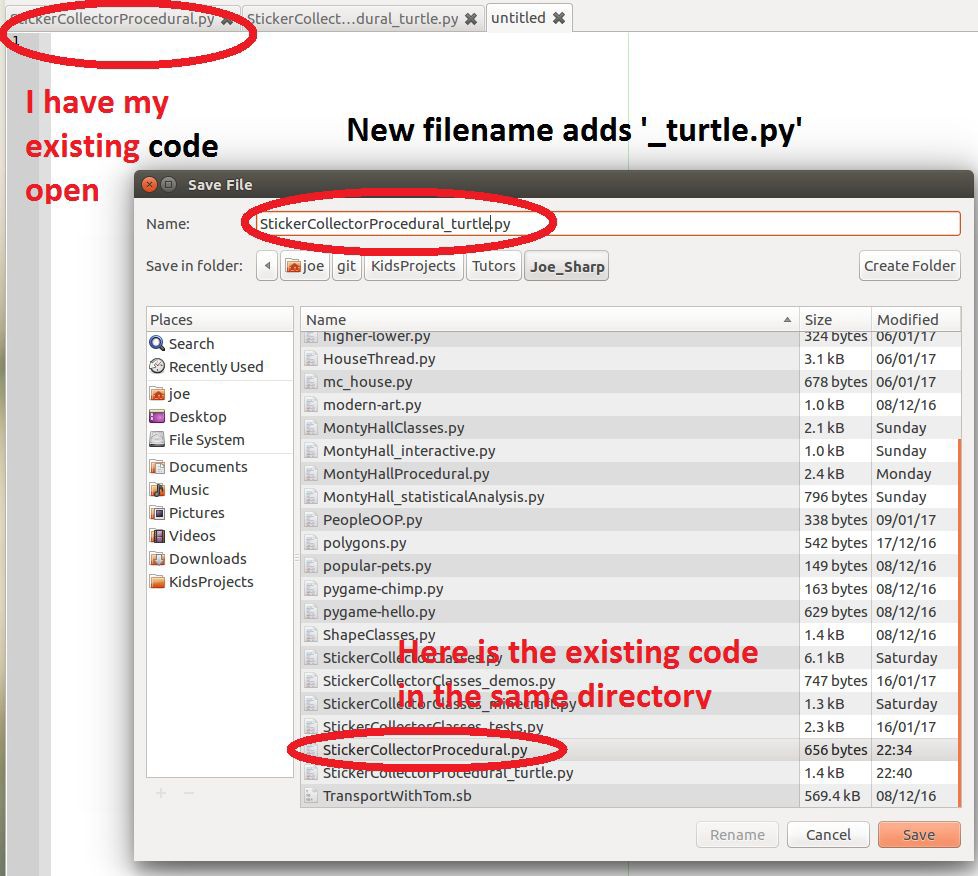
I am assuming you have your existing Sticker Collection code saved to a python file. We are going to write another python file and **import** the existing one. Our existing python file will now become a *library*.

Create a new file in geany, and save it into the same directory as your existing sticker collection python code. Give it a different name. For example:

Myy original file is called **StickerCollectorProcedural.py**

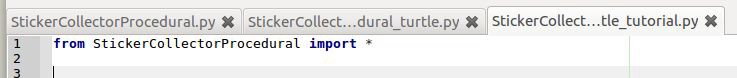
My new file will be called **StickerCollectorProcedural\_turtle.py**

When I save my new file, it looks like this:



Import the Existing Source File

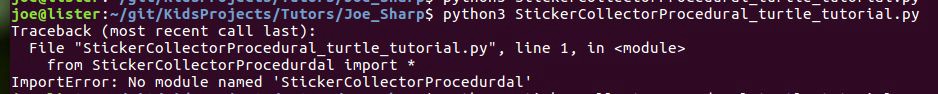
You are now editing the new source file, add an import statement that looks like this (your filename may vary)



Test the Import

Before we go any further, let’s test the import process. Run the code by using F5 or by invoking it from the terminal.

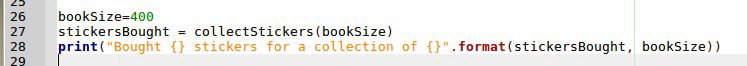
If the import is not working you will see an error like this:



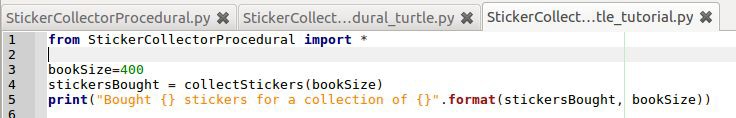
After I corrected the spelling error it looked like this:



This shows that the **new** source file has successfully imported the **old** source file. It has run the sticker simulation using the code in the **old** source file. We will now move the lines shown:



Cut and Paste them into your **new** source file (the one with the import).



**TEST YOUR CODE:** If you run the **new** source file, the test should run. Note that if you run the

**old** code nothing will happen. It defines a function but the **old** code will no longer call it.

Allow Observer Function

Our turtle drawing will fill in the stickers as we collect them, for this to work we must intercept the code where the stickers are collected. How do we do this? We pass a function into the **collectStickers** function. I will call this function the **observer**.

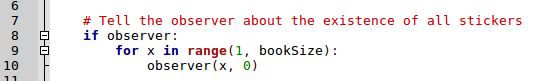
Edit the **collectStickers** function as shown, we have just added a parameter called **observer**

and given it a default value of **None**.

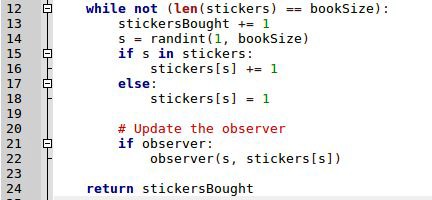


We should tell the observer about all of our stickers at the start, add the code shown from lines

8 to 10 before the main collection loop.



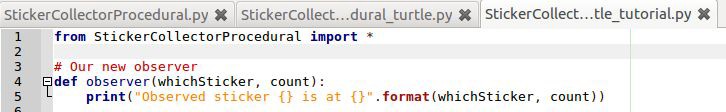
Now we need to intercept the stickers each time they are collected. This will happen inside the main collection loop. Lines 20 to 22 show the call to the observer.



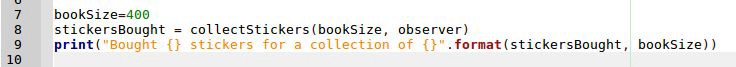
**TEST YOUR CODE**: Run the **new** source file and check that you haven’t broken anything. The test should still run. We have not provided an observer yet, we will do this in the next section.

Write Observer Function

The **collectStickers** function now allows us to provide an observer. We will now write one and pass it in. Edit your **new** source file by adding the function shown from lines 4 to 5 (3 is just a comment).

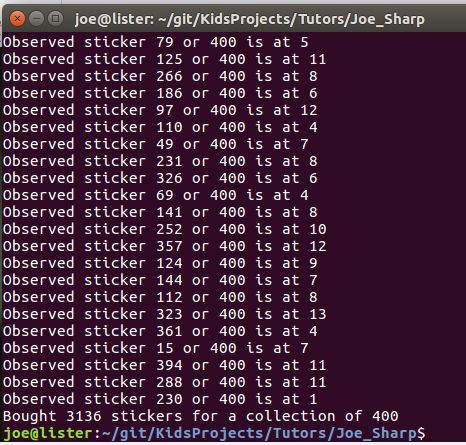


We will now pass this function into the **collectSticker** function as shown on line 8.



When you run this code, it should collect the stickers as before, but this time each time a sticker is collected the observer function will be called. This will result in a print statement as per line 5 above.

**TEST YOUR CODE**: Run the new source file and it should look something like this:



Can you see how our observer function is being called? Time to draw something!

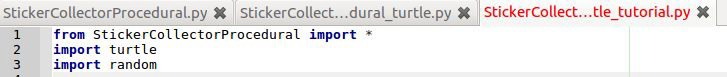
Write a Create Turtle function

We will create a single turtle for each sticker that we need to collect.

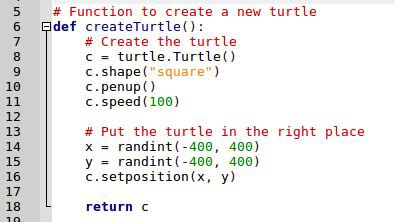
● When the sticker is first created it with a count of zero, we will make the turtle RED.

● When a sticker is collected (count > 0) we will make it GREEN

● As we get duplicates of a sticker, we will increase the SIZE of that turtle. Import the turtle library and the random library (lines 2 and 3)



The code below shows our new turtle function from line 6 to 18



I am placing each turtle in a random location within the game area.

I am calling penup() to stop the turtle drawing lines as it moves from the centre to its location. I am setting the speed so that the animation is quick.

The function returns the created turtle, we will be calling this from our observer. Add the following lines to the end of the code just to test that the function works.

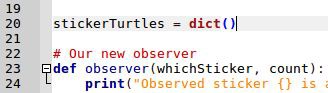


**TEST YOUR CODE**: Run the code and it should run the observed test as before, then create a random single turtle (square shape) and leave the window open.

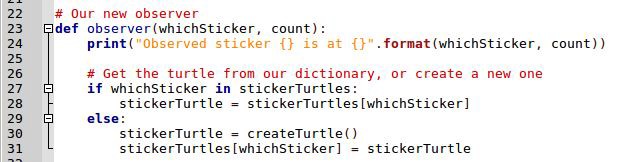
Create a Turtle for Each Sticker

In this section we will create a **dictionary** of turtles. The keys will be the sticker numbers, the values will be the turtles created by our **createTurtle** function.

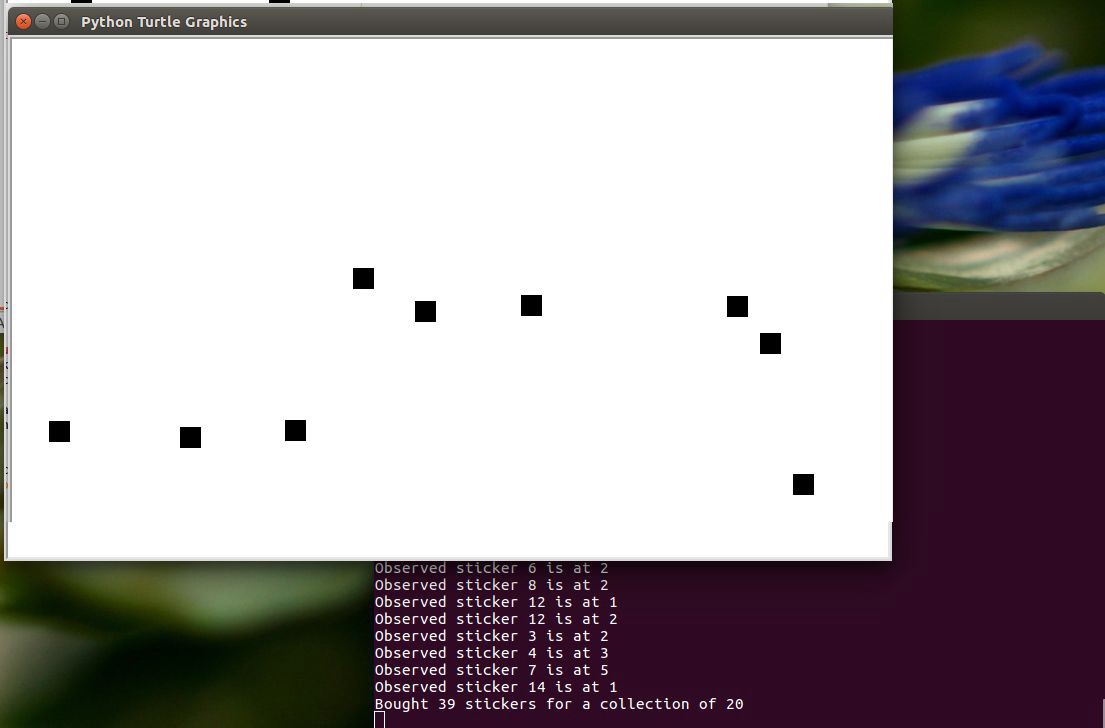
Add the following lines above the **observer** function (shown here as line 20)



Add the lines 27 to 31 to your observer function.

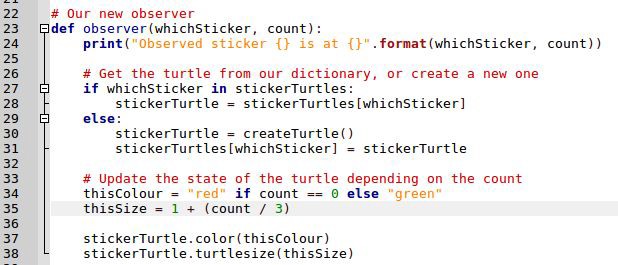


**RUN YOUR CODE**: This should now create a turtle for each sticker that is first received. At this stage we are just plonking them onto the screen and leaving them there. When the code has run it should look a bit like this (I have shortened the turtle window here):



Modify Turtles as Stickers are Collected

We will be modifying the turtles colour and size as the stickers are collected. Lines 34 to 38 show how I am doing this.



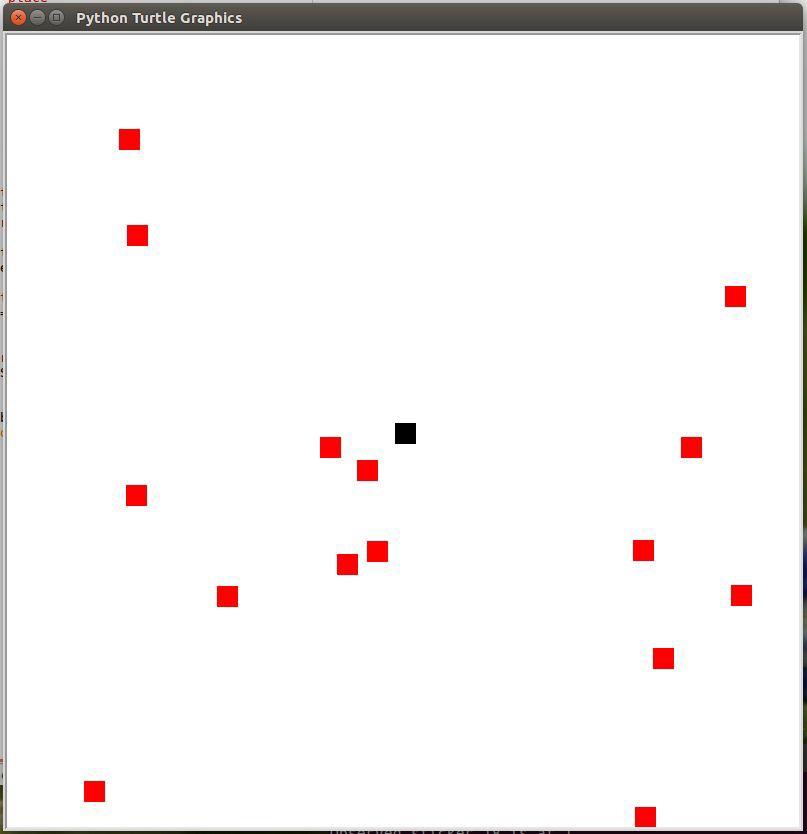
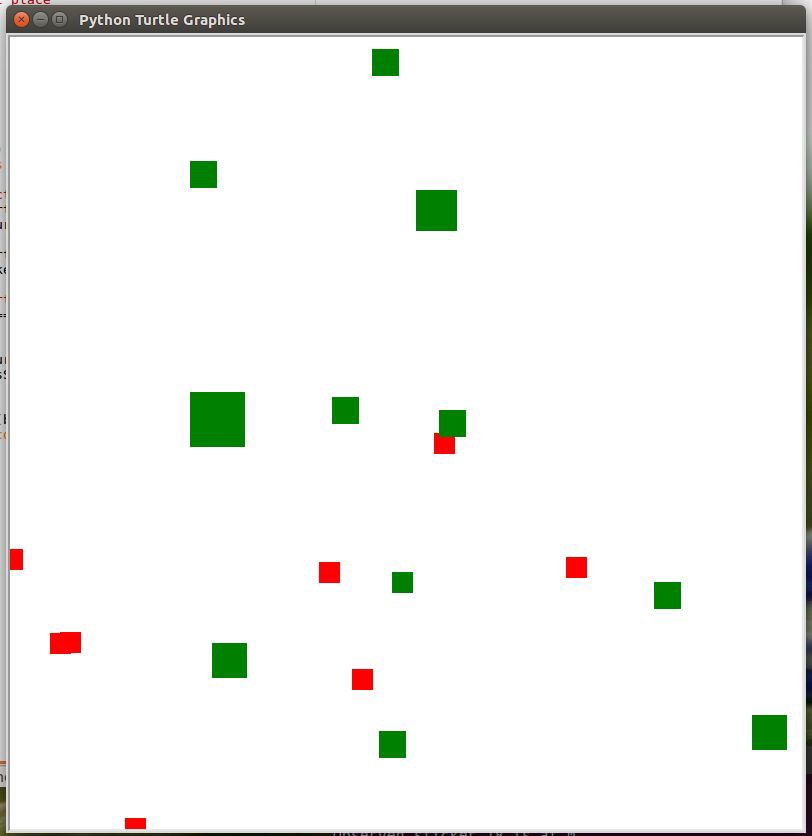
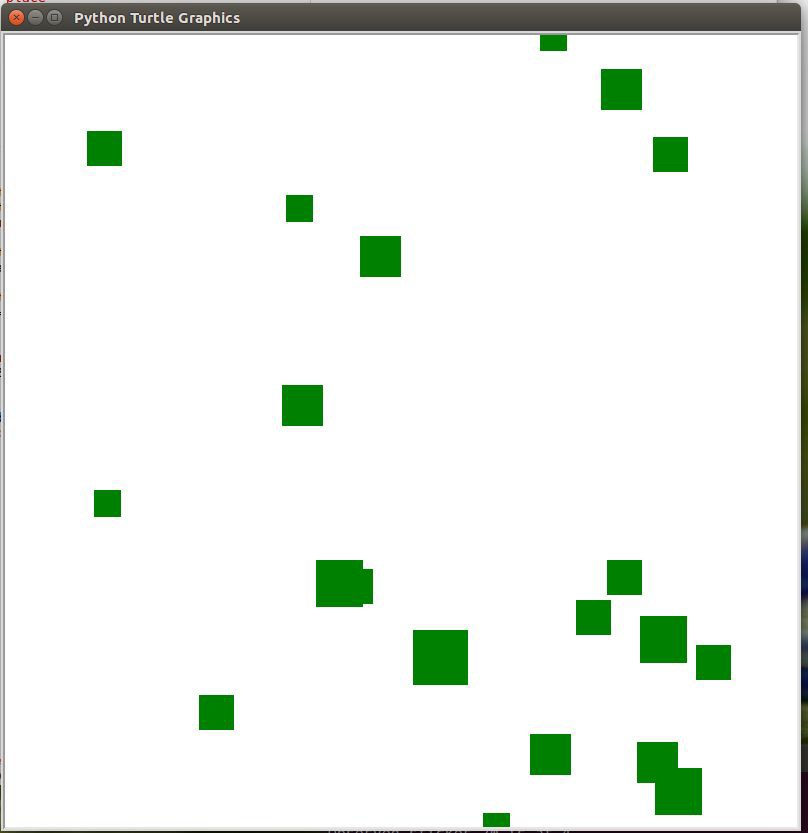
We populated the **stickerTurtle** variable from the dictionary. Creating a new turtle if required. Line 34 shows us calculating the colour by using the count. Sticker we have yet to collect are

RED, collected stickers are GREEN.

Line 35 shows us calculating the size of the turtle. It must be at least 1, I then divide the count by 3 to prevent high counts drowning out the display.

Lines 37 and 38 show us setting the color and size of the turtle. Simples.

**TEST YOUR CODE**: The project is now finished, when you run your code it should end up looking like this:



**CONGRATULATIONS, YOU HAVE FINISHED THE STICKER COLLECTOR TURTLE**

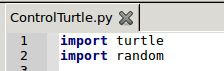
Python - Turtle Power

In this project we will be learning how to respond to key presses when using the Turtle module in Python.

Create Turtle

We must import the turtle library, we will also be using the random library so we can generate random movement of non player controlled turtles.

Add the following imports to a new python file.



We will be creating an instance of the turtle to be controlled by the player. For this create a variable named **player** and assign a new Turtle object to it.



We also need access to the screen so we can set-up key listening events and the game loop. Add the following code to get the Screen in a variable called **w**.

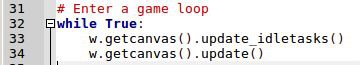


Set-up Listening and Game Loop

We will now write the last section of our code, first we will turn on key listening.



We will now write our Game Loop. This will go round forever (or at least until the user quits) and handle any tasks we may wish to run in addition to the key bindings



This code must stay *AT THE END* of your python file. We are about to start putting in event handlers, these will go between the setup and listening code. Watch out for that!

Up Arrow Goes Forward

We will be using the **onkey** function on the Screen to listen for a keypress. We must first write a function to call when the **Up Arrow** is pressed. Pay attention to *indentation*!



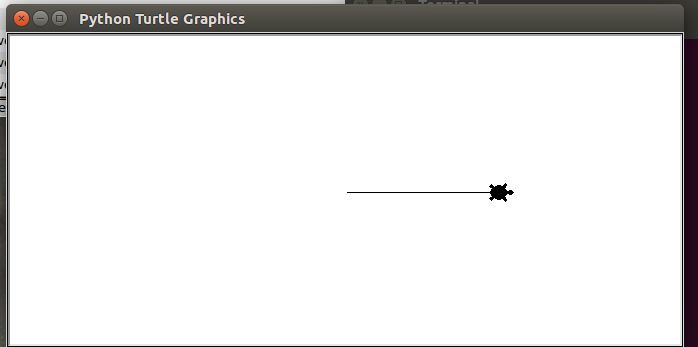
Now we can tell the screen object about our key handler using the code in line 13 shown:



The “Up” refers to the up arrow, a complete list of available key presses can be found here <https://www.tcl.tk/man/tcl8.4/TkCmd/keysyms.htm>

Test

Press F5 (assume using Geany) to play your new game, you should be able to go forward by repeatedly pressing the Up arrow.



Handle Turning

Now that we know how to respond to key presses, write a function to turn left as shown:



Attach a key handler to this by adding line 17 shown



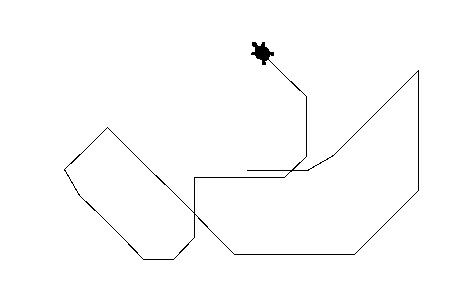
Test your code, can you turn left?

For working out yourself

Can you write the function and key binding for turning right?

Test again

Now that you have turning left and right working, play with the turtle to check it all works. Here is my turtle after a quick spin round.



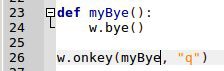
Handle Quitting

At the moment you have to close the window yourself to finish the program. It would be better if we provided a key handler for quitting. I am going to use the letter **q** to force a quit.

Add the following function to handle the quit command.



To call this function when **q** is pressed, add line 26 shortly after the new function.



Test Quitting

Play with the turtle again, press **q** to see if your program exits correctly.

Computer Controlled Turtles

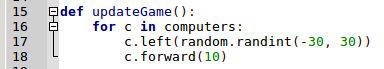
Now we are going to add the ability to spawn computer controlled turtles when the player presses the **spacebar**. Just for extra credit we will make the computer turtles random colours.

Create a list of turtles by adding lines 9 and 10, these go after the player creation.



We will now create a function that gets called for every frame of the game. I will call it

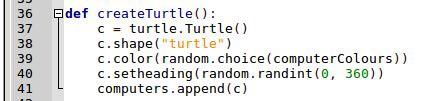
**updateGame** and it looks like this. Put it just after the definition of the computers list.



This will loop through all the turtles in the **computers** list and change their direction (randomly)

and then move them forward a bit.

How to we create these turtles? We will add a function called **createTurtle** and call it whenever the user presses the **spacebar**. The **createTurtle** function looks like this.

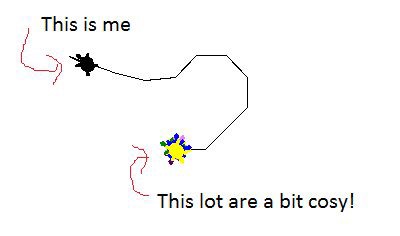


Then add the key handler as shown.

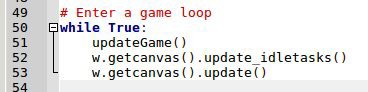


Test Code - Turtles don’t move?

Run the code now and press **spacebar** a few times. You should find the program creates turtles in the centre, points them in a random direction but *they do not move*.



We need to invoke the **updateGame** function in our core game loop. The game loop should be at the end of the code, so add the code shown here as line 51.



Test Again - Randomly Moving Coloured Turtles?

Run the code again and press **spacebar** a few times. You should find that the **updateGame** function is being called in our loop and the computer controlled turtles will spider out from the centre.

You now know how to respond to user key presses and also run computer controlled code during the game loop. What else can you add to this program?

CONGRATULATIONS - YOU HAVE FINISHED THE TURTLE POWER TUTORIAL

**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](http://technovisualeducation.co.uk/) [twitter.com/mindexplorers](http://twitter.com/mindexplorers)

GET STARTED WITH

**PYGAME ZERO**

Pygame Zero is a great choice for anyone who wants to start writing

computer games on the Raspberry Pi

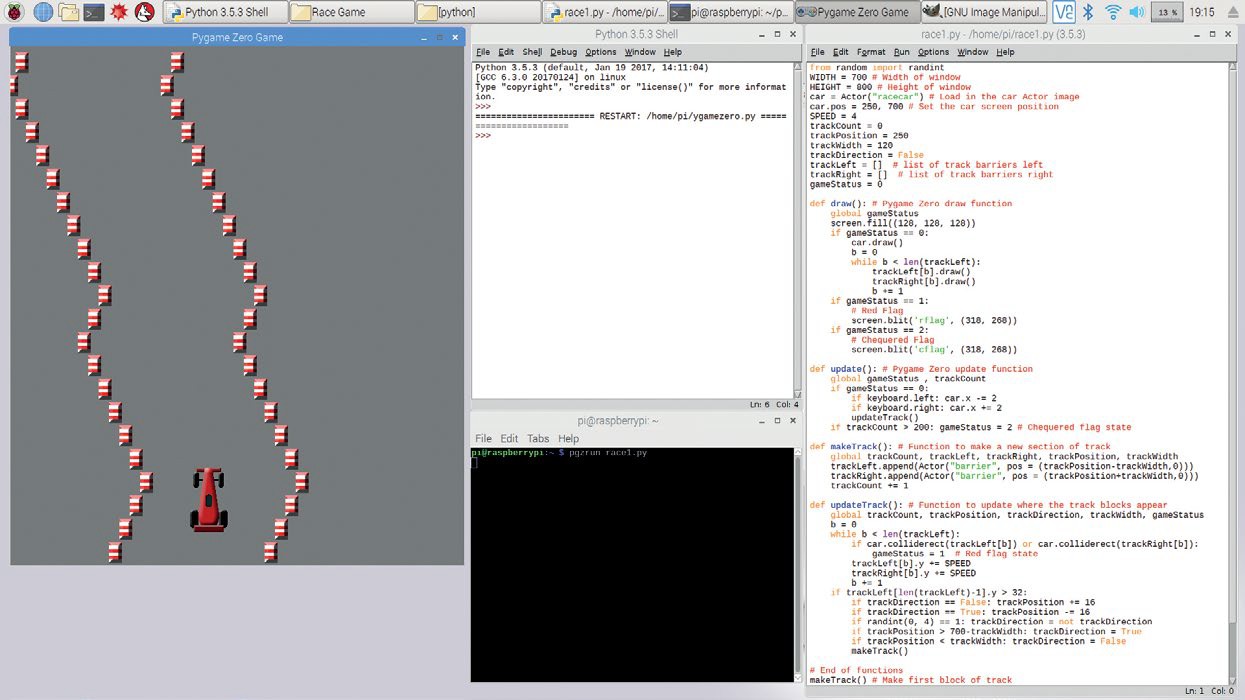
> Raspbian Jessie or newer

> An image manipulation program such as GIMP

> A little imagination

> A keyboard

**The Python Shell window that appears when**



**we open IDLE**

f you’ve done some Python coding and wanted

**I** 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.



**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**

**GET STARTED WITH PYGAME ZERO**

race1.py

Tutorial

**>PYTHON**

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

01.

02.

03.

04.

from random import randint WIDTH = 700 # Width of window HEIGHT = 800 # Height of window

car = Actor("racecar") # Load in the car Actor image

**DOWNLOAD:**

[magpi.cc/VcqutR](http://magpi.cc/VcqutR)

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.

05.

06.

07.

08.

09.

10.

11.

12.

13.

14.

15.

16.

17.

18.

19.

20.

21.

22.

23.

24.

25.

26.

27.

28.

29.

30.

31.

32.

33.

34.

35.

36.

37.

38.

39.

40.

41.

42.

43.

44.

45.

46.

47.

48.

49.

50.

51.

52.

53.

54.

55.

56.

57.

58.

59.

60.

61.

62.

63.

64.

car.pos = 250, 700 # Set the car screen position

SPEED = 4

trackCount = 0

trackPosition = 250

trackWidth = 120

trackDirection = False

trackLeft = [] # list of track barriers left

trackRight = [] # list of track barriers right

gameStatus = 0

def draw(): # Pygame Zero draw function global gameStatus

screen.fill((128, 128, 128))

if gameStatus == 0:

car.draw()

b = 0

while b < len(trackLeft):

trackLeft[b].draw()

trackRight[b].draw()

b += 1

if gameStatus == 1:

# Red Flag

screen.blit('rflag', (318, 268))

if gameStatus == 2:

# Chequered Flag

screen.blit('cflag', (318, 268))

def update(): # Pygame Zero update function global gameStatus , trackCount

if gameStatus == 0:

if keyboard.left: car.x -= 2

if keyboard.right: car.x += 2

updateTrack()

if trackCount > 200: gameStatus = 2 # Chequered flag

state

def makeTrack(): # Function to make a new section of track global trackCount, trackLeft, trackRight, trackPosition,

trackWidth

trackLeft.append(Actor("barrier", pos = (trackPosition-

trackWidth,0)))

trackRight.append(Actor("barrier", pos =

(trackPosition+trackWidth,0)))

trackCount += 1

def updateTrack(): # Function to update where the track blocks appear

global trackCount, trackPosition, trackDirection, trackWidth, gameStatus

b = 0

while b < len(trackLeft):

if car.colliderect(trackLeft[b]) or

car.colliderect(trackRight[b]):

gameStatus = 1 # Red flag state

trackLeft[b].y += SPEED

trackRight[b].y += SPEED

b += 1

if trackLeft[len(trackLeft)-1].y > 32:

if trackDirection == False: trackPosition += 16

if trackDirection == True: trackPosition -= 16

if randint(0, 4) == 1: trackDirection = not

trackDirection

if trackPosition > 700-trackWidth: trackDirection =

True

if trackPosition < trackWidth: trackDirection =

False

makeTrack()

# End of functions

makeTrack() # Make first block of track

[raspberrypi.org/magpi](http://www.raspberrypi.org/magpi)

July 2018 53

July 2018 53

figure1.py

**>STEP-07**

**All the world’s a stage**

The screen object used in Step 5 is a predefined

01.

02.

03.

04.

05.

WIDTH = 700

HEIGHT = 800

def draw():

screen.fill((128, 128, 128))

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

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,

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

Actors in Pygame Zero are

dynamic graphic objects, much the same as sprites

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).

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**](http://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

figure2.py

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

01.

02.

03.

04.

05.

def makeTrack(): # Function to make a new section of track global trackCount, trackLeft, trackRight,

trackPosition, trackWidth trackLeft.append(Actor("barrier", pos =

(trackPosition-trackWidth,0)))

trackRight.append(Actor("barrier", pos = (trackPosition+trackWidth,0)))

trackCount += 1

**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

**>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.

**>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

figure3.py

Figure 2

**The makeTrack() function. This creates two new Actors with the barrier image**

**at the top of the screen**

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**

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.

01.

02.

03.

04.

05.

06.

07.

08.

09.

10.

11.

12.

13.

14.

def updateTrack(): # Function to update where the track blocks appear

global trackCount, trackPosition, trackDirection, trackWidth

b = 0

while b < len(trackLeft): trackLeft[b].y += SPEED trackRight[b].y += SPEED b += 1

if trackLeft[len(trackLeft)-1].y > 32:

if trackDirection == False: trackPosition += 16 if trackDirection == True: trackPosition -= 16 if randint(0, 4) == 1: trackDirection = not

trackDirection

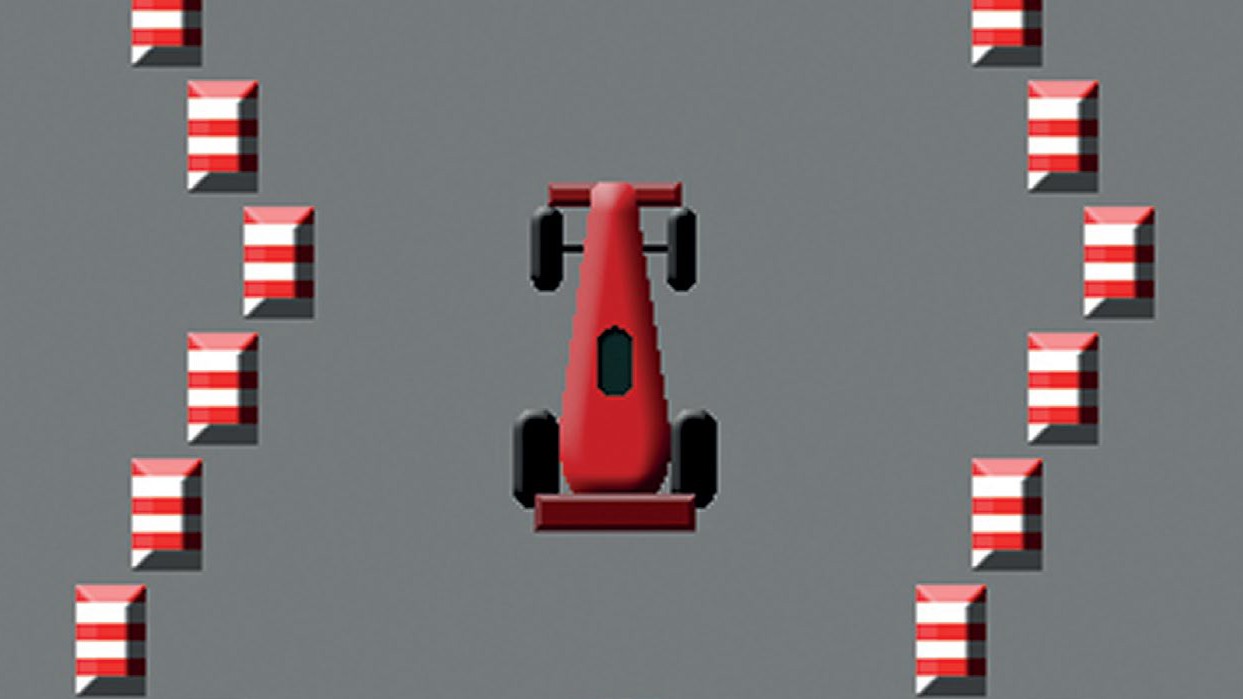
if trackPosition > 700-trackWidth:

trackDirection = True

if trackPosition < trackWidth: trackDirection = False

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.

**>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

figure4.py

should see a track snaking down towards the car. The only problem is that we can move the car over

01.

02.

03.

04.

05.

06.

07.

08.

09.

10.

11.

12.

13.

14.

15.

16.

17.

18.

19.

20.

21.

22.

23.

def draw(): # Pygame Zero draw function global gameStatus

screen.fill((128, 128, 128))

if gameStatus == 0:

car.draw()

b = 0

while b < len(trackLeft): trackLeft[b].draw() trackRight[b].draw()

b += 1

if gameStatus == 1:

# Red Flag

if gameStatus == 2:

# Chequered Flag

def update(): # Pygame Zero update function global gameStatus , trackCount

if gameStatus == 0:

if keyboard.left: car.x -= 2 if keyboard.right: car.x += 2 updateTrack()

if trackCount > 200: gameStatus = 2 # Chequered flag state

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.

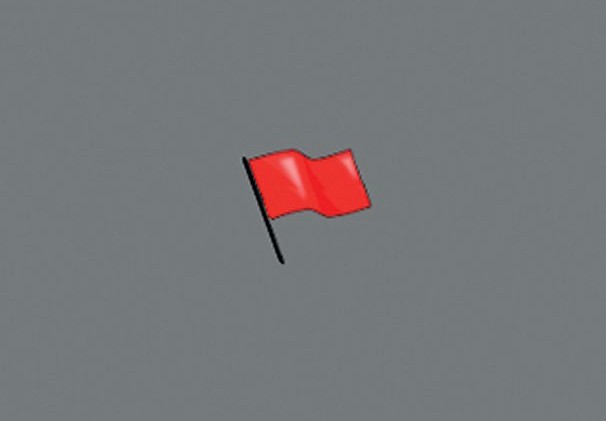
**>STEP-17**

**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,

**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

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:



**>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](http://magpi.cc/fBqznh)

**Anagram Advent Calendar - Python Step 1 - Build the Scramble Word Function** This step will build the function that will scramble words.

We will require the random library from Python, this will be used to shuffle a list of indexes. Those indexes will be used to pick the letters in a random order.

\* import random at the top of the code.

\* define the scramble word function, with a single parameter which I have called word.

\* generate a list of letter indexes by using the range function, wrapped in the list function.

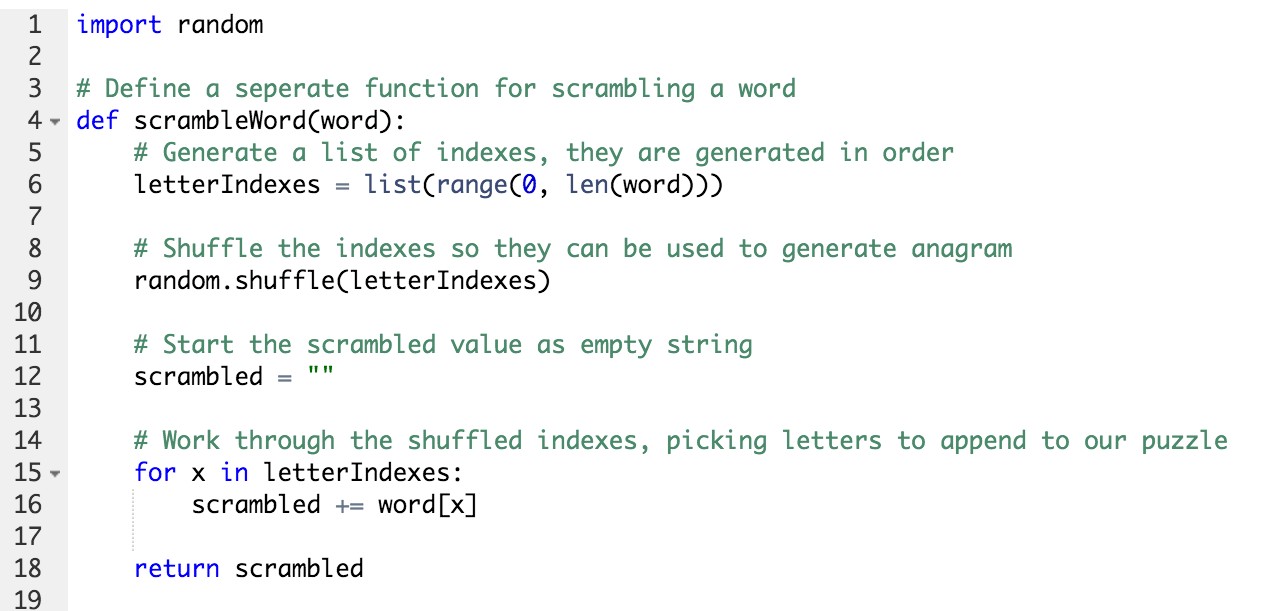
\* shuffle the list

\* create the variable that will hold the scrambled word

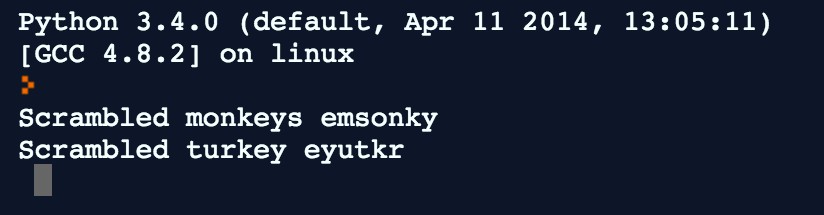
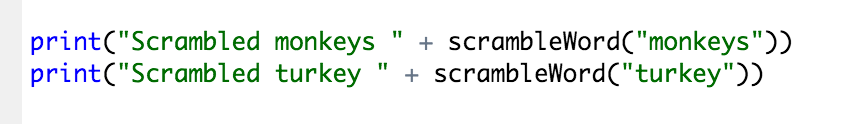
\* iterate through the letter indexes adding the letters to our scrambled word.

\* return the scrambled word.

When you have done this, the code should look like this



To test the function, call it a couple of times and print the results, for example.



**Step 2 - Build Player Interaction**

This step will show you how to ask the player for an answer and then check their answer.

\* Define a new function called testUser, it should have one parameter called puzzle

\* Ask the user for their attempt using the input function.

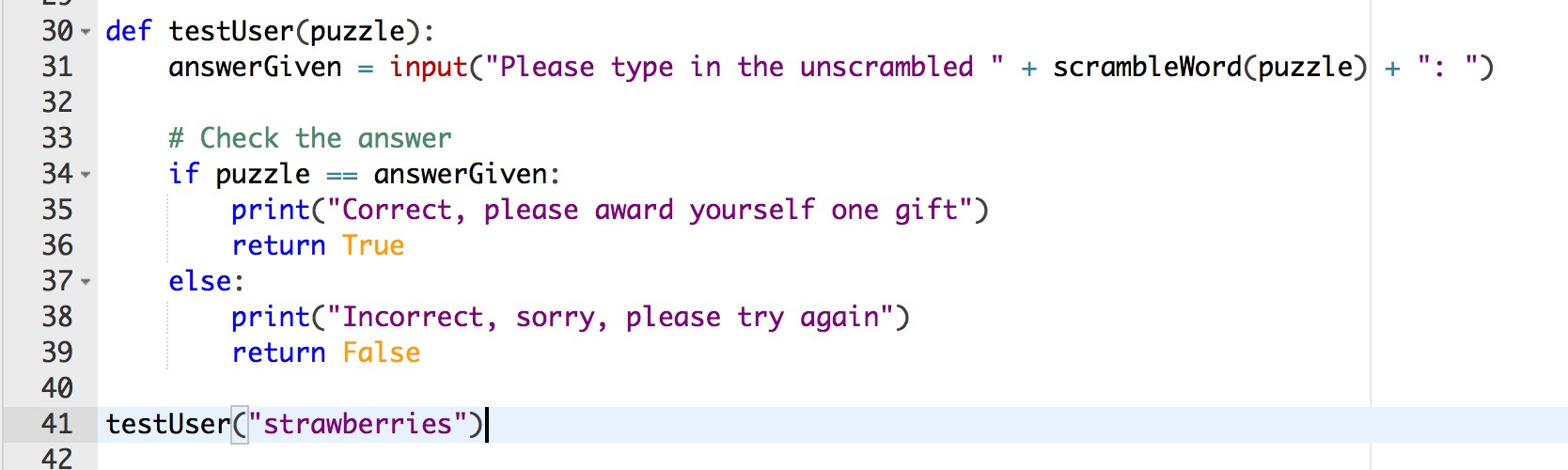
\* Check that the answer given equals the puzzle

\* Print congratulations or commiserations as appropriate.

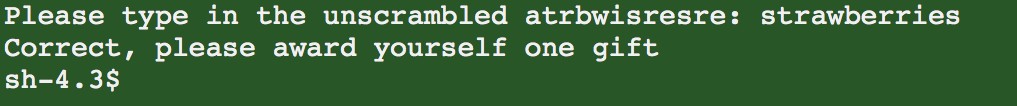
\* Return True for success, return False for failure. This will be required later

\* Call the testUser function, passing in a fixed puzzle

\* Run the code to try it out

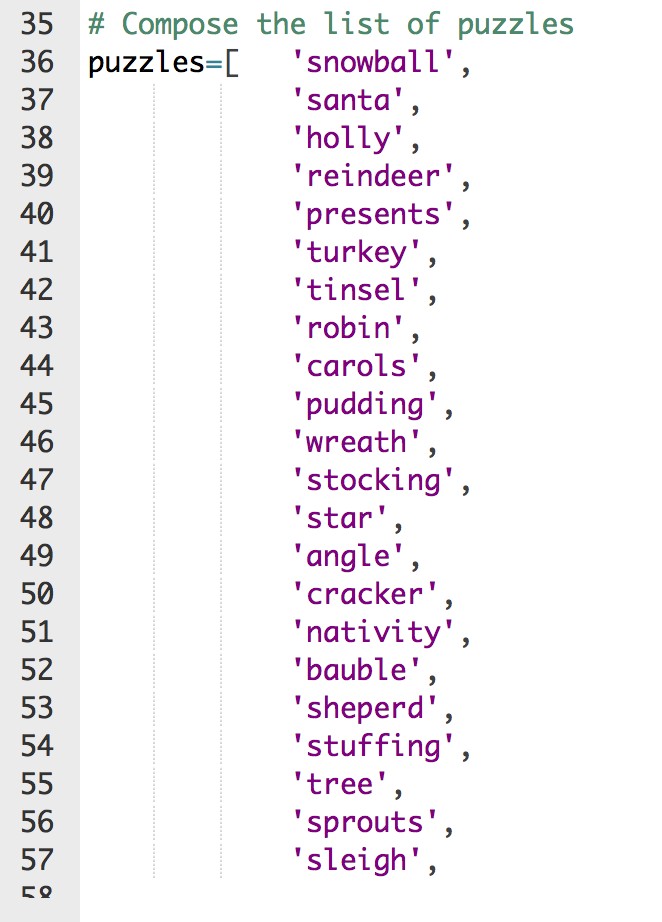


Once you have done this, comment out the call to testUser() using the # symbol



**Step 3 - Create Puzzle List**

In order to provide a different puzzle for each day, we should create a list with many puzzles in. The following code will create a list variable with a set of festive words in.



**Step 4 - Solve Today’s Puzzle**

In order to get the current day, we will need the datetime library. We can then use this library to access the current day.

\* Add the import date/time library call to the top of the python code

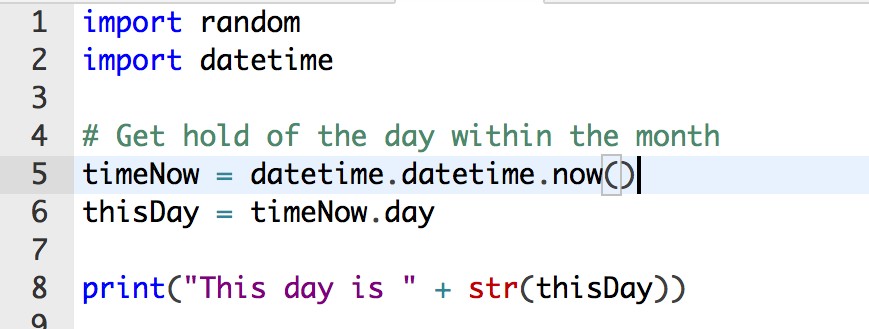
\* Just underneath call the datetime.datetime.now() function, this returns a time structure with

various fields required to define the current date and time.

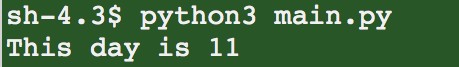
\* Create a new variable called thisDay and use the value of timeNow.day.

\* Put a print statement in to check that it is working.

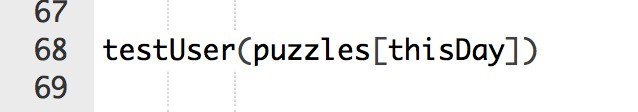
The top of the code should look like this



Run the code and the following should be printed



Now we need to present today’s puzzle to the user. We need to add the following steps at the bottom of the code (after the puzzles list has been defined).



Run the code again and you should be presented with a puzzle for the current day. If you run this code tomorrow it will present a different puzzle.

As before, comment out this line in preparation for the next stage. You can always comment it back in to retest things quickly.

**Step 5 - Print Text Based Calendar**

Now that we can present the user with a puzzle selected from our list, it would be nice to present all of the puzzles as hashed out values, the user can then pick their way through the ones they want, effectively ‘opening’ the doors.

\* In order to remember if the player has solved a particular day, a new list variable will be created containing boolean values (true/false). Create an empty list called solved.

\* Populate this list by appending False for each value of puzzle

\* Define a new function to print the puzzles in their current state.

\* In this function, loop round the puzzles by index

\* Compose a new variable called toPrint which is set to a row of hashes.

\* If the solved indicator shows the user has solved that puzzle, change the value of toPrint to the

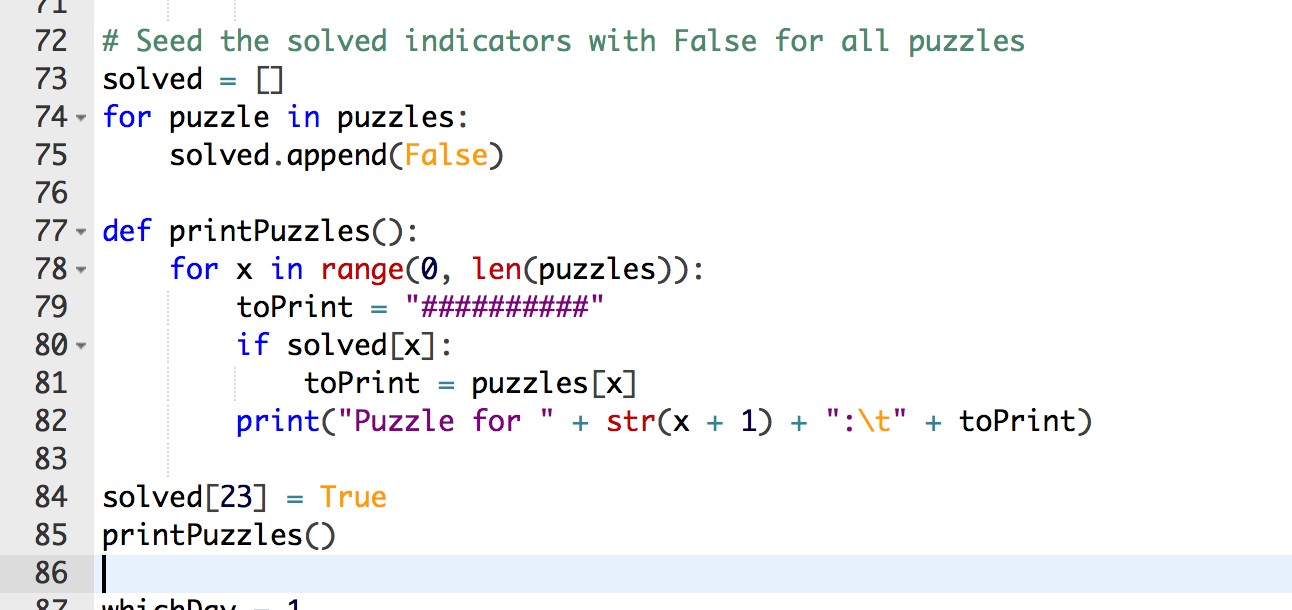
solved puzzle.

\* Print the toPrint value

\* Outside of the function definition, add a couple of lines just to test it, run the test and see what is

printed.

The function should look like this



Comment out the test lines in preparation for the next and final step.

**Step 6 - Loop Puzzles with User**

Now that we have our print function, we should continue calling out, allowing the user to solve as many puzzles as they want to.

\* Define a variable called whichDay, this will be used to store the day the user has selected to solve.

\* Start a while loop which runs forever

\* Prompt the user for a day selection using the input function

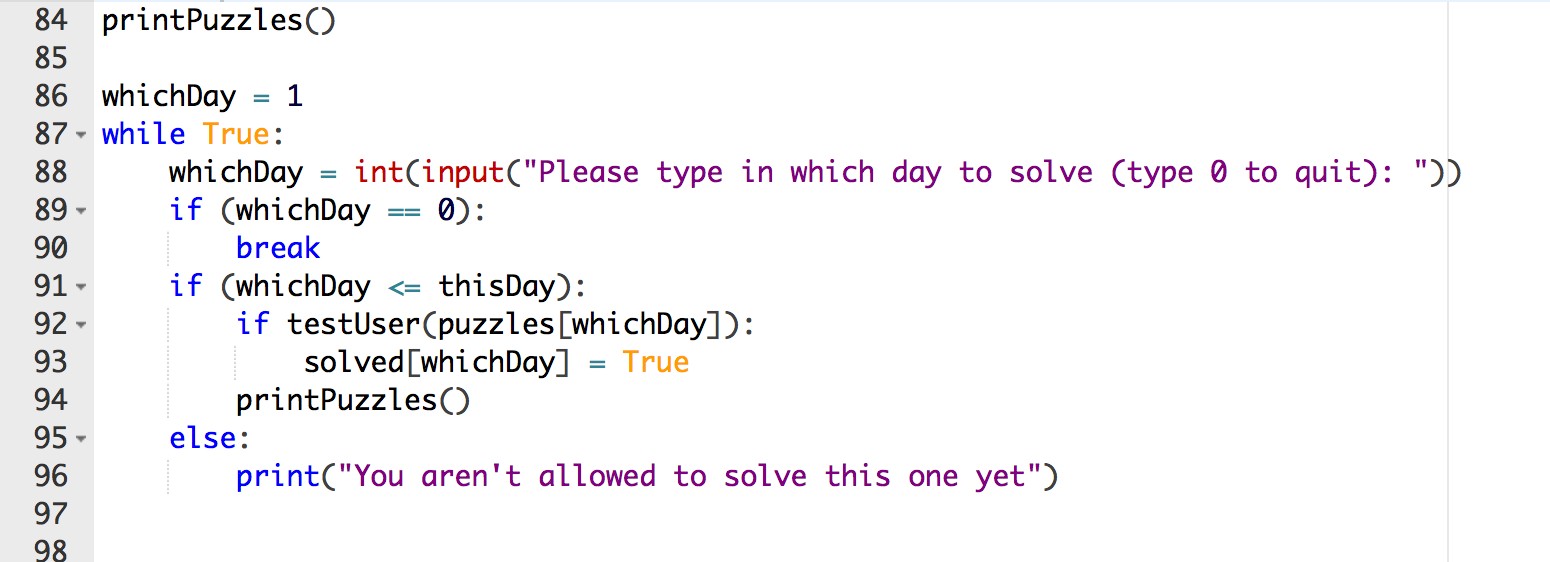
\* If the user types in zero, break out of the loop

\* Check the day is valid, then present the puzzle for the selected day

\* If the user correctly solves the puzzle, the testUser function will return true and we should mark it

as solved.

\* After the user has solved/not solved the puzzle, reprint the calendar. Once completed, this interaction loop code should look like this



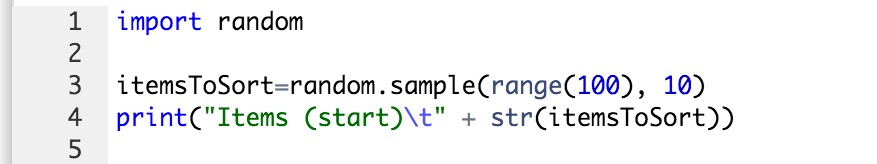
**CONGRATULATIONS YOU HAVE NOW COMPLETED THE ADVENT CALENDAR**

My own implementation can be found at<http://goo.gl/D0PdYD>

**Bubble Sorting Algorithm**

**Step 1 - Build a List**

In python this is very easy, we will use the random library to generate a list of 10 numbers within a range of 1-100. The following code will do this

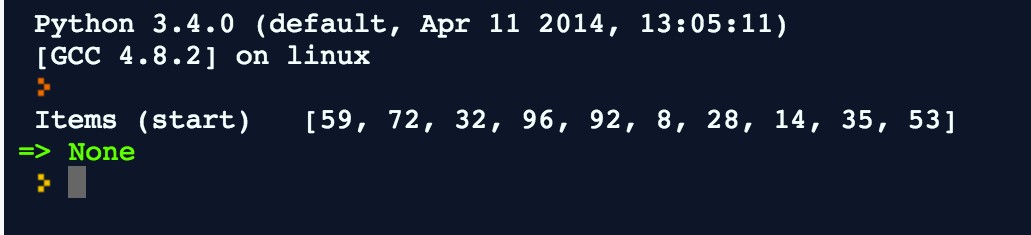


Line 1 tells Python that we wish to use the random library.

Line 3 generates the list and assigns it to the new variable **itemsToSort**

Line 4 prints out the randomised list so we can see it in the console.

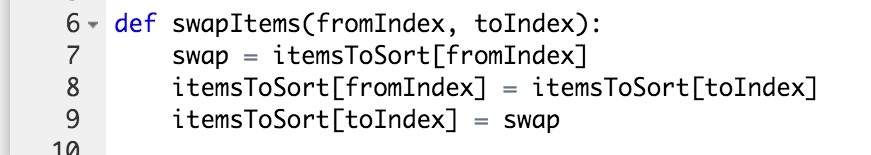
Press Ctrl-Enter (if using repl.it) and the following should be printed in the console window.



**Step 2 - Write a Swap Function**

This function will accept two indexes and it will swap the items in our **itemsToSort** list. Type the following code

The swap variable (created on line 7) is used to keep the item in **fromIndex** whilst the item in **toIndex** is being copied over it. This function operates on the **itemsToSort** variable declared above, we could have passed the list in as an argument too, which would make the function more re-usable.

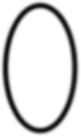
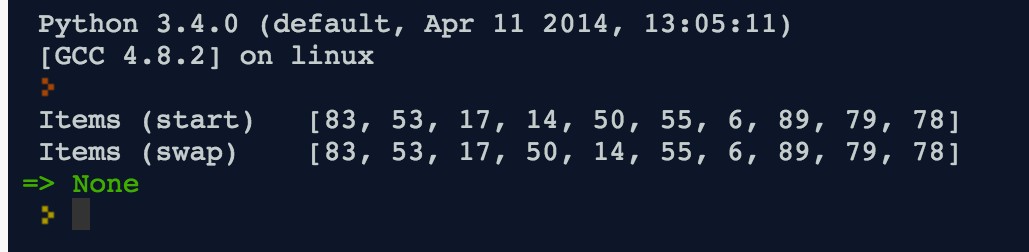


**Step 3 - Test the Swap Function**

In order to test the function, first we must call it. At the bottom of the code type the following lines.

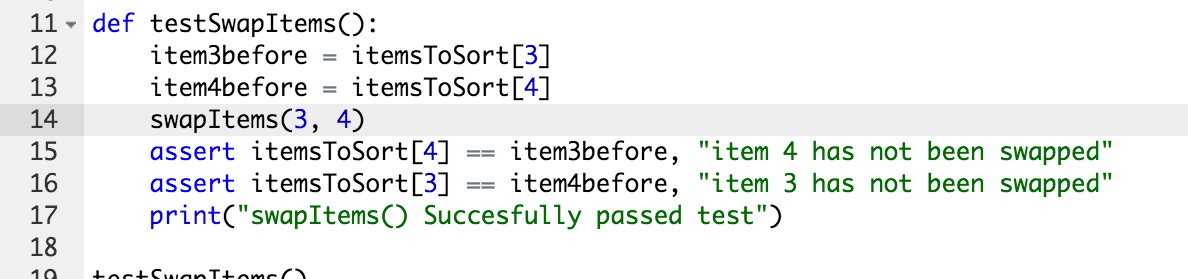


Line 11 calls our new function, Line 12 prints the list after the swap. From this we can visually inspect the output to see if it has worked.



In order to be more systematic in testing our function, we will write a small test function. This test function will execute the swap function, and check that it works properly. Once we have written this test, we can always use it to demonstrate the code working, without even having to do visual inspection as we have above.

Replace the code we wrote before with the following



Let’s break this down by line

12. The variable item3before remembers the value in index 3 *before* the swap is called.

13. This variable remembers the value in index 4.

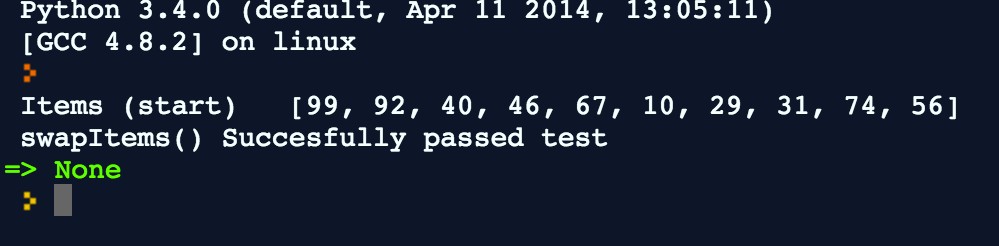
14. Calls the swapItems function with indexes 3 and 4

15. Checks that the item in index 4 is equal to the item that used to be in index 3. It will print a message if this is not correct for any reason.

16. Checks the opposite item, also prints a message.

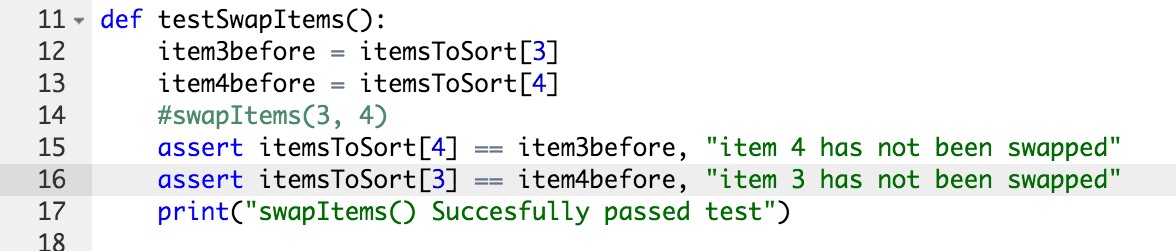
17. Prints to console so we can see it has run ok

Run this code and the following should be shown

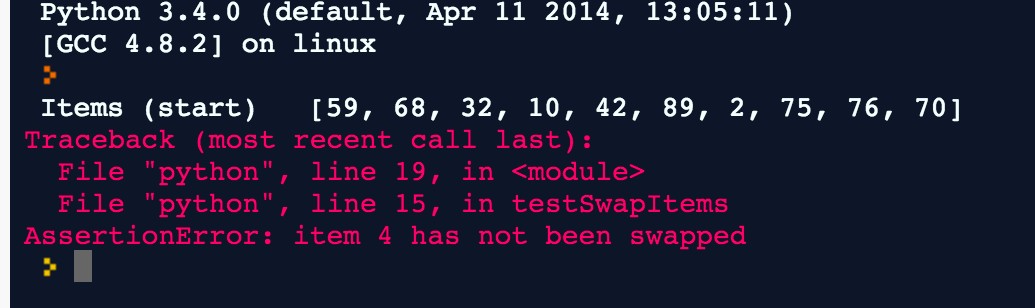


Assert statements are special statements that allow a programmer to state something that they expect at a given moment in time. In this case, at line 15, we expect that itemsToSort[4] contains the value that used to be in itemsToSort[3]. If this was not true, the program would throw an error.

To demonstrate this, comment out the line that calls the swap function. This will mean the swap is not run, so the test should fail.



Run the code again and the following should be shown

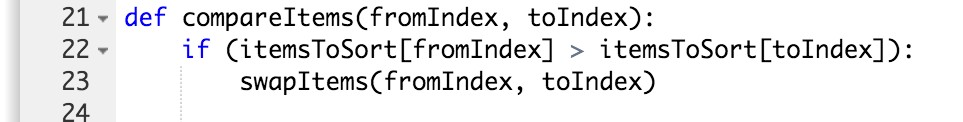


The last line shows the message we wrote, which helps us (the programmer) find the error and fix it. Uncomment the line 14 and we will continue.

**Step 4 - Write the Comparison Function**

The comparison function will be given two arguments, the fromIndex and toIndex (same as swap). But rather than simply swap the items, it first checks to see if they are in the correct order. If two items are already in the correct order, there is no need to swap them.

The code for the comparison looks like this



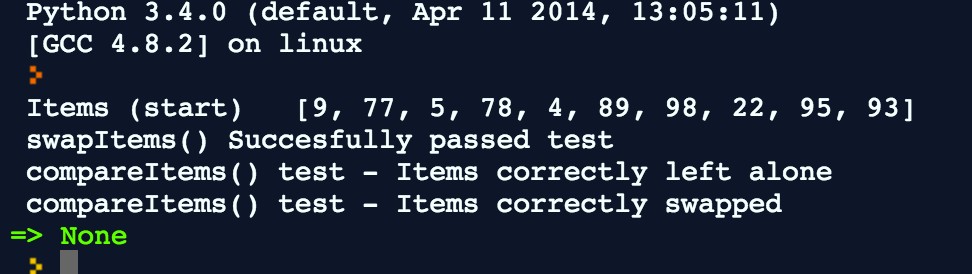
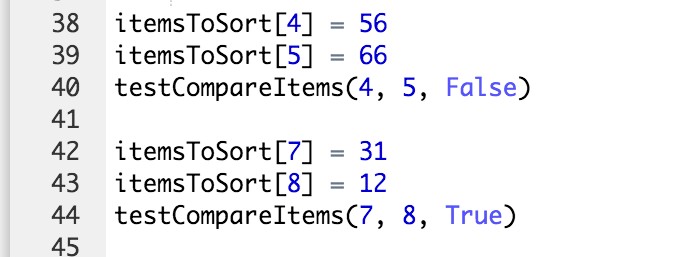
Line by line

21. This defines the function, it declares two arguments (fromIndex and toIndex).

22. This conditional statement compares the item in the fromIndex with the item in the toIndex. If the item in the lower position is actually larger than the item in the higher position, then the swap function is called.

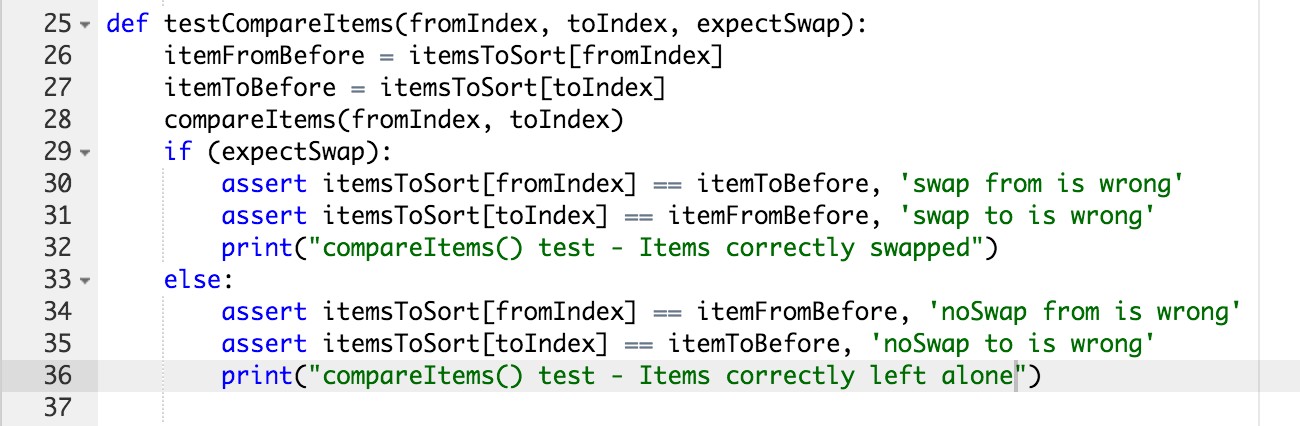
Now we need to test the function!

**Step 5 - Test the Comparison Function**



For this test to run properly, we must run two tests. The first will compare two items that we know are in the correct order, so they *should not* be swapped. The next will compare two items which we known are the in the wrong order, so they *should* be swapped.

First write the code for the test function itself



Line by line

25. This defines a test function that requires arguments, it means we can use this function for both cases. The two indexes are passed in along with an indication that they should be swapped, or not.

26 - 27. Remember value in fromIndex and toIndex before compareItems() is called

28. Calls compareItems, passing in the indexes under test

29. The assertions are different depending on if we expect a swap to happen.

29 - 31. Check that from and to were correctly swapped, print a success indicator

33. If expectSwap was False, we will check that the items have not moved

34 - 36. Check that the items are in their starting positions, print success indicator

This function now must be called in order to run any test. Before calling this function a bit of setup is required. We will put some fixed values into the list, in fixed positions. This allows the test to have predictable results.

Lines 38 - 39 put fixed values into positions 4 and 5. They are in the correct order, so a swap is not expected.

Line 40 then calls the function with these two indexes and indicates that a swap is not expected. Lines 42 - 43 put fixed values into position 7 and 8. These are in the wrong order, so a swap will be required to get them into the correct order.

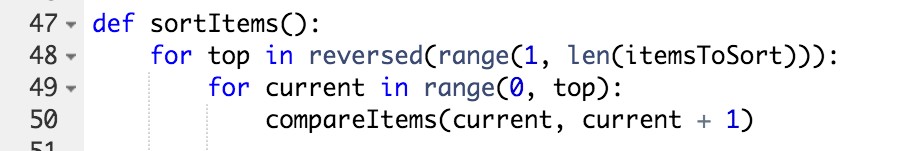
Line 44 calls the test function, indicating that a swap is expected.

When you run the code now, the output shown just above indicates that the tests have run

correctly. Again, try changing the expectSwap value on lines 40 or 44 and see if you get the correct errors.

**Step 6 - Write the Sort Function**

The sort function will use the comparison function in a repeated fashion until the list is in order. The sort function itself looks like this.



Line by line

46. Defines the sortItems() function. This requires no arguments as it will operate on the whole of the **itemsToSort** list that we have already created.

47. This is quite complex, so let’s break it up.

a. The **top** variable indicates the last item in the list that has not yet been sorted. At the start of the algorithm, the last item in the list is the top. But each run through will bring the highest value to the top, so therefore **top** reduces by 1 each time

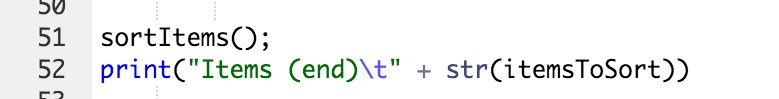
b. We need to count down from the length of the itemsToSort to 1. The range function will then give us a list of numbers like this [1, 2, 3, 4, 5, 6, 7, 8, 9]

c. The reversed() function takes this list and reverses it. This means that top will now count down from 9 to 0. It will now look like this [9, 8, 7, 6, 5, 4, 3, 2, 1]

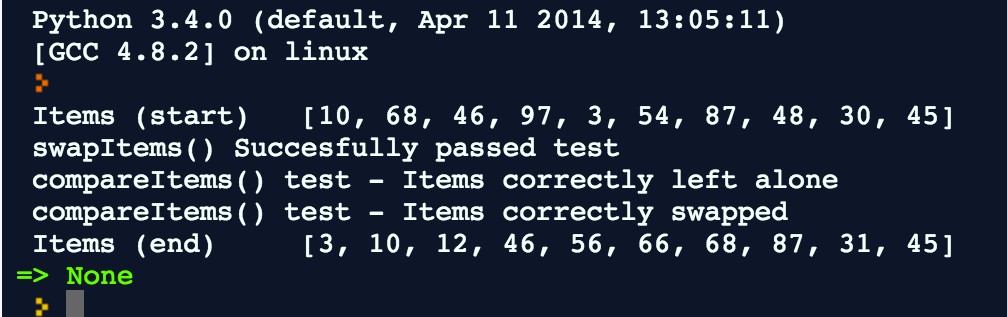
48. Within each iteration, we must count up from the first item to the last unsorted item. The statement inside the loop will then compare the **current** item with the **current + 1** item.

49. Call the compareItems() function itself. Each iteration will compare the **current** item with the one after it, if they are in the wrong order, they will be swapped.

The following code can be written on the end to call our function and reprint the list so we can see that it has been sorted.



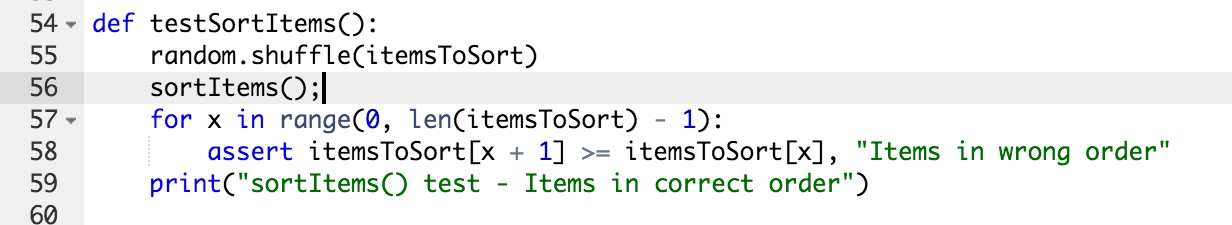
Run this code now, if everything has gone well, we should see something like this.



**Step 7 - Test the Sort Function**

We can visually inspect the printed list and check it is in order, but just like the other functions, it is a good idea to write a test. This test will do a single run through the items checking they are all in the correct order.

The test function looks like this.



54. Declares the test function, no arguments required

55. In order to run a valid test, we should shuffle the list at the start.

56. Call sortItems() to get the items back in order

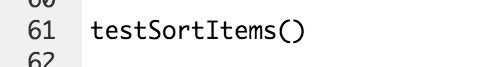
57. Run through all the items and check they are in the right order. The loop goes up to the length

-1. The minus 1 is required because the comparison will check x against x + 1, and we don’t want to run off the end.

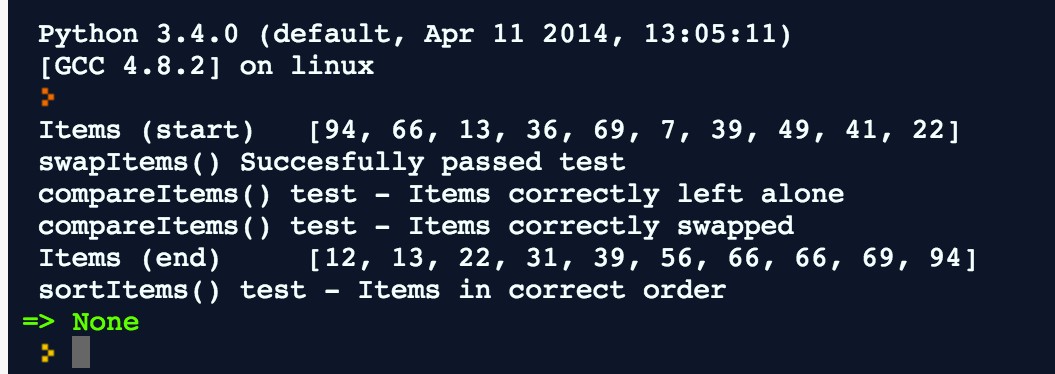
58. Assert that the current item is in the correct order relative to the next one in the list.

59. Print success

Call the testSortItems() by simply calling it.



Run the code and the following should be shown. This indicates that the program is working correctly.



**Step 8 - Tidy up and recap**

Before we finish, lets rearrange the code a bit. The order will now be as follows

1. Functions required by sort

1. swapItems()

2. compareItems()

3. sortItems()

2. Test functions

1. testSwapItems()

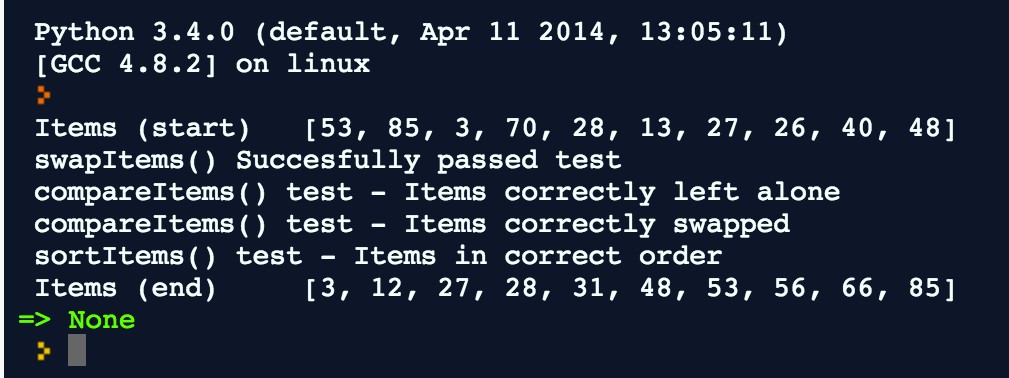
2. testCompareItems()

3. testSortItems()

3. Call test functions

4. Demonstration call

The next page shows the reorganised code, when run the output should now look like this. A working solution can be found here https://repl.it/B2GZ/54



**CONGRATULATIONS! YOU HAVE NOW IMPLEMENTED A BUBBLE SORT IN PYTHON**

1 import random

2

3 itemsToSort=random.sampler(ange(100),10)

4 print "Items (start\)t" + str(itemsToSort))

5

6T def swapltems(fromlndex, tolndex):

7 swap = itemsToSort[fromlndex]

8 itemsToSort[fromindex]= itemsToSort[toindex]

9 itemsToSort[toindex]= swap

10

llT def compareltems(fromlndex, tolndex):

12T *if* (itemsToSort[fromlndex]> itemsToSort[toindex]):

13 swapltems(fromlndex, tolndex)l

14

15T def sortltems():

16T

17T

18

19

for top in reversed range(l, len(itemsToSort))):

for current in range 0, top):

compareltems(current, current + 1)

20T def testSwapitems():

21 item3before = itemsToSort3[]

22 item4before = itemsToSort4[]

23 swapltems3(, 4)

24 assert itemsToSort4[]== item3before, "item 4 has not been swapped"

25 assert itemsToSort3[]== item4before, "item 3 has not been swapped"

26 print("swapltems()Succesfully passed test")

27

28T def testCompareitems(fromindex, tolndex, expectSwap):

29 itemFromBefore = itemsToSort[fromindex]

30 itemToBefore = itemsToSort[toindex]

31 compareltems(fromlndex, tolndex)

32T *if* (expectSwap):

33 assert itemsToSort[fromlndex]== itemToBefore, 'swap from is wrong'

34 assert itemsToSort[toindex]== itemFromBefore, 'swap to is wrong'

35 print("compareltems()test - Items correctly swapped")

36T else:

37 assert itemsToSort[fromindex]== itemFromBefore, 'noSwap from is wrong'

38 assert itemsToSort[toindex]== itemToBefore, 'noSwap to is wrong'

39 print("compareltems()test - Items correctly left alone")

40

-rv

41T def testSortitems(:)

42 random.shuffle(itemsToSort)

43 sortltems );

44T for x in range 0, len itemsToSort)- 1):

45 assert *i* temsToSort[x + 1]>= *i* temsToSort[x], "Items in wrong order"

46 print "sort!tems()test - Items in correct order")

47

48 testSwapitems()

49

50 itemsToSort4[]= 56

51 itemsToSort5[]= 66

52 testCompareltems4(, 5, False)

53

54 itemsToSort7[]= 31

55 itemsToSort8[]= 12

56 testCompareitems7(, 8, True)

57

58 testSortitems(

59

60 sortltems();

61 print "Items (end\)t" + str itemsToSort )

62

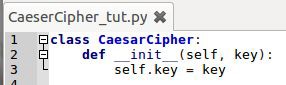
Caesar Cipher - Python

In this tutorial we will write an object oriented version of the Caesar Cipher. The Caesar Cipher was used to encrypt simple text messages by shifting all the letters by a fixed amount (the key). I am assuming you know the mechanics of the cipher, but if you don’t you can read about it here.

<https://learncryptography.com/classical-encryption/caesar-cipher>

Create a Class

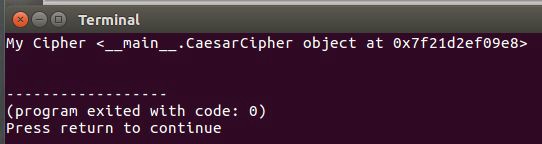
In Object Oriented programming, *classes* are used to associate methods with data. This allows the programmer to encapsulate concepts in reusable units of code. We will see how this works by diving straight into our cipher class. Start a new Python file and type the following.



This gives us a class called **CaesarCipher**, and we are going to give that class a single bit of data to play with; the key. The function that starts on line 2 is a special function known as a *constructor*. It will be called when we create an *instance* of our class. Let us do this now.



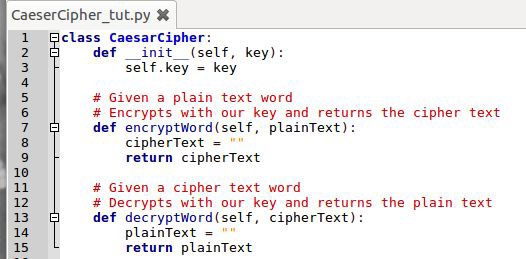
Line 5 creates an instance of our cipher, and sets the **key** to 4. We do not have to pass in **self**, this is done for us by Python. Line 6 is just to print the object we have created. Run the code and the output will look something like this:



Eww...well we don’t need to print the cipher object anyway.

Add Methods to Class

For our class to be useful, we should now add functions that can be called. I will be adding 2 functions to start with, **decryptWord** and **encryptWord**. Take note of the indentation, the methods are shown to be inside the class. The functions will look like this (lines 5 to 15):



Note that I have added some comments to the functions, this will help programmers that read the code to understand what is going on. They are ignored by Python during execution though so you can leave them out if you want to.

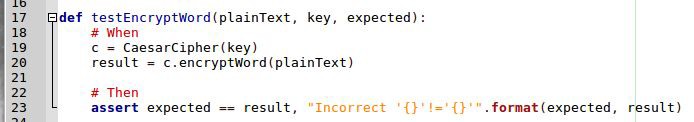
Both of our class methods require the first parameter to be **self**, later you will see how this is used to refer to the self.key in the encrypt and decrypt functions.

Write a Test

It should be fairly obvious that our functions do not yet work, they simply return a blank string no matter what the input is. But let us write a test anyway. We can then develop the encryption and decryption until the tests pass!

I will write a function for testing the encryption of a word, it will use the **assert** keyword to ensure that the result is correct. This will give us a chance to see how assertions are handled when

they fail. Write the following function below the Caesar Cipher class (note indentation; it will be outside of the class).

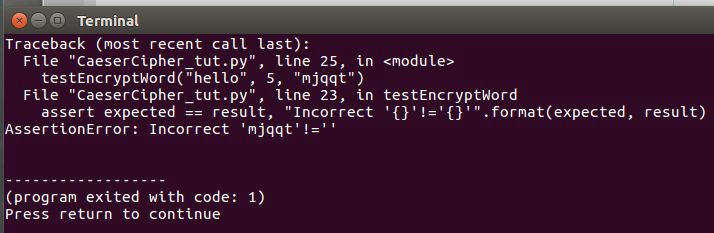


The assert line checks that the result matches the expected one, it prints an error message using the format function on string. The curly brackets {} are used as placeholders for the parameters in the format function.

Let us see this test function in action, delete the two lines (shown here as 25 and 26) we wrote earlier and add the line shown here as line 28



Run this and the test should fail, the terminal print will look something like this.



The part after the **AssertionError:** is the message we composed in our code, this lets you see what the problem is.

Do Yourself

Write a decrypt word function using the following test data

● Cipher Text: mehbs

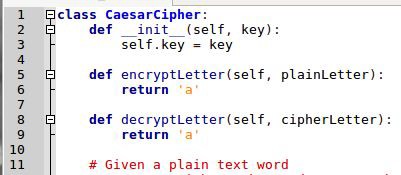
● Key: 15

● Expected Plain Text: world

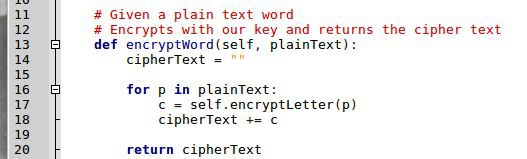
Write Encrypt/Decrypt Letter Functions

Our encryption algorithm can work on each character in the word separately, so I am going to add functions for **encryptLetter** and **decryptLetter**. We can then loop through the letters in our words in the **encryptWord** and **decryptWord** functions and build our text that way.

Add the following functions to our class. I have put them above our decryptWord and encryptWord functions.



I will show you how to use the encryptLetter function in our encryptWord function.



I am using a simple for loop to iterate through the letters of our plainText, then using the neat +=

operator to add the encrypted letter to our cipher text. Rerun the tests and the failure should look different.



The incorrect value is now the right length, but the characters are all ‘a’ which is obviously no good.

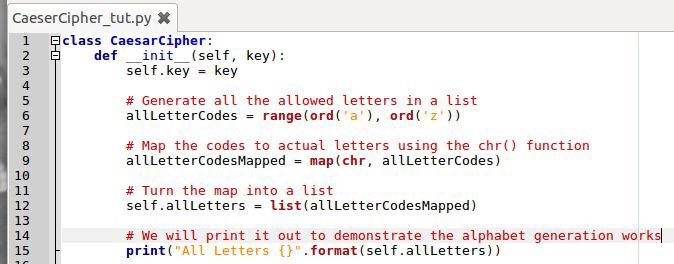
Do Yourself

Fill in the decrypt word function in a similar fashion. Rerun the test from earlier (comment out the encrypt test call so it doesn’t interfere). You should find the decrypted text is the right length, but the wrong contents.

Generate an Alphabet

Our algorithm requires an alphabet of letters in a list, we will use the range and map functions in

Python to do this very succinctly. Add the following code to our constructor.

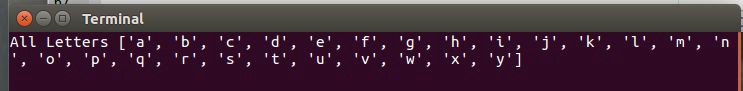


There is a lot of cool stuff here, so let’s break it down.

Line 6: The **ord()** function is used to get the numerical representation of a character. Here we are going from lowercase a to z and generating a **range()**.

Line 9: The **map()** function is shorthand for a loop, it loops round the **allLetterCodes** list and applies the **chr()** function. This syntax is much tighter than using a for loop.

Line 12: The **list()** function is used to take the map and generate a list. I have then printed the **allLetters** variable (which is put onto the object using **self**) so that you can see it working properly.



Implement Encrypt Letter Correctly

We will now write the actual meat of the algorithm, the letter shifting. My method for this will be to create an array containing all the letters in the alphabet.

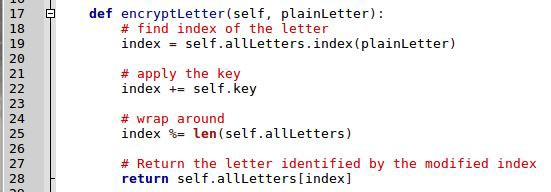
When given a letter to encrypt our program will:

● Find the index of that letter in our alphabet

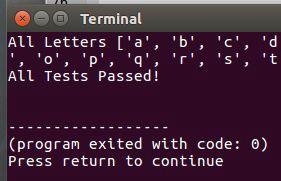
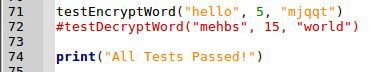
● Shift the index using the key

● If the index ends up larger than the alphabet we will use modulo to wrap it.

● Return the letter found at the new index. The code looks like this



Run your encryptWord test from earlier and you should find it passes. If the assertion passes then nothing is printed, so you will just see the All Letters print from earlier and the usual (program exited….) so you may wish to add a print line to the end of the code as shown, I have commented out the decrypt test (that you wrote earlier) so that it doesn’t spoil the glorious moment of successful encryption.

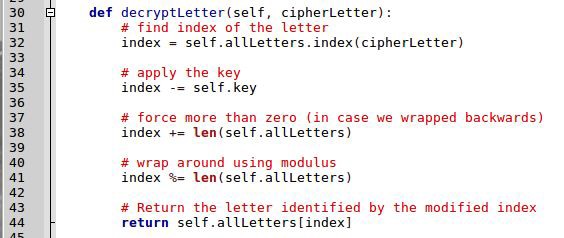


You may wish to comment out the alphabet print once you have satisfied yourself it is working.

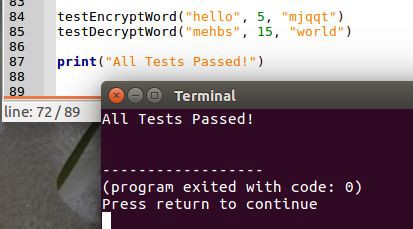


Implement Decrypt Letter Correctly

We will go through essentially the reverse process in decrypt letter. The code for that function will look like this.



Let your decrypt test function run and you should now find that all tests pass!



Congratulations! You have built a Caesar Cipher and done a bit of testing.

Further Work

Now that your cipher works, try adding some user interaction using the input function. There are three things to capture

● If the user wishes to encrypt or decrypt

● The text to process

● The key

The program can then just print out the answer.

**from** mcpi**.**minecraft **import** Minecraft

**from** mcpi **import** block

**import** DungeonError

**class Dungeon:**

\_mc **=** Minecraft**.**create**()**

\_rooms **= {}**

\_MIN\_X **= -**10

\_MIN\_Y **= -**30

\_MIN\_Z **= -**80

\_MAX\_X **=** 120

\_MAX\_Y **= -**12

\_MAX\_Z **=** 80

\_ROOM\_WIDTH **=** 10

\_ROOM\_LENGTH **=** 10

\_ROOM\_HEIGHT **=** 4

###########################################################

#

# Construct a room with the given dimensions. Place torches

# around the walls, optionally put a random piece of

# treasure in the room.

#

###########################################################

**def** MakeRoom**(**self**,** x1**,** y1**,** z1**,** x2**,** y2**,** z2**):**

# Plot the room itself

self**.**\_mc**.**setBlocks**(**x1**,** y1**,** z1**,** x2**,** y2**,** z2**,** block**.**AIR**.**id**)**

# Determine height of torch th **=** min**([**y1**,** y2**]) +** 2

# Place torches along each wall.

**for** i **in** range**(**x1**+**1**,** x2**-**1**):**

**if** i **%** 6 **==** 0 **and (**self**.**\_mc**.**getBlock**(**i**,** th**,** z1**-**1**) !=** block**.**AIR**):**

self**.**\_mc**.**setBlock**(**i**,** th**,** z1**,** block**.**TORCH**)**

**if** i **%** 6 **==** 0 **and (**self**.**\_mc**.**getBlock**(**i**,** th**,** z2**+**1**) !=** block**.**AIR**):**

self**.**\_mc**.**setBlock**(**i**,** th**,** z2**,** block**.**TORCH**)**

**for** i **in** range**(**z1**+**1**,** z2**-**1**):**

**if** i **%** 6 **==** 0 **and (**self**.**\_mc**.**getBlock**(**x1**-**1**,** th**,** i**) !=** block**.**AIR**):**

self**.**\_mc**.**setBlock**(**x1**,** th**,** i**,** block**.**TORCH**)**

**if** i **%** 6 **==** 0 **and (**self**.**\_mc**.**getBlock**(**x2**+**1**,** th**,** i**) !=** block**.**AIR**):**

self**.**\_mc**.**setBlock**(**x2**,** th**,** i**,** block**.**TORCH**)**

###########################################################

#

# Construct a flight of steps going downwards. The steps

# must go through water so must include walls and ceiling.

#

############################################################

**def** MakeSteps**(**self**,** x**,** y**,** z**,** l**):**

**for** step **in** range**(**l**+**1**):**

# Floor steps

self**.**\_mc**.**setBlock**(**x**+**step**,** y**-**step**,** z**-**1**,** block**.**STAIRS\_COBBLESTONE**.**id**,** 1**)** self**.**\_mc**.**setBlock**(**x**+**step**,** y**-**step**,** z**,** block**.**STAIRS\_COBBLESTONE**.**id**,** 1**)** self**.**\_mc**.**setBlock**(**x**+**step**,** y**-**step**,** z**+**1**,** block**.**STAIRS\_COBBLESTONE**.**id**,** 1**)**

# Ceiling steps

self**.**\_mc**.**setBlock**(**x**+**step**,** y**-**step**+**4**,** z**-**1**,** block**.**STAIRS\_COBBLESTONE**.**id**,** 4**)** self**.**\_mc**.**setBlock**(**x**+**step**,** y**-**step**+**4**,** z**,** block**.**STAIRS\_COBBLESTONE**.**id**,** 4**)** self**.**\_mc**.**setBlock**(**x**+**step**,** y**-**step**+**4**,** z**+**1**,** block**.**STAIRS\_COBBLESTONE**.**id**,** 4**)**

# Sidewalls

self**.**\_mc**.**setBlocks**(**x**+**step**,** y**-**step**,** z**-**2**,** x**+**step**,** y**-**step**+**self**.**\_ROOM\_HEIGHT**,** z**-**2**,**

block**.**STONE**)**

self**.**\_mc**.**setBlocks**(**x**+**step**,** y**-**step**,** z**+**2**,** x**+**step**,** y**-**step**+**self**.**\_ROOM\_HEIGHT**,** z**+**2**,**

block**.**STONE**)**

# Ensure that nothing is in the void.

self**.**\_mc**.**setBlocks**(**x**+**step**,** y**-**step**+**1**,** z**-**1**,** x**+**step**,** y**-**step**+**3**,** z**+**1**,** block**.**AIR**)**

###########################################################

#

# Construct a flight of steps going downwards to join rooms.

#

############################################################

**def** StepsDown**(**self**,** x**,** y**,** z**):**

self**.**\_mc**.**setBlocks**(**x**+**3**,** y**,** z**-**1**,** x**+**self**.**\_ROOM\_HEIGHT**+**4**,** y**,** z**-**1**,** block**.**FENCE**)** self**.**\_mc**.**setBlocks**(**x**+**3**,** y**,** z**+**1**,** x**+**self**.**\_ROOM\_HEIGHT**+**4**,** y**,** z**+**1**,** block**.**FENCE**)** self**.**\_mc**.**setBlocks**(**x**+**self**.**\_ROOM\_HEIGHT**+**4**,** y**,** z**-**1**,** x**+**self**.**\_ROOM\_HEIGHT**+**4**,** y**,** z**+**1**,** block**.**FENCE**)**

**for** step **in** range**(**1**,** self**.**\_ROOM\_HEIGHT**+**3**):**

# Floor steps

self**.**\_mc**.**setBlock**(**x**+**step**,** y**-**step**,** z**,** block**.**STAIRS\_COBBLESTONE**.**id**,** 1**)**

# Space to walk

self**.**\_mc**.**setBlocks**(**x**+**step**+**1**,** y**-**step**,** z**,** x**+**self**.**\_ROOM\_HEIGHT**+**3**,** y**-**step**,** z**,** block**.**

AIR**)**

############################################################

#

# Construct a new room.

#

############################################################

**def** newRoom**(**self**,** start**,** direction**):**

handle **=** len**(**self**.**\_rooms**)**

**if** direction**.**upper**() not in [**"N"**,** "S"**,** "E"**,** "W"**,** "U"**,** "D"**]:**

**raise** DungeonError**.**DirectionError**(**"Direction must be: \"N\", \"S\", \"E\",

\"W\", \"U\" or \"D\""**)**

**else:**

**if** start **not in** self**.**\_rooms**:**

**raise** DungeonError**.**RoomError**(**"Unknown room: " **+** str**(**start**))**

**else:**

x**,** y**,** z **=** self**.**\_rooms**[**start**]**

**if (**direction**.**upper**() ==** "N"**):**

x **+=** self**.**\_ROOM\_LENGTH**+**2

**elif (**direction**.**upper**() ==** "S"**):**

x **-=** self**.**\_ROOM\_LENGTH**+**2

**elif (**direction**.**upper**() ==** "E"**):**

z **+=** self**.**\_ROOM\_WIDTH**+**2

**elif (**direction**.**upper**() ==** "W"**):**

z **-=** self**.**\_ROOM\_WIDTH**+**2

**elif (**direction**.**upper**() ==** "U"**):**

y **+=** self**.**\_ROOM\_HEIGHT**+**2

**else:**

y **-=** self**.**\_ROOM\_HEIGHT**+**2

**if (**x **>** self**.**\_MAX\_X**-**self**.**\_ROOM\_LENGTH

**or** x **<** self**.**\_MIN\_X

**or** z **<** self**.**\_MIN\_Z

**or** x **>** self**.**\_MAX\_Z**-**self**.**\_ROOM\_WIDTH

**or** y **>** self**.**\_MAX\_Y

**or** y **<** self**.**\_MIN\_Y**):**

**raise** DungeonError**.**RoomError**(**"Not enough space in direction \"" **+**

direction **+** "\" from room " **+** str**(**start**))**

**else:**

**if (**self**.**\_mc**.**getBlock**(**x**+**1**,** y**,** z**+**2**) ==** block**.**AIR**.**id**):**

**raise** DungeonError**.**RoomError**(**"There is already a room in direction

\"" **+** direction **+** "\" from room " **+** str**(**start**))**

**else:**

self**.**MakeRoom**(**x**,** y**,** z**,** x**+**self**.**\_ROOM\_LENGTH**,** y**+**self**.**\_ROOM\_HEIGHT**,** z**+**

self**.**\_ROOM\_WIDTH**)**

self**.**\_rooms**[**handle**] = [**x**,** y**,** z**]**

# Create the ajoining corridore or steps

**if (**direction**.**upper**() ==** "N"**):**

self**.**\_mc**.**setBlocks**(**x**-**2**,** y**,** z**+**self**.**\_ROOM\_WIDTH**/**2**,** x**,** y**+**1**,** z**+**self**.**

\_ROOM\_WIDTH**/**2**,** block**.**AIR**)**

**elif (**direction**.**upper**() ==** "S"**):** self**.**\_mc**.**setBlocks**(**x**+**self**.**\_ROOM\_LENGTH**,** y**,** z**+**self**.**\_ROOM\_WIDTH**/**2**,** x**+**self**.**\_ROOM\_LENGTH**+**2**,** y**+**1**,** z**+**self**.**\_ROOM\_WIDTH**/**2**,** block**.**AIR**)**

**elif (**direction**.**upper**() ==** "E"**):**

self**.**\_mc**.**setBlocks**(**x**+**self**.**\_ROOM\_LENGTH**/**2**,** y**,** z**-**2**,** x**+**self**.**

\_ROOM\_LENGTH**/**2**,** y**+**1**,** z**,** block**.**AIR**)**

**elif (**direction**.**upper**() ==** "W"**):** self**.**\_mc**.**setBlocks**(**x**+**self**.**\_ROOM\_LENGTH**/**2**,** y**,** z**+**self**.**\_ROOM\_WIDTH**,** x**+**self**.**\_ROOM\_LENGTH**/**2**,** y**+**1**,** z**+**self**.**\_ROOM\_WIDTH**+**2**,** block**.**AIR**)**

**elif (**direction**.**upper**() ==** "U"**):**

self**.**StepsDown**(**x**+**1**,** y**,** z**+**2**)**

**else:**

self**.**StepsDown**(**x**+**1**,** y**+**self**.**\_ROOM\_HEIGHT**+**2**,** z**+**2**)**

**return** handle

**def** init **(**self**):**

# Create a large underwater block of stone for the dungeon. self**.**\_mc**.**setBlocks**(**self**.**\_MIN\_X**, -**4**,** self**.**\_MIN\_Z**,** self**.**\_MAX\_X**, -**30**,** self**.**\_MAX\_Z**,** block

**.**STONE**)**

self**.**\_mc**.**setBlocks**(**self**.**\_MIN\_X**-**2**,** 0**,** self**.**\_MIN\_Z**-**2**,** self**.**\_MAX\_X**+**2**, -**3**,** self**.**\_MAX\_Z**+**2**,**

block**.**WATER\_STATIONARY**)**

self**.**\_mc**.**setBlocks**(**self**.**\_MIN\_X**-**4**,** 0**,** self**.**\_MIN\_Z**-**4**,** self**.**\_MAX\_X**+**4**,** 30**,** self**.**\_MAX\_Z**+**4**,**

block**.**AIR**)**

# Create external plinth for player to stand on self**.**\_mc**.**setBlocks**(-**6**,** 0**, -**5**,** 0**, -**3**,** 5**,** block**.**STONE**)**

**def** create**(**self**):**

# Create a dungeon as the start point self**.**MakeRoom**(**10**, -**12**, -**5**,** 20**, -**8**,** 5**)** self**.**\_rooms**[**0**] = [**10**, -**12**, -**5**]**

# Place some steps down into the dungeon. self**.**MakeSteps**(**1**,** 0**,** 0**,** 12**)**

# Put the playr near the dungeon entrance. self**.**\_mc**.**player**.**setPos**(-**2**,** 1**,** 0**)** self**.**\_mc**.**setting**(**"world\_immutable"**, True) return** 0