**Session 4**

In the last session we left the problem on the lcd of how to display words that have more than eight letters like the word Australia. We will now return to that problem and in doing so will show you two ways to approach this type of issue, TDD and commenting.

**Test Driven Development (TDD)**

Up to now we have written our code and then written some tests to make sure this works. However many of the best programmers do this the other way around, they write their tests first. This has a number of advantages:

* It helps to clearly define what you are trying to achieve
* It builds lots of tests very quickly which make your code robust and maintainable
* It makes problem solving much easier since it breaks the problem into small pieces
* The approach works particularly well when used to create functions which return data, in this case it is often called Unit Testing
* There are lots of tools, tutorials and support materials around to help you, we will look at one of these tools in Session 6

So let’s use this approach to work out of how to display words that are longer than 8 characters. Create a new file with the code below and run it.

if \_\_name\_\_ == "\_\_main\_\_":

 assert True

When you run this you see absolutely nothing happening. So let’s add one more line

if \_\_name\_\_ == "\_\_main\_\_":

 assert True

 print("All correct")

You now see the print statement working. So what happens with the other line?

**assert True** - **assert** is a Python keyword used for testing expressions. If the expression is true it moves to the next line, if the expression is false it stops the program and throws an error. To see this change **True** to **False** and run the program again. We can use **assert** to build a bank of tests and quickly run them to see whether everything works properly.

So let’s build our first real test. Change the **assert** to

assert splitter("hello")=="hello"

When you run this you get an error

NameError: name 'splitter' is not defined

We now have an error so we can work to fix this. This is exactly how TDD works.

1. You first write a test that currently fails but should pass
2. You then write enough code to make it pass, AND NO MORE
3. You refactor your code (make it better) while checking your code still works
4. Then write another failing test.

Our test fails so we can write some real code to make it pass. We want to write a splitter function which will split a word up if it is longer than 8 letters. Let’s make it that we split it after 6 letters and use a hyphen -. However our test is showing we don’t have that function at all yet, so let’s write the function. Above the if statement is the place to write our real working code. Remember in TDD you do enough to make the error go away and no more. So we write

def splitter():

 '''This function will split a word after 6 letters and add a hypen'''

 pass

**pass** is a Python Key word which is used to create empty functions which will be filled in later. Now when you run the test you get a different error

TypeError: splitter() takes 0 positional arguments but 1 was given

This is a different error, we are progressing, soon we will have no errors. This error is because in the test we are giving the function splitter some data ('hello') but our function does not have a parameter to accept this yet. Let’s fix this error

def splitter(data):

We now get an Assertion Error but Python gives us no help. We must work this one out ourselves. If we were using tools we would get more information at this stage, but in this session we are not using tools so we need to solve this for ourselves.

The problem here is that we expect the function splitter to give us some data, but at the moment the body of the function just has the keyword **pass** in it, which is just a placeholder. So what does the function currently give us? Let’s write a print statement to find out. Above the assert statement write the following:

if \_\_name\_\_ == "\_\_main\_\_":

 print(splitter("hello"))

 assert splitter("hello") == "hello"

This gives us the same error, because the assert line breaks us out of the program before the print line can actually work. To get the print line to work we need to comment out the assert. Do this we see that the print statement prints out the value returned by the splitter function which is **None**.

In Python all functions will return some data, whether you asked them to or not. If you don’t specify what the return value is, Python will give it a value of **None**. This is actually a good thing because you can test for that.

In this case we want splitter to return a value (‘hello’) but it is actually returning **None**. These are not the same which means the expression is false and assert will throw an error.

In TDD you must fix your code with the simplest solution, which is the following

def splitter():

 '''This function will split a word after 6 letters and add a hypen'''

 return "hello"

Uncomment the assert statement and your tests now run and pass.

We can refactor if we need to and then write another test. We haven’t done much yet but we already are on our way. Let’s write another test.

Let’s test another word.

assert splitter("Its") == "Its"

We get another error this time at our new test, our first test still passes. So now we must fix the function. The simplest way is the way we already tried, let’s just make the following change

def splitter(data):

 return "Its"

But this also throws an error, this time our first test fails. We need another solution.

We are giving the function two different words and expect two different answers, so we must use the data that is being given to the function. Change the function to the following:

def splitter(data):

 return data

and now both test pass. Time to refactor, but we still don’t have enough to do this properly so let’s write another test, with a longer word.

assert splitter("already") == "already"

This test also passes. Time to refactor.

Let’s do some thinking here. Our working code does not need any changes but our tests need to change. We need some words that we know won’t work and it would be nice to have a sentence to work with, so let’s do that. Add the following line:

if \_\_name\_\_ == "\_\_main\_\_":

 quote="Its already tomorrow in Australia."

 assert splitter("hello")=="hello"

and then change the two current tests to:

 assert splitter(quote.split()[0])=="Its"

 assert splitter(quote.split()[1])=="already"

You should understand these lines of code if you looked closely at the last Session and activities, though you might not have realise that Python can do all this work in one line. We will now test a word of 8 characters long which should not be split and a word of 11 characters long which should be split. Let’s test the 8 character word first:

assert splitter(quote.split()[2])=="tomorrow"

Again this passes. Now for a failing test.

assert splitter(quote.split()[4])=="Austra- lia"

This line fails, so we can now do some real work. We still have all our old tests which should always work, but now these tests make sure that we don’t break anything when we make new changes. This test also clearly defines what we want our function to do. So make the following changes to the splitter function

def splitter(data):

 return data[:6] + "- " + data[6:]

This line returns the first six letter of the word, adds a hyphen and a space which takes the word up to 8 characters and then adds the rest of the word. Let’s run our tests

Oh No the first test now fails, in attempting to fix the problem for Australia we broke all the other solutions. Let’s think about this. We really only want this solution to happen when the word is more than 8 character long, not for every word. However we haven’t added that part into our solution yet. Let’s add that part in. Last session you were shown the function **len** which can find the length of a sequence. Let’s use this function here.

def splitter(data):

 if len(data)<9:

 return data

 return data[:6] + "- " + data[6:]

Now all our previous tests work so we haven’t broken anything, but our Australia test is still broken. It’s time to see exactly what our function is producing. To do this let’s print out the result for this test

print(splitter(quote.split()[4]))

Once we do this we see what is wrong, we were using the wrong data in the test, we forgot about the full stop. We can now fix the test and we get the all clear.

Our function now works perfectly with all our test data. We could now sell it to Amazon, who are keen to add this functionality to their Kindle, and join the ranks of the millionaire programmers.

*Actually our function is not yet production ready. Is 9 the correct number? What do we do for words longer than 16 characters, 24 characters etc. Is hyphen the best way to do the split, does this work in all languages. Amazon aren’t interested yet, but it should work fine for our lcd display.*

To be useful for our lcd display we also need a sentence splitter, since our current lcd library function works at the sentence level. So let’s write that using TDD. Our first test is

assert sentence\_splitter("Its already")=="Its already"

and the code to get that to pass is

def sentence\_splitter(data):

 return data

Now for the real test

assert sentence\_splitter(quote)=="Its already tomorrow in Austra- lia."

Which of course fails and will need some thinking to get right. This solution is not just one line of code. Time to work through the second method, commenting.

**Commenting (sometimes called Pseudo Code**)

Whenever you have a problem like this one solution is to write out my solution is English first before trying to get it to work in Python. Use comments to write your English solution so you can then turn this into code. These comments are sometimes called Pseudo Code since the comments can stand for code.

def sentence\_splitter(data):

 #split up the sentence to words

 #run word through splitter

 #reassemble and return new sentence

 return data

Each of these lines represents one or more lines of code. Let’s start with the first one, the code for this is:

#split up the sentence to words

for word in data.split():

Now that we can get to the words one at a time, time to work on the second issue. The solution for this is

#run word through splitter

splitter(word)

however this does not capture the new data to be use elsewhere so it might be better to use

new\_word = splitter(word)

Now for the last issue. The solution to this is more than one line. We need an empty **String** to start, build it through the loop and return it. This technique was discussed in Session 3. So the lines are:

#reassemble and return new sentence

new\_sentence=""

new\_sentence = new\_sentence + " " + new\_word

return new\_sentence

Now we just need to put them together in the correct order and our function looks like

def sentence\_splitter(data):

 new\_sentence=""

 #split up the sentence to words

 for word in data.split():

 #run word through splitter

 new\_word = splitter(word)

 #reassemble and return new sentence

 new\_sentence = new\_sentence + " " + new\_word

 return new\_sentence

When you run your test it still does not pass, but if you print out the result you see it is very close. It adds as space at the front during the re-assembly stage, which is easy to take out with

return new\_sentence[1:]

This now passes and we have code ready to sell, or at least use in our lcd library. We can now refactor. There is a variable new\_word which is not really needed and you can delete the comments that were only there to help me work out how the function works and some other tidying up. The final function could be:

def sentence\_splitter(data):

 new\_sentence=""

 for word in data.split():

 new\_sentence += " " + splitter(word)

 return new\_sentence[1:]

**Vehicle**

The Zumo has motors and its own power source to run these motors. Let’s now work out how they work.

Create a new folder, copy the PiZumo file into the folder and create a new file and write the following code.

#! /usr/bin/env python3

from PiZumov1B import Zumo

from time import sleep

zumo = Zumo( )

sleep(1)

while True:

 zumo.zumoMotors(200,200)

 sleep(1)

 zumo.zumoMotors(0,0)

 sleep(1)

This turns both motors on to a speed of 200 and leaves them on for 1 second it then turns both motors off for 1 second. The first argument is a number for the left motor and the second is a number of the right motor. These numbers must be between -400 and 400, with negative numbers turning the motor backwards.

There are also functions for working just the left and right motor (leftMotor and rightMotor) which work in a similar way to just control one of the sides of the vehicle.

**Running your vehicle**

The Zumo has three buttons attached labelled A, B and C. We can use these to start and stop our vehicle movement so we can program our vehicle, then move it to a location to start the vehicle movement. The library function to check whether button A has been pressed is buttonAisPressed, which output either **False** or **True**, with **True** being the button has been pressed. So we can move the code which moves the vehicle into an if statement and this means the vehicle moves forward and then stops waiting for us to press the Button A. The final code is

#! /usr/bin/env python3

from PiZumov1B import Zumo

from time import sleep

zumo = Zumo()

sleep(1)

while True:

 if zumo.buttonAisPressed():

 zumo.zumoMotors(200,200)

 sleep(1)

 zumo.zumoMotors(0,0)

 sleep(1)

**Glossary**

|  |  |
| --- | --- |
| **Concepts** | **Meaning** |
| Command | These are key words used by the language that perform a function for the language. It is also possible to create your own commands to be used in your programming. Most languages have libraries of commands that have already been built for you to use as well as those you build yourself. |
| Functions | The most common way to create your own commands. These can accept data, process data in some way and can return the changed data to be used in other parts of the program. In a well structured program almost everything is done in small single purpose functions. |
| Arguments | These are bits of data that are used by functions, so that they have information to work with. |
| Parameters | These are the place holder variables that are used in function definitions. These are replaced by the arguments that are used when the function is run. Often the two terms are used interchangeably.In Python Parameters can be given a default value, which means that an argument is not required when the function is called. |
| String | Data type, String refers to ordinary words.  |
| Integers | Data type, int refers to whole numbers.  |
| Float | Data type, Float used for decimals, but they are not accurate.  |
| Variable | Way of representing data for the program to work on. |
| Libraries | These contain commands that have been developed and tested and are ready to use. Many of the libraries have been written by the people who originally developed the language while other libraries have been developed by companies or individuals that use the language (Google, Yahoo, Apache etc have all developed extensive language libraries for a wide variety of languages). You can also develop your own library of commands. |
| Assignment | Uses = to give a variable name a value |
| Expression | An operation which must be true or false |
| Comments | These are notes for people reading the code. |
| Array | Structure used to group data together, in Python there are many types including List, Set, Tuple, Dictionary etc. |
| Object methods | All Python objects have built in methods to make them more useful. These include |
| String concatenation | Adding to strings together to make one string. This can be done using the + sign. |
| Slicing | Python methods for getting inside sequences |
|  |  |
| **Python Commands** | **Meaning** |
| print | Python function which will show the data on the console |
| def | Python key word used to define your own functions |
| input | Python function which gets data from the keyboard |
| int | Python function which changes a String to its Integer value |
| if | Python keyword used to build decision making structures, must be followed by an expression which is either true or false |
| from … import | Python keywords used to bring in Python objects and function from other files for use in your current file. |
| while | Python keyword which creates a loop based upon the expression |
| True | Python keyword which always evaluates to true, there is also a keyword False. |
| for … in  | Python keyword which creates a loop based upon the iterable structure that it is given. in is also a Python keyword to look inside Arrays and can be used in expressions |
| range | Python function which is used to create a sequence of numbers |
| .append | Python method which adds an element to a list |
| .split | Python method which changes a string to a list |
| enumerate | Python function which is used in for loops to provide both the data and the index of the data |
| len | Python function which finds the length of a sequence |
| assert | Python keyword which is followed by an expression, will throw an error if the expression is false |
| pass | Python keyword which is a placeholder, used to create function without filling in the exact implementation details yet |
| None | The value you turn from a function which does not contain a return statement. None evaluates to false but can also be tested for |
|  |  |
| **Best Practise** |  |
| DRY Principle | Don’t Repeat Yourself. When you need the same code in two places make a function, never copy and paste. This way if there are errors they will only be in one place. Makes you code much easier to maintain and change later on. |
| TDD | Test Driven Development a way of programming where you write your tests first. |
| Refactor | Once we have written our code and it works (with tests to prove it works) we can look at what we have done and make our code better. |
| Pseudo Code | Writing out in English inside comments what you want to do so that you can then turn this explanation into Python code. |