**Comments, functions and scope**

The last example we wrote in conditionals and boolean logic was a bit more complex than anything that came before it. A lot of the scripts we’ll be writing after this point will start to get longer and more involved and so we don’t start getting lost or confused we can use comments to document our code.

**Comments.py**

# Assign the value of 10 to apples

apples = 10

# If the number of apples is less than 10

if(apples < 10):

print("We do not have enough apples.")

# If the number of apples is between 10 and 20

elif(10 <= apples <= 20):

print("We have enough apples.")

# If the number of apples is greater than 20

else:

print("We have too many apples.")

Comments are declared with a ‘#’ symbol and anything after it is ignored by the interpreter. In the above example you can see they allow us to explain in simple terms what is going on in the script, though once you become more familiar with simple operations you may find that documenting absolutely everything is unnecessary.

**Functions**

Functions are incredibly useful, allowing us to package up blocks of code that can be executed whenever we want.

We’ve actually been using some functions already. ‘print()’ and ‘input()’ are functions built-in to python that deal with input and output of data.

The following script is a function in its simplest form.

**Function.py**

# Defining a new function

def my\_function():

print("Calling: my\_function()")

# Calling, or executing, the function

my\_function()

All functions begin with ‘def’ followed by the name of the function, then a pair of brackets with a colon at the end. The code that the function will run is indented inside. We ‘call’ or execute a function by simply stating its name with a pair of brackets.

This example is a bit too simple though. We can do a lot more with functions than simply printing out text. Consider a function like a box that performs a set task. We can give it some input that will be operated on before getting some output on the other side.

To give a function input we simply place our variables or values in the brackets after the function name, each input separated by commas. When we wish to give back some output we can use ‘return’ to do so.

**Double.py**

my\_num = 10

# This function takes a number and doubles it

def double(number):

return number \* 2

print("Before doubling:", my\_num)

my\_num = double(my\_num)

print("After doubling:", my\_num)

Here we give our ‘double’ function a number. It returns the number multiplied by 2. In order to get this output properly we assign the doubled number to ‘my\_num’. We could duplicate line 7 to double our number several times if we wanted to.

Something else to consider is that we can give functions as input to other functions so long as they return a value.

def double(number):

return number \* 2

# Double a number 3 times: 10 -> 80

print(double(double(double(10))))

Functions are quite versatile and powerful tools because of this. Just remember that when we define a function we are basically creating a template that is given placeholder values, such as ‘number’ in our example above. When we ‘call’ or execute the function we replace those placeholder values with actual values, such as the number 10.

**Scope**

This is something we should address before going any further. Up until now the variables we’ve been using in our scripts have been global variables, meaning they can be accessed anywhere within the script it was declared. But variables declared inside a function can only be accessed within that function. These are known as local variables.

**Scope.py**

# Global variable called my\_var

my\_var = 10

def print\_number():

# Local variable called my\_var

my\_var = 5

print("print\_number():", my\_var)

print\_number()

print("global:", my\_var)

The variable ‘ my\_var’ declared inside ‘print\_number()’ only exists inside that function and is completely unrelated to ‘my\_var’ declared at the top. When you run the script you should see that each variable is a different value.

Just like how we can’t access a function’s local variables from outside, we also can’t access global variables from within functions. Normally we would pass the values the function needs by using the parenthesis next to its name but there is a way we can access globals from within functions, though I would advise against this method. To do this you simply declare:

global my\_var

at the top within ‘print\_number()’, but again I would try not to do this if possible. By changing variables elsewhere in your script you could lose control of what data is changing which might cause strange errors down the line. Even if you only wanted to reference a global variable you can easily do this by passing the values through the parenthesis anyway.

**Why functions are useful**

If it is still a little unclear as to why we use functions at all then the next example may help you make some sense of them.

**Circumference\_nofun.py**

pi = 3.14159

radius\_one = 46

radius\_two = 12

radius\_three = 308

radius\_four = 88

radius\_five = 299

circ\_one = int(2 \* pi \* radius\_one)

area\_one = int(pi \* (radius\_one \*\* 2))

circ\_two = int(2 \* pi \* radius\_two)

area\_two = int(pi \* (radius\_two \*\* 2))

circ\_three = int(2 \* pi \* radius\_three)

area\_three = int(pi \* (radius\_three \*\* 2))

circ\_four = int(2 \* pi \* radius\_four)

area\_four = int(pi \* (radius\_four \*\* 2))

circ\_five = int(2 \* pi \* radius\_five)

area\_five = int(pi \* (radius\_five \*\* 2))

result\_one = "Circumference: {}\nArea: {}".format(circ\_one, area\_one)

result\_two = "Circumference: {}\nArea: {}".format(circ\_two, area\_two)

result\_three = "Circumference: {}\nArea: {}".format(circ\_three, area\_three)

result\_four = "Circumference: {}\nArea: {}".format(circ\_four, area\_four)

result\_five = "Circumference: {}\nArea: {}".format(circ\_five, area\_five)

print("\nCircle radius: {}\n{}".format(radius\_one, result\_one))

print("\nCircle radius: {}\n{}".format(radius\_two, result\_two))

print("\nCircle radius: {}\n{}".format(radius\_three, result\_three))

print("\nCircle radius: {}\n{}".format(radius\_four, result\_four))

print("\nCircle radius: {}\n{}".format(radius\_five, result\_five))

This script calculates the circumference and area of 5 circles. As you can see it is relatively straightforward but it is a little lengthy since a lot of code is repeated. The code that calculates the circumference and the area, and the code that prints out the results could be bundled up into functions, code templates that can be reused as often as necessary. Take a look at the next script. It does exactly the same thing as the previous but it is a lot cleaner, much easier to read and if we wish to analyse more circles we only need to add one extra line.

**Circumference.py**

pi = 3.14159

def circumference(radius):

global pi

return int(2 \* pi \* radius)

def circle\_area(radius):

global pi

return int(pi \* (radius \*\* 2))

def print\_circle\_info(radius):

result = "Circumference: {}\nArea: {}".format(circumference(radius), (circle\_area(radius)))

print("\nCircle radius: {}\n{}".format(radius, result))

print\_circle\_info(46)

print\_circle\_info(12)

print\_circle\_info(308)

print\_circle\_info(88)

print\_circle\_info(299)