# Python_Introduction_Part_3

October 20, 2020

Week 3

This week we will be examining the basic functionality that enables you to build up more complex algorithms, which will help you to create routines that can begin to do set tasks; and help us to improve our work-flow.

This involves working with control statements and developing your own functions.

---

## 1 Conditional Statements

In order to create programs that are able to perform useful tasks for us, we will need to invoke conditional statements and loops (in combination with relational and logical operators). I have already used a couple in previous sections.

### 1.1 The *if* statement

The *if* statement is a basic conditional statement.

The format is:

```
if condition:
    expression
```

Here is a basic example:

[1]:
```
x=2
if x==2:                # If x equals 2 then carray out expression
    print('Hi there')   # This line has to be indented
```

```
Hi there
```

The *if* statement basically evaluates the expression in the condition. If the expression is True, then the following indented lines of code are run. Here, if the expression is false, then nothing happens.

## 1.2 if else

You may actually want one of two actions to be performed depending on whether the condition is true of false. We will then need to include the *else* statement

[5]:
```python
bird1=0
bird2=0

spotted='pidgeon'

if spotted=='pidgeon':
    bird1 +=1   # This operation adds 1 onto the previous value of the array
else:
    bird2 +=1

print(bird1,bird2)
```

```
1 0
```

Change the spotted variable to ‘crow’ and see what happens.

The above code can be thought of as *if (logic satisfied) do task1 else do task2* .

## 1.3 if else elif

Sometimes there are more than 2 choices, and we can use the *elif* statement

[1]:
```python
choice = 'cat'

if choice == 'cat':
    print("I like cats too! We should be friends.")
elif choice == 'dog':
    print("Dogs are okay...")
elif choice == 'rat':
    print("Who likes rats?!")
else:
    print("That pet was not an option.")
```

```
I like cats too! We should be friends.
```

Change the variable choice to either ‘dog’ or ‘rat’ to see what happens.

## 2 Loops

### 2.1 *for* loop

A *for* loop will loop over a section of code for the number of times specified. The *for* loops in Python might look slightly different from those in other languages, e.g., Matlab, IDL. The *for* loops in Python are 'collection-based' and they iterate over a collection of objects (see here for additional detials ).

A simple example is:

[7]:
```
for x in range(5):  # object with elements [0,1,2,3,4]
    print('The number is ', x)
```

```
The number is  0
The number is  1
The number is  2
The number is  3
The number is  4
```

The range function here is similar to *arange* in numpy - it generates a sequence of evenly spaced samples. The *for* loop then iterates over the integers.

Here is another example with strings in a list:

[40]:
```
a = ['apple', 'banana', 'carrot']

for fruit in a:     # Note that you can name the iterator what you
    print(fruit)    # like, changing fruit to, e.g. goat will not
                    # effect the execution.
```

```
apple
banana
carrot
```

In both of these cases we have created an *iterator*, which provides a consistent way to iterate over sequences. Here is another example with a dictionary (similar to an example shown previously in week 1):

[2]:
```
some_dict = {'a': 1, 'b': 3, 'c': 5}

for keys in some_dict:
    print(keys)
```

```
c
b
a
```

The code written above does not allow of to iterate over keys and values in the dictionary. You can try a for loop of the form:

```
for keys,values in some_dict:
```

however, it will not work! To iterate over both keys and values we need to use the method dict.items()

[4]:
```
for keys, values in some_dict.items():
    print(keys, values)
```

```
c 5
b 3
a 1
```

You can also control the flow of *for* loops with *break* and *continue* statements.

A break statement will cause a loop to terminate. In the following example we will create a loop break when a conditional statement is satisfied.

[9]:
```
for x in range(5):
    if x == 3:         # If our iterator gets to 3 then break
        break
    print('The number is ', x)
```

```
The number is  0
The number is  1
The number is  2
```

A continue statement terminates the current iteration but the loop continues.

[10]:
```
for x in range(5):
    if x == 3:
        continue
    print('The number is ', x)
```

```
The number is  0
The number is  1
The number is  2
The number is  4
```

You may also want to keep track of the iteration number during the process. A neat way to do this is use the function enumerate.

[3]:
```
for index, keys in enumerate(some_dict):
    print(index, keys)
```

```
0 c
1 b
2 a
```

### 2.2 While loops

Like the *for* statement, the *while* statement starts a loop. However, the looping does not stop after a set number of iterations. It only stops when a given conditional statement is False, i.e., loops while it is True. Essentially, the while loop can be thought of as a looping *if* statement.

```
[1]: prompt = "Can you guess what number I am thinking of? "

     number=input(prompt)            # The input function enables some user␣
      ↪interaction
                                     # - number variable is string
     while number != "42":           # This says ' loop while number is not equal to␣
      ↪42'
         number =  input('No, keep going! ')

     print("Yes, well done!")
```

```
Can you guess what number I am thinking of? 2
No, keep going! 22
No, keep going! 42
Yes, well done!
```

It is worth noting that a *while* loop can potentially go on forever!! This will likely be due to poor construction of the loop. If you find yourself stuck in a while loop, then you can stop the execution with ‘ctrl-c’.

---

# 3 Functions

So far, we have been using various pre-existing functions in Python. However, we can, and will often want to, develop our own functions to perform specific tasks. There are two methods for defining functions. The second method is discussed in Section **??**.

### 3.1 In-line functions

The first method is one that can be done ‘in-line’, in the interpretor.

We have to use the *def* keyword when defining the function. This is then followed by the name of the function and parenthesis, between which you give any parameters for the function.

```
[1]: def welcome_me(name):         # function name - welcome_me, parameter - name
         print('Hello '+ name)      # These two lines are the function definition

     welcome_me('Richard')         # Here I am calling the function
```

```
Hello Richard
```

Information is passed into the function as an argument. Here, *name* is the parameter and ‘Richard’ is the argument.

The function can also return information, and that information be stored in variables.

```
[9]: def square_number(x):
         return x**2

     y=square_number(4)  # Here x in the function takes on the value 4
     print(y)
```

```
16
```

In theory, you can have as many arguments as you want, but when inputting them you will have respect the order they are input.

You can also assign arguments by their names, and here order doesn’t matter.

```
[3]: def subtract_numbers(x, y):
         return x-y


     # Here x in the function takes on the value 4 and y is 3
     z = subtract_numbers(4, 3)
     print(z)

     # Here x in the function takes on the value 3 and y is 4
     z = subtract_numbers(3, 4)
     print(z)

     z = subtract_numbers(y=3, x=4)  # assign argument by name
     print(z)
```

```
1
-1
1
```

### 3.1.1 A small aside

Note that depending upon whether the argument that is passed to the function is mutable or immutable, one of two things can happen.

In programming, arguments passed to functions are either *passed-by-reference* or *passed-by-value*.

If the arguments are passed-by-reference, the function gets an explicit reference to the argument (rather than a copy) - hence if your function changes what the argument variable refers to, the change reflects beck in the calling function. This is the case for mutable data types.

This is slightly technical, but is illustrated with a simple example.

Remember, the *id()* function gives us the unique id number for the Python object.

```
[30]: def change_list(lis):
          lis.append([1, 2, 'cat'])
          print('Inside function ', id(lis))
          return


      lis = ['dog', 3, 4]
      print('Outside function ', id(lis))
      change_list(lis)
      print(lis)
```

```
Outside function  4377955720
Inside function  4377955720
['dog', 3, 4, [1, 2, 'cat']]
```

For immutable data types, e.g., integers, the arguments are passed-by-value. Hence, the function effectively creates a copy of the variable to use within function. There is no change to the variables outside the function.

Again, a simple example highlights this.

```
[31]: def multiply_number_by_five(x):
          x *= 5
          print('Inside function ', id(x))
          return x


      x = 4
      print('Outside function ', id(x))
      print(multiply_number_by_five(x))
      print(x)                            # x is unchanged
```

```
Outside function  4316223952
Inside function  4316224464
20
4
```

Further reading on this can be found here https://www.python-course.eu/python3_passing_arguments.php

## 3.2 Returning multiple values

A function can also return multiple values, and there are a number of ways to access these multiple values. The next two examples use the same function but show two different ways to access the values.

In this first example we provide a single variable to assign the function output to. The result is that the variable is a tuple.

[33]:
```python
def issac_details():
    name = "Issac Newton"
    dob = "4th January 1943"
    pob = "Woolsthorpe"
    return name, dob, pob


details = issac_details()
print(details, type(details))
```

```
('Issac Newton', '4th January 1943', 'Woolsthorpe') <class 'tuple'>
```

This time, we provide 3 variables to assign the function output to.

[36]:
```python
name, dob, place = issac_details()
print(name)
print(dob)
print(place)
```

```
Issac Newton
4th January 1943
Woolsthorpe
<class 'str'>
```

Note that the first method returns a tuple, hence we can't alter the contents of elements in it (tuples are immutable).

## 3.3 Global vs local variables

Not all variables are accessible everywhere in a script.

Variable names in the main body of script, i.e. defined outside of functions, are called *global* variables (said to be in the global scope).

Local variables are defined within the function (local scope). The local variables cease to exist once the function has finished.

Depending on how the function is defined, the function can access these global variables.

First the local scope is checked to see whether a variable exists, then the global scope.

The following function will use global variables:

[38]:
```python
def print_x():
    print(x)      # x is not defined in the function


x = 10
print_x()
```

```
10
```

In the following function, we create a *local* variable in the function (through the argument definition). Even though it has the same name as the global variables, Python assumes we want the local variables - hence gives us an error

[39]:
```
def print_x(x):     # x is defined in the function header,
    print(x)        # so a local variables exists.



x = 10
print_x()
```

```
       ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-39-d628756ed6ce> in <module>()
         3
         4 x=10
   ----> 5 print_x()


       TypeError: print_x() missing 1 required positional argument: 'x'
```

The following will also give an error, even though the local variable is defined after the print statement.

[40]:
```
def print_x():
    print(x)
    x = 20


x = 10
print_x()
```

```
       ␣
↪---------------------------------------------------------------------------

       UnboundLocalError                         Traceback (most recent call␣
↪last)

       <ipython-input-40-f4a48215c6c0> in <module>()
         4
         5 x=10
   ----> 6 print_x()
```

```
    <ipython-input-40-f4a48215c6c0> in print_x()
      1 def print_x():
----> 2     print(x)
      3     x=20
      4 
      5 x=10


    UnboundLocalError: local variable 'x' referenced before assignment
```

What happens if we now just define a *local* variable in the function that doesn't correspond to a global variable.

```
[42]: def f(y):
          temp = y


      x = 10
      f(x)
      print(temp)
```

```
      ␣
 ↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-42-16e4a868b0c3> in <module>()
          4 x=10
          5 f(x)
    ----> 6 print(temp)


        NameError: name 'temp' is not defined
```

I tried to print the local variable *temp* after running the function, however, we get an error saying it is not defined. This is because temp only exists inside the function and is effectively destroyed once the function ends.

## 3.4 Optional Parameters

Functions can also have optional (or default) parameters - they don't have to be given if the function is called.

Writing a function that requires too many arguments to passed can make its usage cumbersome. In many situations you will want the function to perform a specific task, but also allow it to be customisable when the need arises. Hence, one way to achieve this flexibility is to define default arguments. The default arguments are given by a keyword and the default value.

In the following, I define an optional parameter defined by the keyword *value* with the default value of 1. First, I don't give the function anything for *value*. I then provide the optional parameter in two ways.

[45]:
```
def price_of(value='£1000'):
    print('The price of an Ipad is '+value)


price_of()
price_of('£500')
price_of(value='£400')
```

```
The price of an Ipad is £1000
The price of an Ipad is £500
The price of an Ipad is £400
```

You can also define the argument as None - which is a null value. Note, this is not the same as 0.

[9]:
```
def price_of(value=None):
    if value is None:
        value = '£500'

    print('The price of an Ipad is '+value)


price_of()
price_of('$200')
```

```
The price of an Ipad is £500
The price of an Ipad is $200
```

### 3.5 Flexible arguments

You may also want to write a function that takes in a number of arguments, but you don't know in advance how many will be given.

To do this we add _*args_ to the function header. This then passes all arguments as a tuple to the function body.

[1]:
```
def names(*args):
    name_string = ''         # initialize empty string
    for name in args:       # Loop over variables in tuple
        name_string += name
    return name_string


names('Richard', 'Jeff', 'David', 'Brian')
```

[1]: 'RichardJeffDavidBrian'

You can also do something similar for keyword arguments. This time, however, you require a double star, _**kwargs_. The keywords and values are then turned into a dictionary in the function body.

[4]:
```
def some_func(**kwargs):

    for keys, values in kwargs.items():
        print("Key: "+keys+', Value:', str(values))


some_func(Lost='cat', Reward='£500')
```

```
Key: Reward, Value: £500
Key: Lost, Value: cat
```

Note, it is not the names args and kwargs that make this work - it is the single and double asterixes. However, using the names args and kwargs is standard practice.

## 4 Modules & Imports

We have already discussed running a list of instructions in a Python file (i.e. a file ending in *.py*).

However we can also create and use modules (or packages), which are formatted .py files.

We have already come across modules, such as numpy, and modules are imported as:

```
import some_module
```

There are many such modules, each containing special functions, methods or types and are designed to undertake a particular problem.

We can create our own modules simply. Enter the following in a new python file and save it as test_module.py

[ ]:
```
# test_module.py
const = 4.67


def addFour(x):
    return x+4


def multip(a, b):
    return a*b
```

We can load the module in with import, i.e.

```
import test_module
```

and use the constant and functions in the module as such:

```
res=test_module.addFour(7)

print(test_module.const)
```

We can also import the functions as

```
from test_module import addFour, multip, const

res=multip(5,const)
```

Or we can do:

```
import test_module as tm

res=tm.addFour(tm.const)
```

## 5 Error handling & exceptions

A final word in this notebook is on error messages.

As we have seen, Python will throw error (or exception) messages if you do something incorrectly. The error messages are useful for understanding what has gone wrong, and will help you correct any problems with your code.

Any time you see an error message you don't understand - trying googling it.

They may appear cryptic, but will help you fix any issues with you code! The quicker you start to use and understand the error messages, the easier it will be diagnose and fix problems.

Take the following:

```
[5]: float('1.234') # turns string to a float
```

```
[5]: 1.234
```

```
[6]: float('word')
```

```
           ␣
 ↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
 ↪last)
```

```
    <ipython-input-6-cb46d0ec6998> in <module>()
----> 1 float('word')


    ValueError: could not convert string to float: 'word'
```

Here a ValueError has been raised because the float function cannot convert the string ‘word’ to a number.

You can also add error and exception handling to your functions. The following is called a *try/except* block.

[14]:
```
def add(a,b):

    try:
        return a+b
    except:
        return 'Check arguments are both integers or floats'

add(1,'y')
```

[14]:
```
'Check arguments are both integers or float'
```

The except part is only executed if the try part raises an exception.

[ ]:
```
```