Python Primer

This section will give a bit of background information about Python syntax.

We will start with the concept of a variable, which is merely a representation of a changeable value. There are a few special words which you cannot call a variable, such as `class`, otherwise most combinations of letters, numbers and underscores can be used. Variables are case sensitive.

Learning Outcomes

On completion of this brief introduction to Python participants should be able to identify and understand use of:

- variables
- functions
- code blocks
- boolean operators
- loops
- lists
- dictionaries
- data retrieval

Variables

At the outset, the first thing we need to emphasise is the concept of *assignment* of a variable. When we use the = symbol, we are assigning a value to an existing variable. This is not the same meaning as in a mathematical equation which implies the two sides of the sign are equal. The outcome of this is that you can have valid statements such as `x = x + 1`, which would increase the value of `x` by 1, rather than being a statement of truth.

In Python, and many modern programming languages *equivalence* is represented as == i.e., two equals signs. So, by saying `x == x`, you are making a (correct) statement about the variable `x`. The output of this statement therefore would be `True`. If by contrast, you were to say `x == x + 1`, this is another statement, which is incorrect. The outcome of this would therefore be `False`.

Have a look at the code below for some examples. You can run code in the browser by selecting the cell, and then pressing `Crtl` + `Enter`, or by using the "play" button on the menu above.

In [2]:

```
# an example of a dataframe in which python code can be executed in a browser

# variable x is assigned value 10
x = 10

# variable y is assigned value 20
y = 20

# print the result of expressing that x is equal to y (False as x does not equal y)
print(x == y)

# variable 'text' is assigned the string 'x * y ='
text = 'x * y ='

# print the value of variable 'text' followed by the sum of 'x * y', each of which l
print(text, (x * y))

# change the value of 'x' by assigning the value 20
x = 20

print(text, (x * y))

# print the result of expressing that x is equal to y (True as x does equal y)
print(x == y)

# print the output of the `print` function
print(print(x== y))
```

```
False
x * y = 200
x * y = 400
True
True
None
```

There we have an example of different types of variables: integers and a string. The string is surrounded by either `''` or `""`, and can be any combination of numbers and letters.

Integers are just defined as they are. The same can be said for the `float` type, which is any real number, though very large numbers are restricted.

We also used booleans, which are simple True/False values. Boolean statements are known as conditionals, and are used to decide whether to run certain blocks of code.

Finally, we are using the `print` function to output the value to the screen. All the values we want to output are separated by the commas within parentheses. These are called arguments, and will be used within the function. Like a mathematical function, as these values change, so too does the output. This has a type of `None` which is the Python equivalent of `null`. Note that this is different from `0`, which is a value, whereas `None` means no value.

Blocks of Code and Booleans

In Python, the indentation of code is important. So important, that, if it is not correct it will not run at all. Most programming environments which support Python will do this automatically, but it is important to know to expect it. Each new "block" of code MUST be indented by the same amount of spaces.

An important example of this is where we use an `if` statement to decide whether to run the next bit of code. If statements evaluate the result of a boolean variable (`True` or `False`).

Rather than just using `if`, we can also decide what to do in other situations. So, we have

```
if condition_a:
    .....
elif condition_b
    ....
elif condition_c
    .....
else
    .....
```

The flow of this would be as follows.

1. Check for a certain condition
2. If the first condition does not apply, try other conditions - first condition_b and then condition_c
3. If none of these conditions apply, then fall back on the action for all other conditions.

Look at the example below, and try changing the values to see how if/else statements work in practice.

In [3]:

```
x = 0
if 0 == 1: # Will never run!`
    print("If this is running, something has gone wrong!")
# as 0 does not equal 1 the 'else' block runs
else:
    print("If this is running, then we can relax!")

    # assigning the value of 42 to the variable 'x'
    x = 42

print(x)

# We can use variables to make these checks as well
x = 42
if x > 42:
    print("x is greater than 42")
elif x == 42:
    print("x is equal to 42")
else:
    print("x is less than 42")
```

```
If this is running, then we can relax!
42
x is equal to 42
```

Booleans

We dealt with `if` statements in the previous cell, which use the outcomes of truth statements to help them to decide. The first example of `0 == 1` is an equivalence test, which will always be false. No matter what, 0 will never be equivalent to 1. There are other operators used ot make these decisions, as follows:

`!=` means not equal to. I.e.,

In [4]:

```
print(0 != 1, 0 != 0)
```

```
True False
```

Mathematical operators like >=, >, <=, < have the usual meanings, where > means greater than and < means less than.

In [5]:

```
print('These should be true')
print(42 > 40, 42 >= 42, 42 < 64, 42 <= 42)
```

```
These should be true
True True True True
```

Python also has the concept of "truthy" or "falsy" conditionals. Various values will default to either `True` or `False`. Values such as `0`, `''`, `None` will default to `False` and most others will default to `True`.

It is generally considered to be good practice to not try and evaluate a variable compared to `True` or `False`, but rather to take advantage of these truthy or falsy variables. Consider the following:

In [6]:

```
truthy_var = 1
if truthy_var:
    print('truthy_var evaluated to %s' % str(bool(truthy_var)))

falsy_var = None

if not falsy_var:
    print('falsy_var evaluated to %s' % str(bool(falsy_var)))
```

```
truthy_var evaluated to True
falsy_var evaluated to False
```

Finally, we can also combine booleans into statements using `and` or `or`. These have their usual logical meaning: If both conditions are true, then `and` will evaluate to `True`. If one or both is `True`, then an `or` statement will evaluate to `True`.

The use of the word `not` will reverse the output. So the statement `not True == False` evaluates to `True`.

It is also possible to combine `and` and `or` statements, and these can be included in parentheses so those grouped together will be evaluated together. Consider the following code:

In [7]:

```
truthy_1 = True
truthy_2 = 1
falsy_1 = False
falsy_2 = 0

if truthy_1 and falsy_1:
    print('This should never run')


if truthy_1 and truthy_2:
    print('This should run because truthy_1 and truthy_2 are both truthy')

if truthy_1 or falsy_2:
    print('This should run, because it contains a truthy variable')

if falsy_1 or falsy_2:
    print('This should not run, because both are falsy')

# Try and work out what the output to the code below would be
# Then uncomment and run for yourself to see if you are right
# if not(truthy_1 or (falsy_1 and truthy_2)):
#     print('This is True')
# else:
#     print('This is False')
```

```
This should run because truthy_1 and truthy_2 are both truthy
This should run, because it contains a truthy variable
```

Functions

A function is a self-contained block of code where, by "calling" it, the code within that block will be executed. A function is defined as follows:

```
def super_function():
    """
    A comment about what the purpose of the function is, for documentation p
urposes
    """
    return 42
```

Let's break that down a bit.

- We start off with `def`, which means that you are defining a function
- The function is called `super_function`. Every time someone writes `super_function()` in their code after this, it will execute this function.
- The brackets include any parameters to be passed to the function. In this case, there are no parameters. This will be discussed more further down.
- The `:` means that the function definition is finished. Everything else should be one tab further in.
- The `"""` syntax defines a comment. This is reserved for documenting functions or modules rather than providing notes about a specific bit of code.
- `return 42` means that the output of the function is `42`, like would be the case for a mathematical function. You could assign the output of a function to a variable, as in the code below.

In [8]:

```
def super_function():
    """
    A comment about what the purpose of the function is, for documentation purposes
    """
    return 42

x = super_function()
print(x == 42)
```

```
True
```

The other important element of functions is their parameters. This means that you can put in the names of variables and give them a value when you call the function. Consider the function `hello` defined below. This has a parameter of `name`, and then prints a string. This means that any name you "pass" the function, it will print that.

The term "arguments" is used to describe the actual values passed to the function when it's called, which is slightly different to the "parameters" which are in the function definition

In [9]:

```
# instead of writing `print ("Hello, %s.  I hope you're having a nice day!" % name)
#an output is required the function `hello` is called

def hello(name):
    print("Hello, %s.  I hope you're having a nice day!" % name)

# call the function 'hello', in this case the parameter 'name' is passed to the func
hello ('Timothy')
hello('Bob')
hello('Anna')
```

```
Hello, Timothy.  I hope you're having a nice day!
Hello, Bob.  I hope you're having a nice day!
Hello, Anna.  I hope you're having a nice day!
```

At this point, it is important to think about what happens to the scope of variables connected to the parameters, and variables declared within a function. In general, functions - and any other code block - will take the value of the variable with the smallest scope where there is a choice.

The code below provides an illustration of this in action:

In [16]:

```
x = 'x'
y = 'y'
def foo(x):
    y = 'new y'
    print(x, y)

foo('new x')
```

```
new x new y
```

Loops, Lists, and Dictionaries

The final thing to know about is loops and lists. A loop is another example of a code block, and everything within that block will be executed for a certain set of conditions. The most common way that this might work is by working over each element in a list.

Look at the example below, and run it:

In [17]:

```
listy = [1, '4', 9, 16, 25, '6 x 6', 49, 64, 81, 100, "Have we finished yet?", 0 ==
# the variable 'listy' is assigned a list [., ., ....]

listy.append(True)
# the variable (listy) has element 'True' added to the list

for li in listy:
# itterate through 'listy'

    print(li)
# output to screen members of list 'listy'

print(listy)
```

```
1
4
9
16
25
6 x 6
49
64
81
100
Have we finished yet?
False
True
[1, '4', 9, 16, 25, '6 x 6', 49, 64, 81, 100, 'Have we finished yet?',
False, True]
```

This example shows you three things.

- Firstly, how a list is set out - it is a series of values separated by commas.
- Secondly, it shows how to add an element to the list, `append()` adds an item onto the end
- It shows how to iterate through a list

The format for a loop is similar to English like a lot of Python which is one of the reasons for its popularity. All that's saying is "for each item called li in the list listy do the following things". In this case, we took the variable `li`, and printed it.

Elements of a loop are numbered, so you can access an individual element by using its index. For historical reasons, a lot of programming languages start at `0`, and Python is no exception. So for a list of `n` items, the series of indexes are `0, 1, ..., n-1`.

The first element is accessed like this:

```
var_name = listy[0]
```

It would also be possible, and preferable in some situations, to iterate through the loop accessing elements by index. This uses the `range` function, and can be done as follows:

In [20]:

```
# Run this code
# This is a string output of 'listy' and \n means to add an empty line
print((listy), '\n')

for li in range(2, len(listy)):

    # in 'listy' element 0 is integer value 1, element 1 is string '4', element 2 is
    # the range function counts each element from element 2 onwards i.e. 2, 3, 4, ..

    # This is the index of the list
    print('li has value %d' % li)
    # The index points to the corresponding value at each index point
    print('The value at listy[%d] is: %s' % (li, str(listy[li])))
```

```
[1, '4', 9, 16, 25, '6 x 6', 49, 64, 81, 100, 'Have we finished yet?',
False, True]

li has value 2
The value at listy[2] is: 9
li has value 3
The value at listy[3] is: 16
li has value 4
The value at listy[4] is: 25
li has value 5
The value at listy[5] is: 6 x 6
li has value 6
The value at listy[6] is: 49
li has value 7
The value at listy[7] is: 64
li has value 8
The value at listy[8] is: 81
li has value 9
The value at listy[9] is: 100
li has value 10
The value at listy[10] is: Have we finished yet?
li has value 11
The value at listy[11] is: False
li has value 12
The value at listy[12] is: True
```

This might look a bit complicated, because we're using the output of a function as a parameter. That's okay, we can just evaluate them one at a time.

`len(listy)` takes a list as a parameter, and returns the length of the list. In the case of `listy`, there are 13 elements, so that number is returned. This means that we are calling the `range` function as: `range(13)`.

The `range` function takes a number `n` as a parameter, and returns a list from `0` to `n-1` (there are more options with this function, but we won't go into them here).

Dictionaries

Another Python data structure is a dictionary, which is a series of key/value pairs. The key/values are represented within curly brackets as `{key: value}`, separated by commas. They can be operated on in a similar way to lists, except that the key is taking the place of the index. The keys can be iterated over in the same way as a list. See the following example:

In [21]:

```
dicty = {'abc': 123, 'def': 456, 'ghi': 789}
for d in dicty:
    print(d, dicty[d])
```

```
def 456
abc 123
ghi 789
```

In the example, we print both the key (as `d`) and the value, using `d` as an index. There are a couple of things to be careful of when dealing with dictionaries compared to lists:

- Order is not guaranteed. Just because you add something first does not necessarily mean that it will be iterated first.
- Keys are unique. If we were to add another pair with a key `'ghi'`, it would overwrite the existing value (see below)
- If you try to access a key which is not in the dictionary, it will give an error. We can check this using the format: `if 'ghi' in dicty:`

The following code will illustrate this

In [22]:

```
print(dicty)
dicty['ghi'] = 'Different value!'
print(dicty)

if 'ghi' in dicty:
    print(dicty['ghi'])

if 'jkl' in dicty:
    print(dicty['ghi'])
else:
    print("The key jkl doesn't exist")
```

```
{'def': 456, 'abc': 123, 'ghi': 789}
{'def': 456, 'abc': 123, 'ghi': 'Different value!'}
Different value!
The key jkl doesn't exist
```

Bringing it all together

A function called `get_csv`, which retrieves data stored in a csv file has been written for use with the Python exercise introduced in week 2. The parameters the function takes are:

- `weekday` The day as a digit number, between 1 and 7
- `year` The year as a 4 digit number, between 2012 and 2015
- `month` The month as an integer between 1 and 12
- `amount_of_lines` The amount of lines of the CSV file to return
- `show_headers` Whether to show the headers of the CSV columns. Default value: `False`

Aside: What is a CSV File?

CSV stands for "comma-separated values", and is a simple file format which holds data as a series of rows which are separated by commas. It can be opened by Microsoft Excel but is easier to use for programming because it's just plain text and not a proprietary format.

The following example gives average journey times over route sections for dates and times specified as parameters of a function. Traffic data is made available through the 'Data.Gov.UK' website. As it would not be practical to store this volume of data in a single file the data is stored according to year, month then day of week. The data used in these exercises was downloaded from 'Data.Gov.UK' then 'cleaned'.

In [2]:

```
# from the 'magic' library import all functions listed
from magic import *

# though the function 'get_csv' has been imported from the 'magic' library it is als

def get_csv(weekday, year, month, amount_of_lines, show_headers=False):
    for p in [weekday, year, month, amount_of_lines]:

# check 'p' is an interger, if not display an error message
        if not type(p) is int:
            raise ValueError("Invalid parameter type '%s'.  This parameter should be

# check 'year' is in the range 2012 - 2014 inclusive, if not display error message
    if year < 2012 or year > 2014:
        raise ValueError("Please enter a year between 2012 and 2014")

    if weekday < 1 or weekday > 7:
        raise ValueError("Please enter a group number between 1 and 7")

    if month < 1 or month > 12:
        raise ValueError("Please enter a month number between 1 and 12")

    # you should recognise that 'prettify_month' as a function - check by opening th
    month = prettify_month(month)

    return_string = ''

    # The 'with open' block allows a file to be pointed to at the location '/srv/jup
    # Within the block, the file is opened, Python deals with closing it automatical
    with open('/srv/jupyterhub/data/group%d/%s-%s.csv' % (weekday, year, month)) as
        reader = csv.reader(f)
        i = 0
        for row in reader:
            if row[1].strip() == 'Date' and not show_headers:
                continue
            if i > amount_of_lines:
                break
            i += 1

            return_string += '%s\n' % str(row)
        return return_string

# the variable 'stringy' is assigned the results of calling function 'get_csv' which
# the selection parameters passed to the function are weekday (1-7), year (2012-201

stringy = get_csv(1, 2013, 10, 10, True)

# the headers for each column are shown in the first row,
print(stringy)

# in this instance Linkref refers to a route section, Date and TimePeriod refer to w
# AverageJT refers to average journey time
```

```
['LinkRef', 'Date', 'TimePeriod', 'AverageJT']
['4', '2013-10-07', '00:00', '67.78']
['4', '2013-10-07', '00:15', '71.20']
['4', '2013-10-07', '00:30', '72.43']
['4', '2013-10-07', '00:45', '73.97']
```

```
['4', '2013-10-07', '01:00', '71.16']
['4', '2013-10-07', '01:15', '71.74']
['4', '2013-10-07', '01:30', '75.13']
['4', '2013-10-07', '01:45', '73.33']
['4', '2013-10-07', '02:00', '77.80']
['4', '2013-10-07', '02:15', '79.28']
```

Hack the code

Print journey times for Thursdays in Febuary, 2012. You can do this by editing the code in the cell above or inserting a new cell below with appropriate code.