Introduction to Python - Lesson 2

Matteo Sani
matteosan1@gmail.com

October 11, 2019

1 Introduction to Python

1.1 Recap

In the last lesson the basic concept of Python programming have been looked at

- `print` statements and variables
- mathematical expressions
  - first import of a module (`math`)
- boolean expressions
- string expressions
- indentation, `if/elif/else` blocks and loops
- lists
- dictionaries

1.2 Overview

This time we will look at:

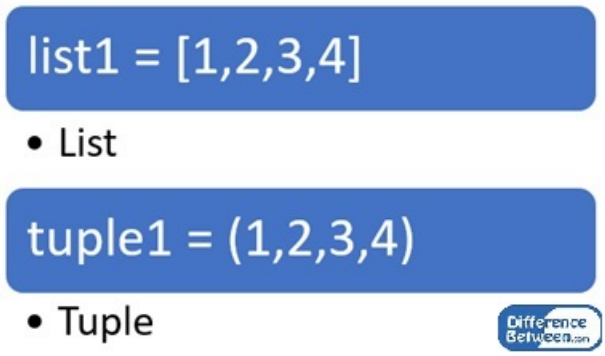
- tuples
- dates
- functions and modules (how to write your own module)
- few exercises for next week

1.3 Tuples

Tuples create a bit of confusion for beginners because they are very similar to lists but they have some subtle conceptual differences. Nonetheless, tuples do appear when programming in Python so it's important to know about them.

Like lists, tuples are containers of any type of object. Unlike lists though they are *immutable* which means that once they have been created the content cannot be changed (i.e. no append, insert or delete of the elements). Furthermore since they are immutable they can be used as dictionary keys (lists cannot).

```
In [ ]: atuple = (1, 2, 3)
        atuple
```



At first glance list and tuples look very similar, but they are not...

```
In [ ]: print ("Length: {}".format(len(atuple)))
        print ("First element: {}".format(atuple[0]))
        print ("Last element: {}".format(atuple[-1]))

In [ ]: x, y, z = (10, 5, 12) # this is called unpacking and it's another
                               # way to access tuple elements
        print ("coord: x={} y={} z={}".format(x, y, z))

In [ ]: tuple2 = (1,) # this is tricky, don't forget the comma otherwise
                      # it won't be a tuple
        print(type(tuple2))
        tuple2 = (1)
        print(type(tuple2))

In [ ]: tuple1 = (1, 2, 3)
        tuple2 = tuple1 + (4, 5) # to add elements you need to create a new tuple
        tuple2

In [ ]: # as said they can be used as dictionary keys
        d = {
        ('Finance', 1): 'Room 8',
        ('Finance', 2): 'Room 3',
        ('Math', 1): 'Room 6',
        ('Programming', 1): 'IT room'
        }
        d
```

1.4 Dates

Dates are not usually included in a standard beginner Python tutorial, however since they are pretty essential for finance we are going to cover this topic. In Python the standard date class lives in the `datetime` module. We are also going to import `relativedelta` from the `dateutil` module, which allows us to add/subtract days/months/years to dates.

```
In [ ]: from datetime import date
        from dateutil.relativedelta import relativedelta
```

```
In [ ]: date.today()

In [ ]: date.today() + relativedelta(months=2)

In [ ]: date.today() - relativedelta(days=3)

In [ ]: one_day = relativedelta(days=1)
        date.today() - 3 * one_day

In [ ]: date1 = date(2019, 7, 2)
        date2 = date(2019, 8, 16)
        (date2 - date1).days

In [ ]: print (type(date1))
        print (date1)
        date1 = str(date1) # converts (or 'cast') a date to a string
        print (date1)
        print (type(date1))

In [ ]: date1 = date(2019, 7, 2)
        date1.strftime("%Y-%b-%d (%a)") # dates can formatted in many ways
                                        # check the docs for more details

In [ ]: # a string can be convered to dates too
        from datetime import datetime
        datetime.strptime('25 Aug 2019', "%d %b %Y").date()

In [ ]: date1.weekday() # 0 = monday, ..., 6 = sunday
```

1.4.1 Exercise 2.1

Write code in the notebook to:

- print the day of the week of your birthday
- print the weekday of your birthdays for the next 120 years

(expected output: Sun 1 Mon 2 Sun ... 119 Thu 120 Sun)

1.5 Functions

A function is a block of organized, reusable code that is used to perform a single action. Functions provide better modularity for your application and high degree of code reusing.

```
In [ ]: # sum up all the integers between 1 and n
        def my_function(n): # this function take one input only (n)
            x = 0
            for i in range(1, n+1):
                x += i
            return x # the function returns a number
```

```
In [ ]: my_function(5) # 5 + 4 + 3 + 2 + 1
```

To be clear functions can return any kind of objects (numbers, strings, lists, complex objects...) but it is not mandatory, so you can write a function **without** a `return` statement.

```
def printing(mystring):
    print (myString)
```

In addition the syntax of the `return` is different from `Visual Basic`, the returned object doesn't have to have the same name as the function.

```
In [ ]: def my_function_2(n, x):
            return "The result is : {}".format(str(my_function(n)*x))
        # returns x * result of my_function(n)
        # so function of function
```

```
In [ ]: my_function_2(5, 10)
```

Functions can also call themselves too (i.e *recursion*). In the next example we write a function that computes the factorial exploiting the following relationship:

$$n! = n \times (n-1)! \quad (\forall n > 1)$$
$$n! = 1 \quad (\forall n \leq 1)$$

```
In [ ]: def factorial(n):
            if n <= 1:
                return 1
            else:
                return n * factorial(n-1)
```

```
In [ ]: factorial(10)
```

Functions can accept and return any values and arguments can have default values.

```
In [ ]: def powers(xs, n = 2, c = 0):
            return {x: x**n+c for x in xs}
```

```
In [ ]: powers([5, 11, 6]) # calling powers like this results in 25, 121 and 36
                           # since n = 2 and c = 0 by default if not specified
```

A nice feature of Pyhton functions is that we can associate an help message to them so that we can easily check what a function is for by simply asking

```
help(function_name)
```

```
In [ ]: def powers(xs, n=2, c=0):
            ''' a shifted power function example in lesson 2

            x ** n + c

            '''
            return {x: x ** n + c for x in xs}
```

```
In [ ]: help(powers)
```

1.6 Advanced - Variable scope

Not all variables are accessible from all parts of our program, and not all variables exist for the same amount of time. We call the part of a program where a variable is accessible its scope.

A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file. Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them (usually they are represented by an uppercase name). Only objects which are intended to be used globally, like functions and classes, should be put in the global namespace.

Global variables can be accessed directly inside a function but cannot be modified. To modify them you have to use the keyword `global`:

```
In [ ]: AGLOBALPARAM = 10

        # Here you just use AGLOBALPARAM value, but do not modify it
        # param is just a copy of AGLOBALPARAM
        def multiplyParam(param):
            param = param * 10
            return (param)

        # Here you actually use AGLOBALPARAM
        # you modify it directly
        def divideParam():
            global AGLOBALPARAM
            AGLOBALPARAM = AGLOBALPARAM / 10
            return (AGLOBALPARAM)

        # Here you try to use AGLOBALPARAM but gives you an error
        # it is not accessible !
        def sumParam():
            AGLOBALPARAM = AGLOBALPARAM + 10
            return (AGLOBALPARAM + x)

        print ("AGLOBALPARAM is {} to start.".format(AGLOBALPARAM))
        print ("Let's multiply it by 10.")
        multiplyParam(AGLOBALPARAM)
        print ("AGLOBALPARAM is still {}".format(AGLOBALPARAM))
        print ("Let's divide it by 10")
        divideParam()
        print ("Now AGLOBALPARAM is {}".format(AGLOBALPARAM))
        print ("Let's sum it to 10")
        sumParam()
```

A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function (e.g. the parameter names in the function definition behave like local variables).

```
In [ ]: # functions are not evaluated if not called
        def test_scope(max_val):
```

```
        for i in range(max_val):
            print (i)
        print ("max_val in 'test_scope' function is {}".format(max_val))

    # the Python interpreter starts evaluating the code from here
    max_val = 10
    test_scope(5)
    print ("max_val in global scope is {}".format(max_val))
    print (i)
```

1.7 Exercises for next week

1.7.1 Exercise 2.2

Write code which, given the following list

```
input_list = [3, 5, 2, 1, 13, 5, 5, 1, 3, 4]
```

prints out the indices of every occurrence of

```
y = 5
```

1.7.2 Exercise 2.3

Given the following variables

```
S_t = 800.0 # spot price of the underlying<br>
K = 600.0   # strike price<br>
vol = 0.25  # volatility<br>
r = 0.01    # interest rate<br>
ttm = 0.5   # time to maturity, in years<br>
```

write out the Black Scholes formula and save the value of a call in a variable named 'call_price' and the value of a put in a variable named 'put_price'

1.7.3 Exercise 2.4

Given the following dictionary mapping currencies to 2-year zero coupon bond prices, build another dictionary mapping the same currencies to the corresponding annualized interest rates.

```
d = {
'EUR': 0.98,
'CHF': 1.005,
'USD': 0.985,
'GBP': 0.97
}
```

1.7.4 Exercise 2.5

Build again dates as in Exercise 2.1 (i.e. the weekday of your birthdays for the next 120 years) and count how many of your birthdays is a Monday, Tuesday, ... , Sunday until 120 years of age. Print out the result using a dictionary. (expected output somenthing like: `python {6: 10, 0: 10, 2: 9, 3: 10, 4: 10, 5: 10, 1: 9}`)