Introduction to Python - Practical Lesson 1

Matteo Sani
matteosan1@gmail.com

October 8, 2019

1 Introduction to Python - Part 1

In the first two lessons of this course we'll take a quick tour of the Python programming language and see how to write a simple function which would actually be useful in a real-world finance environment.

1.1 What is Python

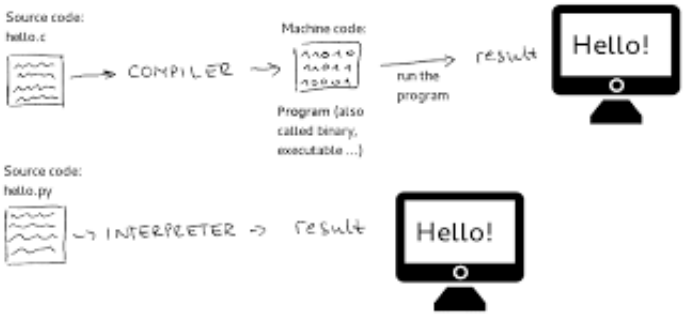
Python is a so called *interpreter*, it takes some code (a sequence of instructions or commands), reads and executes it. This is different from other programming languages like C or C++ which compile code into a language that the computer can understand directly (*machine language*). As a result, Python is essentially an *interactive* programming language, you can program and see the results almost at the same time.

1.2 Which Python should I use ?

Python, as basically all programs, comes in different version and flavours.

The latest version is `3.7` (but it is continously evolving), however you'll see older versions floating around (e.g. `2.7`). This is because there are some big differences between `Python2.X` and `Python3.X` which prevent a sizeable portion of `Python2` users to stick with it (moving to `Python3` would require sizeable amount of work for big projects).

We will go for `Python3.X` anyway !



Interpreted vs compiled language

Any Python interpreter, available at http://www.python.org, comes with a standard set of *packages* (will see in a while what they are), but if you want more functionality, you can download more of them (there are zillions of packages out there).

Some examples are:

- Numpy - which provides matrix algebra functionality;
- Scipy - which provides a whole series of scientific computing functions;
- Pandas - which provides tools for manipulating time series or dataset in general;
- Matplotlib - for plotting graphs;
- Jupyter - for notebooks like this one;
- . . . and many more.

1.3 How can I use Python ?

Once you have downloaded a Python distribution, there are various ways of actually using it.

- The most immediate way is to just execute `python.exe` on the command line to get a Python console for interacting with the interpreter.
- If you are learining Python or do some simple data analysis, Jupyter notebooks (i.e. this document) allow to see the results of your code as you write it, as well as make notes, plot graphs and so on.
- If you are a programmer and want to do more complex things, you'll usually want one or more scripts, perhaps linked together, and execute one of them again using `python.exe`. For this last case an integrated development environment (IDE) can be very useful. An IDE is a graphical user interface which makes writing complex code easier by providing a text editor, a file browser, a debugger (a tool that helps you to spot mistakes in your code) all in one software application. Good example is `PyCharm` (https://www.jetbrains.com/pycharm/) or repl.it an online IDE.

1.4 How we will use it

To avoid time consuming installations we will mainly use online tools so that all you need is just a browser (Firefox :-), Chrome :-), Explorer :-(). For completeness from time to time I will show you code running in in a more advanced IDE called `PyCharm` but you won't need to install it. So our tool are:

- `repl.it`, an online IDE to develop the more complicated projects in these lessions;
- `colab`, an online notebook editor from Google.

As you can imagine there are many more similar tools available which have more or less the same functionality as the one proposed (e.g. cocalc a possible replacement of colab). Another possibility, quite flexible but slightly more complicated to setup, is Anaconda Python. It's free to download and works on Windows, Mac or Linux and has been used in the past years for this course. Those interested can take a look at https://www.anaconda.com.

1.5 Online courses

Python popularity is growing every day so it is very easy to find good (and free) online courses looking in Google. Since in this course we do not have time to cover in depth the potentiality of

this language I strongly suggest you to spend some time in watching one of them. One example could be

MITx: 6.00.1x Introduction to Computer Science and Programming Using Python https://courses.edx.org/courses/course-v1:MITx+6.00.1x+2T2017_2/course/

1.5.1 Let's spend few minutes all together to setup the software

1.6 Python basics

Try the commands I will explain in the remaining part of the lession either typing them in a `colab` notebook or using the interactive shell of `repl.it`.

```
In [1]: # print is a keyword, reserved words that have a special meaning
        print ("Hello world !")

Hello world !
```

```
In [2]: print ("Welcome")
        print ("to")
        print ("everybody")

Welcome
to
everybody
```

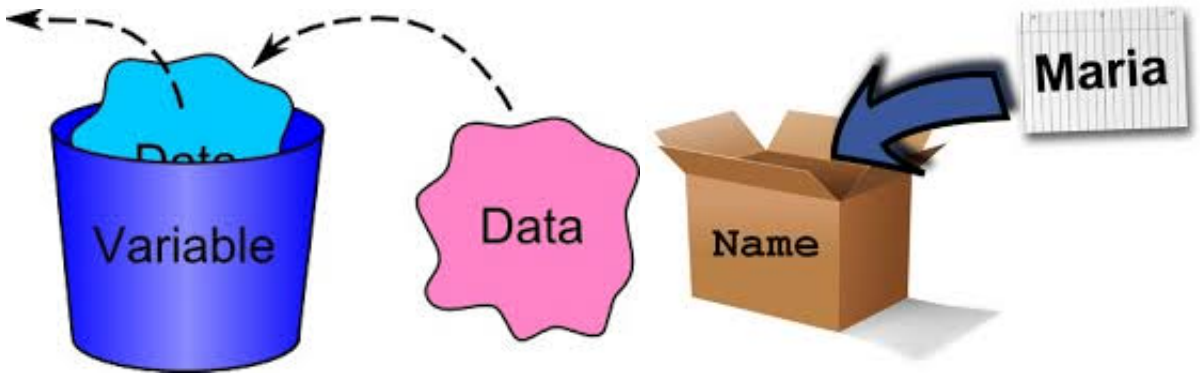
```
In [3]: # this is a comment and the next line prints "Ciao"
        print ("Ciao") # comments like this are useful to explain what's going on in the
                       # code you write

Ciao
```

1.6.1 Exercise 1.1

Write few lines of code in your notebook or in the interactive shell that print on the display "Hello !"

1.7 Variables

Variables are used to store information to be referenced and manipulated in a computer program (e.g. a number, a string...).




```
In [4]: x = 9 # assign number 9 to variable named x

In [5]: x # in console typing just a variable is equivalent to print its value

Out[5]: 9

In [6]: myphone = "Huawei P10Lite"

In [7]: print (myphone)

Huawei P10Lite


In [8]: print (type(x))        # the type keyword tells you which kind of object is
                               # stored in a variable
        print (type(myphone)) # int->integer, str->string we will see later in more
                               # detail what is a string

<class 'int'>
<class 'str'>
```

1.7.1 Python variable name rules

A Python variable name must: * begin with a letter (myphone) or underscore (_myphone); * other characters can be letters, numbers or more _; * variable names are case-sensitive so myphone and myPhone are two distinct variables;

There are some reserved words which you cannot use as a variable name because Python uses them for other things (e.g. `print`, `type`, `for`...**).**

To use GOOD variable names always choose meaningful names instead of short names (i.e. `numberOfCakes` is much better than simply `n`), try to be consistent with your conventions (e.g. choose once and for all between `number_of_cakes` or `numberofcakes` or `numberOfCakes`), usually begin a variable name with underscore (_) only for a special case (will see later when this is usually done).

1.8 Mathematical expressions

```
In [9]: 1 + 2
```

```
Out[9]: 3
```

```
In [10]: 40 - 5
```

```
Out[10]: 35
```

```
In [11]: x * 20 # remember that we set x equal to 9
```

```
Out[11]: 180
```

```
In [12]: x / 4
```

```
Out[12]: 2.25
```

```
In [13]: print (type(2.25)) # this is a new type: floating-point value
```

```
<class 'float'>
```

```
In [14]: x // 4 # interger division - result will be truncated to the
                # corresponding integer (no rounding)
                # 11 / 3 = 3.666666 -> 11 // 3 = 3
```

```
Out[14]: 2
```

```
In [15]: y = 3
         x ** y # x to the power of y
```

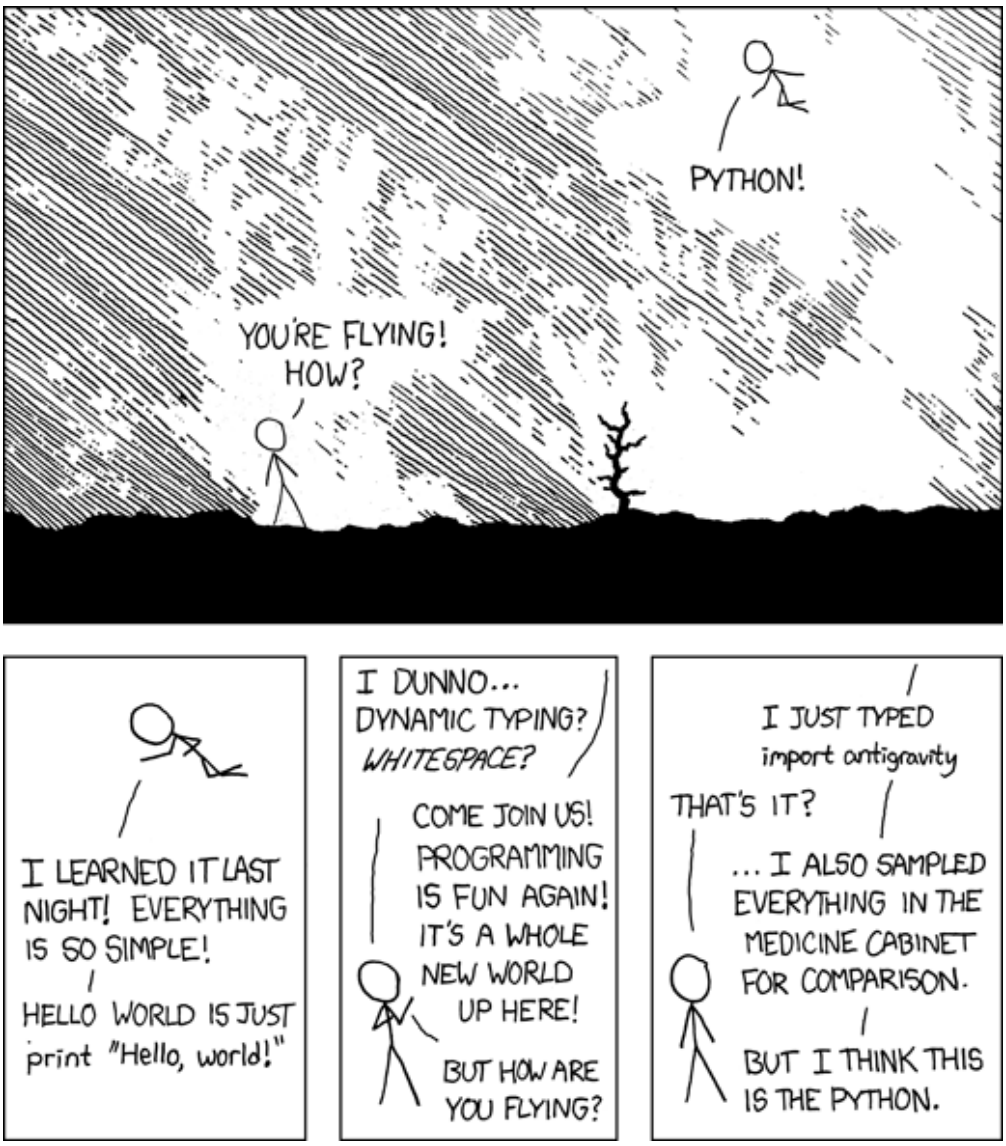
```
Out[15]: 729
```

```
In [16]: 3 * (x + y)
```

```
Out[16]: 36
```

```
In [17]: log(3) # causes an error because the logarithm function
                # is not available by default
```

```
        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)

        <ipython-input-17-ffde4d60496a> in <module>()
    ----> 1 log(3) # causes an error because the logarithm function
          2        # is not available by default


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



Python has many modules for download on the web...

1.9 Modules

Useful functions can be saved in libraries (called *modules*) so that thay can be re-used in different programs. The keyword *import* allows to load functions and data from other Python files (*modules*) and make them available in your program. Python already comes with a lot of built-in modules for doing lots of different tasks (the so called *standard library*) but many more modules are available for download on the web and you can of course write your own !

```
In [18]: import math
         dir(math) # dir keyword lists the content of a module
```

```
Out[18]: ['__doc__',
          '__loader__',
          '__name__',
```

```
'__package__',
'__spec__',
'acos',
'acosh',
'asin',
'asinh',
'atan',
'atan2',
'atanh',
'ceil',
'copysign',
'cos',
'cosh',
'degrees',
'e',
'erf',
'erfc',
'exp',
'expm1',
'fabs',
'factorial',
'floor',
'fmod',
'frexp',
'fsum',
'gamma',
'gcd',
'hypot',
'inf',
'isclose',
'isfinite',
'isinf',
'isnan',
'ldexp',
'lgamma',
'log',
'log10',
'log1p',
'log2',
'modf',
'nan',
'pi',
'pow',
'radians',
'remainder',
'sin',
'sinh',
'sqrt',
```

```
          'tan',
          'tanh',
          'tau',
          'trunc']
```

Once a module has been imported, the functions it contains can be accessed with a *dot* "module_name.function_name":

```
In [19]: math.log(3)

Out[19]: 1.0986122886681098

In [20]: math.exp(3)

Out[20]: 20.085536923187668

In [21]: print (type(math.log)) # yet another type: builtin function
         print (type(math.log(3)))

<class 'builtin_function_or_method'>
<class 'float'>
```

Since we are lazy and we don't want to type "math." every time we compute a logarithm or an exponential, we can use the following syntax:

```
In [22]: from math import log, exp
         print (log(3))
         print (exp(3))

1.0986122886681098
20.085536923187668
```

Putting together what we have learned so far we can try to save the result of an expression in a variable (or even in the same variable we started with). As an example let's compute the interest rate r that produces a return R of about 1051.71 Euro when investing 10000 Euro for 2 years:
$R = Ne^{r\tau} \rightarrow r = \frac{1}{\tau}\log(\frac{R}{N})$

```
In [23]: rate = (1/2)*log(11051.71/10000)
         print (rate)
         x = x + 1    # the value of x was 9 when was used last
         x += 1       # += is a shortcut for x = x + 1
         print (x)

0.05000003706410832
11
```

1.9.1 Exercise 1.2

Write code in your notebook or in the interactive shell to print the natural logarithm of your year of birth (expected something like: 7.587817219993427)

1.10 Boolean expressions

The expressions we have seen so far evaluate to a number. Boolean expressions evaluate to `true` or `false`. Sometimes they involves logical or comparison operators like `or`, `and`, > (greater than), < (less than)... Let's see some example.

```
In [24]: 1 == 2
         # single = assigns a value to a variable like in x = 9
         # double == checks the equality of two objects

Out[24]: False

In [25]: 1 != 2 # != is the "not equal to" operator

Out[25]: True

In [26]: 2 < 2

Out[26]: False

In [27]: 2 <= 2  # in this case we allow the numbers to be equal too

Out[27]: True

In [28]: print (x)
         15 <= x and x <= 20 # this expression could also be written as 15 <= x <= 20

11


Out[28]: False

In [29]: 15 <= x or x <= 20

Out[29]: True

In [30]: not (x > 20) # the not keyword negates the following expression

Out[30]: True
```

1.11 String expressions

A "string" is a sequence of characters (letters, digits, spaces, punctiation, new lines...)

```
In [31]: mystring = "some text with punctuation, spaces and digits 10"

In [32]: "abc" + "def" # it is possible to concatenate strings with +

Out[32]: 'abcdef'

In [33]: "The number " + 4 + " is my favourite number"
         # this causes an error since we are trying to concatenate a string
         # with a number so two different kind of objects


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-33-b9f65c5a45f7> in <module>()
    ----> 1 "The number " + 4 + " is my favourite number"
          2 # this causes an error since we are trying to concatenate a string
          3 # with a number so two different kind of objects


        TypeError: can only concatenate str (not "int") to str
```

To avoid this error is possible to *cast* an object to a different type, Python will try then to convert it to the desired type. This is not always possible though: for example a number can be converted to a string (e.g. from the integer 4 to the actual symbol "4") but the opposite is not possible (e.g. cannot convert the string "matteo" to a meaningful number)

```
In [34]: "The number " + str(4) + " is my favourite number" # str() cast the number to a string

Out[34]: 'The number 4 is my favourite number'

In [35]: print (type(3.4))
         print (type(str(3.4)))

<class 'float'>
<class 'str'>
```

Instead of using + to concatenate strings, Python allows for a prettier formatting using the following syntax (which for example allows for float rounding):

```
In [36]: "The speed of light is about {:.1f} {}".format(299792.458, "km/s")
         # each {} is mapped to the variables listed later in the "format"

Out[36]: 'The speed of light is about 299792.5 km/s'
```

1.12 Indented blocks and the `if/else` statement

Unlike other languages which uses parenthesis to isolate blocks of code Python uses indentation. A first example of this is given by the `if/then` statements where based on some condition we can run different part of the code.

```
In [37]: # remember x value is 13 now
         if x == 1:
             print ("This will not be printed")
             # the block of code that is run if the first condition is met is indented
         elif x == 15:
             print ("This will not be printed either")
             # again the block of code that is run here is indented to be "isolated" by the rest
         else:
             print ("This *will* be printed")
```

```
This *will* be printed
```

```
In [38]: # if by mistake I forget to indent some block I get an error
         if x == 1:
         print ("This will not be printed")
         elif x == 15:
             print ("This will not be printed either")
         else:
             print ("This *will* be printed")
```

```
          File "<ipython-input-38-4535a45a6419>", line 3
        print ("This will not be printed")
            ^
    IndentationError: expected an indented block
```

```
In [39]: if x != 1:
             print ("x does not equal to 1")
```

```
x does not equal to 1
```

As an example, in C++ the previous code would have been:

```
if (x == 1) {
 print ("This will not be printed");
}
else if (x == 15) {
  print ("This will not be printed either");
}
else {
print ("This *will* be printed");
}
```

N.B. Notice how indentation doesn't matter at all here.

1.13 Loops

Another very import feature of a language is the loop that allows to repeat the same block of code many times. In Python loops can be coded with `for` or `while` keywords.

1.13.1 for

```
In [40]: # the range keyword returns a "list" of integers starting from 0
         for i in range(5):
             print (i)
```

```
0
1
2
3
4
```

```
In [41]: for i in range(25, 30): # here both boundaries are specified
             print (i)
```

```
25
26
27
28
29
```

```
In [42]: for i in range (30, 25, -1): # here both boundaries and step are specified
             print (i)
```

```
30
29
28
27
26
```

```
In [43]: for i in range(10):
             if i == 5:
                 continue # this line skips the current iteration,
                          # 5 is actually missing from the output below
             print (i)
```

```
0
1
2
```

```
3
4
6
7
8
9
```

```
In [44]: for i in (4, 6, 10, 20): # here we loop directly on a list of numbers
             print (i)
```

```
4
6
10
20
```

```
In [45]: phrase = 'how to loop over a string'
         for c in phrase:
             print (c)
```

```
h
o
w

t
o

l
o
o
p

o
v
e
r

a

s
t
r
i
n
g
```

1.13.2 while

In a for loop we go through all the elements of a list of objects, the `while` statement instead repeats the same block of code untill a condition is met.

```
In [46]: x = 1
         while x ** 2 < 50: # the following block of code is run if x squared is <50
             print (x)
             x += 1 # each time we increment by 1 the variable x
```

```
1
2
3
4
5
6
7
```

It is possible to exit prematurely from a while loop using the `break` keyword.

```
In [47]: x = 1
         while True: # True is always true :-) so this would run forever
             if (x ** 2 > 50):
                 break # this line exit from the while loop
             print (x)
             x += 1
```

```
1
2
3
4
5
6
7
```

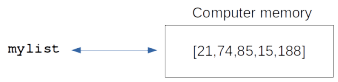
1.14 Lists

A list in Python is a container that is a *mutable*, ordered sequence of elements. Each element or value that is inside of a list is called an item. Each item can be accessed using squared brackets (list indexing is zero-based). It is considered mutable since you can add, remove or update the items in the list. Ordered instead means that items are kept in the same order they have been added to the list.

```
In [48]: mylist = [21, 32, 15]
         mylist
```

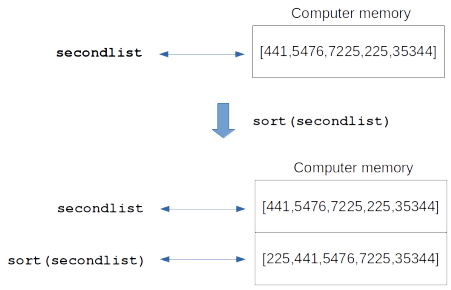
```
Out[48]: [21, 32, 15]
```

```
In [49]: print (type(mylist))

<class 'list'>


In [50]: for i in range(len(mylist)): # len() returns the number of items in a list
             print (mylist[i])

21
32
15


In [51]: len(mylist)

Out[51]: 3

In [52]: mylist[0] # accessing an item by index,remember the first item is element 0

Out[52]: 21

In [53]: mylist[1] = 74 # we can change list items since it's *mutable*
         mylist

Out[53]: [21, 74, 15]

In [54]: mylist[3] # error ! it doesn't exists, the list has only 3
                   # elements, so the last is item 2


         ---------------------------------------------------------------------------

         IndexError                                Traceback (most recent call last)

         <ipython-input-54-3a32eb4a7169> in <module>()
    ----> 1 mylist[3] # error ! it doesn't exists, the list has only 3
           2           # elements, so the last is item 2


         IndexError: list index out of range


In [55]: mylist[-1] # negative index starts from the last element

Out[55]: 15

In [56]: mylist[1:] # : is called slice, all items starting from the second
                    # (remember indexing starts from 0)

Out[56]: [74, 15]
```

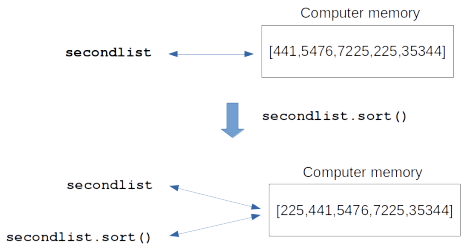


Representation of mylist in the computer memory

```
In [57]: mylist[:2] # elements up to but excluding item 2

Out[57]: [21, 74]

In [58]: mylist.append(188) # append add an item at the end of the list
         mylist

Out[58]: [21, 74, 15, 188]

In [59]: mylist.insert(2, 85) # insert an item in the desired position
                              # (2 in this example)
         mylist

Out[59]: [21, 74, 85, 15, 188]

In [60]: for item in mylist: # iterate over the list
             print (item)

21
74
85
15
188

In [61]: for i, item in enumerate(mylist): # enumerate keyword iterates
                                           # returning the item and its index
             print (i, item)

0 21
1 74
2 85
3 15
4 188

In [62]: secondlist = []
         for item in mylist: # create second list using mylist as input
             secondlist.append(item ** 2)
         secondlist

Out[62]: [441, 5476, 7225, 225, 35344]

In [63]: # for a more compact code the previous lines can be shrinked to
         secondlist = [ item**2 for item in mylist]
         secondlist

Out[63]: [441, 5476, 7225, 225, 35344]
```



Sorted function returns a new list



Sort method instead directly changes the current list

Tricky point here !!!

```
In [64]: print(sorted(secondlist)) # sort returns a new list doesn't
                                   # change secondlist itself
         print(secondlist)
```

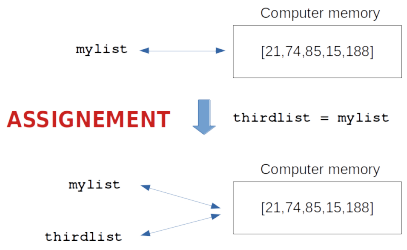
```
[225, 441, 5476, 7225, 35344]
[441, 5476, 7225, 225, 35344]
```

```
In [65]: secondlist.sort() # instead change secondlist
         secondlist
```

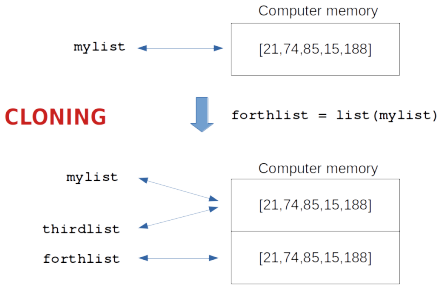
```
Out[65]: [225, 441, 5476, 7225, 35344]
```

```
In [66]: thirdlist = mylist # this is tricky, since variables *point* to objects
         print (mylist)
         print (thirdlist)
         thirdlist[0] = 0
         print (mylist)
         print (thirdlist)
```

```
[21, 74, 85, 15, 188]
[21, 74, 85, 15, 188]
[0, 74, 85, 15, 188]
[0, 74, 85, 15, 188]
```



In thist case mylist is assigned to a new variable thirdlist



In this case mylist has been cloned into fourthlist

```
In [67]: mylist == thirdlist # comparison by value,
                             # i.e. they contains the same items

Out[67]: True

In [68]: mylist is thirdlist # as said before mylist and thirdlist
                             # *point* to the same list

Out[68]: True

In [69]: fourthlist = list(mylist) # now mylist and fourtfhlist *point* to different list
         mylist == fourthlist

Out[69]: True

In [70]: fourthlist = list(mylist) # now mylist and fourtfhlist *point* to different list
         fourthlist is mylist

Out[70]: False
```

List can contain objects of different kind but indices has to be integer

```
In [71]: mixedlist = [1, 2, "b", math.sqrt]
         print (mixedlist)

[1, 2, 'b', <built-in function sqrt>]
```

```
In [72]: print (mixedlist[0])
         print (mixedlist['k'])
```

```
1
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-72-aea4c7f9789e> in <module>()
  1 print (mixedlist[0])
----> 2 print (mixedlist['k'])


TypeError: list indices must be integers or slices, not str
```

1.15 Dictionaries

Dictionaries are objects which map keys to values. Keys can be (almost) any kind of object (strings, numbers...) and they do not have to be sequential.

$$"apple" \rightarrow 4$$

$$"banana" \rightarrow 5$$

Dictionaries are very flexible so that different keys can be different type of objects.

```
In [73]: adict = {"apple": 4, "banana": 5}
         adict["apple"] # values are accessed through their key
```

```
Out[73]: 4
```

```
In [74]: adict["pear"] # error ! this key doesn't exists
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-74-9d051ebd10de> in <module>()
----> 1 adict["pear"] # error ! this key doesn't exists


KeyError: 'pear'
```

```
In [75]: "pear" in adict # indeed
```

```
Out[75]: False

In [76]: len(adict)

Out[76]: 2

In [77]: adict["banana"] = 2
         print (len(adict))
         print (adict)

2
{'apple': 4, 'banana': 2}


In [78]: adict[math.log] = math.exp

In [79]: adict.keys() # returns all the keys (they are not ordered in any way)

Out[79]: dict_keys(['apple', 'banana', <built-in function log>])

In [80]: for key in adict.keys():
             print (key)

apple
banana
<built-in function log>


In [81]: for value in adict.values():
             print (value)

4
2
<built-in function exp>


In [82]: for item in adict.items(): # items() returns a pair of key, value
             print (item)

('apple', 4)
('banana', 2)
(<built-in function log>, <built-in function exp>)


In [83]: seconddict = {"watermelon": 0, "strawberry": 1}
         adict.update(seconddict) # update adict with the items in seconddict
         adict

Out[83]: {'apple': 4,
          'banana': 2,
          <function math.log>: <function math.exp>,
          'watermelon': 0,
          'strawberry': 1}
```

2 Advanced hints

If you would like to know much **more** about python during the course without the EDX site you can look these videos during the course in your spare time:

Some more information about different kind of languages: https://www.youtube.com/watch?v=9oYFH4OmYDY

Basic data types: https://www.youtube.com/watch?v=XIjrEt2lz1U

Variables: https://www.youtube.com/watch?v=z2NLjdfxEyQ

Branching: https://www.youtube.com/watch?v=8vr3nyg5QcM

Tuples: https://www.youtube.com/watch?v=CwZyWaap5Z8

Lists: https://www.youtube.com/watch?v=eMyWO0tcxKg

List Operations: https://www.youtube.com/watch?v=rQBho4-bI3o

Mutation, Aliasing, Cloning: https://www.youtube.com/watch?v=2SRXg8Or-Pc

Functions as Objects: https://www.youtube.com/watch?v=pheM3rVmGMU

Dictionaries: https://www.youtube.com/watch?v=elSt5hke-Rs