train-python-basics

November 18, 2020

1 First steps in programming

- In this part, we will give you some basics of the Python language. This will allow you to understand the structure of Python code so that you will be able to read it more easily. Then we will browse several key concepts to start programming efficiently with Python.

1.1 1. Basics

1.1.1 1.1 Instructions

- An instruction consists in a set of characters defined by the developer (for example you!) in order to perform a specific task. This task can take many forms: print a value, write a condition, define a variable, etc.
- Remember: in the previous section, we have already encountered the following instruction: print("Hello World!"). This instruction ask the system to display (or to print on the screen) the sentence "Hello World!". Running the code print("Hello World!") thus gives the following output.

[113]:
```
print("a")
```

```
a
```

[114]:
```
print("Hello World!")
```

```
Hello World!
```

- Remark: for Python, there is no difference between the instructions print("Hello World!") and print('Hello World!').
- The instruction is read from the left to the right. It starts at the beginning of the line and stops with the line break.
- Remark: you can also use the symbol ; to end an instruction but this is not very useful because this considerably reduces the readability of the code.
- Instructions can also be defined on several rows by means of the symbol . For instance, let us consider the following code.

1.1.2 1.2 Comments

- Comments are very useful in Python. They will allow you to give further details about the instructions.
- A comment appears in the code but is not executed as an instruction when you run the code.
- Comments then only help in improving the understanding (and somehow the readability) of the code.
- The only way to define a line as a comment in Python consists in putting the symbol # at the beginning of this line.
- For instance, you can write something like this

[5]:
```
# The following instruction will display the sentence "Hello World!"
print("Hello World!")
```

```
Hello World!
```

- As you can see, the first line is not executed, it only serves to have a better understanding of what the second line does.

1.1.3 1.3 Key words and reserved words

- In Python, such as for several other programming languages, some key words and reserved words exist.
- Key words are words that will help you to give a structure to your algorithms. These words have a specific meaning and they can't be modified by the developer. In Python there are around 37 words like this. Among others: and, break, class, for, from, if, or, etc.
- Reserved words are words related to common functions, classes or modules. It is recommended not to take these names for any other use, specifically because Python (compared to other programming language) will not prohibit the modification of these words. For instance, print is a reserved word.

1.1.4 1.4 Indentation

- One of the main characteristics of Python is its indentation system. This is one of the main distinction of Python compared to other programming languages (as R for example).
- Concretely, it means that there is no need to delimitate coding blocks using some specific brackets or instructions like begin, end, etc. In Python, basic right shifts (that is, the indentation system) are used as delimiters.
- Let us for example consider a so-called for loop. Such a loop (that will be presented in more details later in this course) consists in a coding block that executes the same instructions several times. Using the indentation system of Python, a for loop will look like this:

[117]:
```
a = 0

for i in range(10):
    #beginning of the instruction to be executed several times
    a = a+i
    #end of the instruction to be executed several times
print(a)
```

```
45
```

- Notice that the instruction a = a+i (that is, the instruction to be looped) is isolated from the rest of the code by means of the indentation system. This is how Python understands that only this particular instruction belongs to the for loop, while the instruction print(a) does not belong to it.

1.1.5 1.5 Symbols and operators

- Symbols can have very different uses in Python.
- Parentheses:
  - Parentheses such as (and) can be used for algebraic computations but also to define tuples, call a function with specific arguments, etc.
  - For instance:

[118]:
```
a = (1+2)/3  # algebraic computation
a
```

[118]:
```
1.0
```

[119]:
```
b = (1,2)  # define a tuple
b
```

[119]:
```
(1, 2)
```

[120]:
```
print(b)  # call a function
```

```
(1, 2)
```

- Square brackets
  - Square brackets such as [and] help to define lists of values.
  - For instance:

[121]:
```
my_list = [1,2,3]  # list of values
print(my_list)
```

```
[1, 2, 3]
```

- Then:
  - When they are coupled with a variable, they can also define a key or / and an index.
  - It's important to know that in Python, the first index of a list is zero, the second index is one, and so on.
  - For instance, one has:

```
[17]: my_list = [1,2,3]     # list of values
      #       0 1 2      # corresponding indices
      print(my_list[0])
      print(my_list[1])
```

```
1
2
```

- Finally:
  - Note that the square brackets can also be used to define a sublist.
  - Given a list named my_list, the instruction my_list[u:v] corresponds to the sublist containing all the indices of the list my_list between the index u and the index v-1 (that is, between the index u included and the index v not included).
  - For instance:

```
[18]: my_list = [1,2,3,4,5]

      print(my_list[0:3]) #from index 0 to index 2
      print(my_list[1:5]) #from index 1 to index 4
```

```
[1, 2, 3]
[2, 3, 4, 5]
```

```
[21]: my_global_list = [my_list, my_list] # a list of list
      my_global_list
```

```
[21]: [[1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
```

- Curly braces:
  - Curly braces such as { and } give information about sets or dictionaries (that will be introduced later in this course).
  - As an illustration:

```
[22]: set = {1,2,3}

      print(set)
```

```
{1, 2, 3}
```

```
[23]: dictionary = {'key1': 'value1' , 'key2': 'value2'}
      print(dictionary)
```

```
{'key1': 'value1', 'key2': 'value2'}
```

```
[24]: dictionary['key1']
```

```
[24]: 'value1'
```

- Operators:

- Operators are characters that have a particular meaning for Python.
- They allow to perform computations, comparisons, assignments, etc.
- Here is a list of such operators:

Operator	Description	Examples
+, -	Addition, Subtraction	10 -3
*, /, %	Multiplication, Division, Modulo	27 % 7
//	Truncation Division (also known as floordivision or floor division)	10 // 3
+x, -x	Unary minus and Unary plus (Algebraic signs)	-3
**	Exponentiation	10 ** 3
or, and, not	Boolean Or, Boolean And, Boolean Not	(a or b) and c
in	Element of	1 in [3, 2, 1]
<, <=, >, >=, !=, ==	Comparison operators	2 <= 3
=, +=, -=, /=, **=	Assignment operators	a += 1
«, »	Shift Operators	6 « 3

- For instance

[122]:
```
print(10-5)    #subtraction
```

```
5
```

[123]:
```
print(12/3)    #division
```

```
4.0
```

[124]:
```
print(10*4)    #product
```

```
40
```

[125]:
```
print(2**3)    #exponentiation
```

```
8
```

1.2 2. Key concepts

1.2.1 2.1 Variable and assignations

- A variable consists in a word (the variable name) that may contain several kinds of information. A variable name has to start with a letter and may contain several letters, numbers or underscores (basically, no space). By convention, variable names will contain only lowercase letters (and no capital letters).

- The declaration of a variable consists in three elements : the variable name, the assignation symbol = and the value of the variable.
- For instance:

```
[29]: example1 = 1
```

- The line of code declares a variable called example1 that contains the value 1.
- It is then possible to use this variable to make some computations. For instance:

```
[30]: print(example1 + 2)
```

```
3
```

- The variable can also be defined through another variable. For instance:

```
[31]: example2 = 1 + 2 * 3

      example3 = example2 + 10

      print(example3)
```

```
17
```

- It is also possible to assign several variables simultaneously. For instance:

```
[32]: a = b = c = 4

      print(b)
```

```
4
```

```
[126]: a, b, c = 1, 2, "Hello"

       print(a)
       print(b)
       print(c)
```

```
1
2
Hello
```

- Note that the first line of the code above is equivalent to:

```
[35]: a = 1

      b = 2

      c = "Hello"
```

- As you can see (and as already mentioned before), variables may contain different kinds of information.
- That refers to the type of the variable.
- For instance, numbers and strings (of characters) are two different types of variable (we will come back to this in the next section).
- This is why you get an error message if you try to run the following line of code:

[37]:
```
print(a + c)
```

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

    TypeError                                 Traceback (most recent call␣
↪last)

    <ipython-input-37-2616b5956feb> in <module>
  ----> 1 print(a + c)


    TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- In that case you have to use a , instead of a +:

[39]:
```
print(a , c)
```

```
1 Hello
```

- It is also possible to concatenate using the standard command + combined with %d which is used for formatting strings and acting as a placeholder for a number. For instance:

[40]:
```
print('%d ' %a + c)
```

```
1 Hello
```

- As you can see, the command %d is replaced by a formatted as a string and then concatenated with c.
- Note that it is not mandatory (and even not possible) to specify the type of a variable before an assignation.
- Moreover the type of a variable may change during a code execution. For instance:

[41]:
```
example4 = 42

# example4 is a number (integer)

example4 = 42 + 0.11
```

```
#example4 has now a different value but it is still a number (float)

example4 = "Hello"

#example4 is now a string!
```

1.2.2 2.2 Data types

- Python has various data types to define variables or more generally to define operations. We can enumerate at least five of them which are:
  - Numbers
  - Strings
  - Lists
  - Tuples
  - Dictionaries

2.2.1 Numbers (float, integer)

- Numbers are basically numerical values you can assign to variables.
- We have already encounter such a type of variable in the previous section.
- For that type of variables, you can apply all the algebraic operators presented earlier. For instance

[72]:
```
var1 = 3
var2 = 6

print(var1 * var2)
print(var2 / 2)
```

```
18
3.0
```

- Python has four numerical types:
  - int (for signed integers);
  - long (for very long integers);
  - float (floating point real values);
  - complex (complex numbers).
- The command type() allows to access to the type of the variable. For instance:

[74]:
```
print(type(12))
print(type(12.1))
print(type(12j))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

1.2.3 2.2.2 Strings

- Strings are set of characters represented between quotation marks.
- Note that + will allow to concatenate strings and * will allow you to repeat strings.
- Moreover, you can access a subset of the string using the following commands: [] and [:].
- For instance:

[76]:
```
s = "Hello World!"

print(s + " - case 1")
print(s[1] + " - case 2")
print(s[6:12] + " - case 3")
print((s+' ')*2 + "- case 4")
```

```
Hello World! - case 1
e - case 2
World! - case 3
Hello World! Hello World! - case 4
```

- Remember that, in Python, the index starts at 0, so that s[1] means the second letter of Hello World! and not the first one.

1.2.4 2.2.3 Lists

- A list is a very useful object that contains an ordered sequence of items. Basically, it's a flexible object that allows to store values of different types.
- These items will be separated by commas and enclosed using brakets []. You will be able to get an access to the listed items using the index or a sequence of indexes, similarly to the case of strings (+ and * are also available).
- For instance:

[78]:
```
l = [ 'abcd', 123 , 4.56, 'hello' ]

print(l)
print(l[0:2])
print((l[3] + " ") * 4)
```

```
['abcd', 123, 4.56, 'hello']
['abcd', 123]
hello hello hello hello
```

- You can easily initialize an empty list or a list containing the same value several times using the operator *:

[79]:
```
# empty list of length 10

l = ['']*10
```

```
print(l)
```

```
['', '', '', '', '', '', '', '', '', '']
```

[80]:
```
# list containing 10 times the value 0:

l = [0]*10

print(l)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

1.2.5 2.2.4 Tuples

- Tuples are very similar to the lists. It's an object that contains an ordered sequence of items. The main differences between a tuple and a list are that:
  - you are not able to change the size of a tuple;
  - you can not update values;
  - you have to use parenthesis such as () to enclose the tuple elements.
- For example:

[81]:
```
t1 = ('abcd', 123 , 4.56, 'hello')
t2 = ('world', 789)

print(t1[0])
print(t1[1:])
print(t1 + t2)
```

```
abcd
(123, 4.56, 'hello')
('abcd', 123, 4.56, 'hello', 'world', 789)
```

1.2.6 2.2.5 Dictionaries

- A dictionary is an unordered collection of "key-value" pairs.
- Dictionaries are usually optimized for retrieving data: you have to know the key in order to retrieve a value.
- Dictionaries are enclosed using curly braces such as {} and elements of a dictionary can be assigned and used thanks to [].
- For instance:

[82]:
```
d = {1:'test1','test2':2}

print(type(d))
print("d[1] = " + d[1]);
print("d['test2'] = " , d['test2']);
print(d.keys())
```

```
d['test3'] = 'a'
print(d)
```

```
<class 'dict'>
d[1] = test1
d['test2'] =  2
dict_keys([1, 'test2'])
{1: 'test1', 'test2': 2, 'test3': 'a'}
```

1.2.7 2.2.6 Conversion

- In some cases, it can be useful to convert data from one type to another. To do so, you just have to use the type of data you want to create as a function. For example:

[83]:
```
x = 7
print(type(x))

y = chr(x)
print(type(y))

z = float(x)
print(type(z))
```

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

1.2.8 2.3 Conditional blocks

- A conditional block is a block that will be executed only if a certain condition is satisfied.
- Such a block is very useful in programming.
- Several types of conditional blocks are introduced below.
- As you will see, the syntax is based on the indentation system.

1.2.9 2.3.1 if instruction

- The key word if will help to define a condition. For instance (Pay attention to the indentation!):

[84]:
```
a = 7
if a <= 10:
    print('a is lower than 10')
```

```
a is lower than 10
```

- All the usual comparison symbols like $<$, $>$, $<=$, $>=$ may be used to formulate the conditions.

- An equality is tested by means of the symbol == and not the symbol = used for assignation. An inequality can be tested using the symbol !=.
- For instance, the instruction a=5 assigns the value 5 to the variable a, while the condition if a==5 tests whether the variable a is indeed equal to 5:

[127]:
```python
a = 5
if a == 5:
    print('a is equal to 5')
```

```
a is equal to 5
```

- Let us now consider another example where the condition is not satisfied:

[86]:
```python
a = 11
if a <= 10:
    print('a is lower than 10')
```

- As you can see, it seems that nothing happens when the condition is not fulfilled.
- This is where the instructions else and elif come in.

1.2.10 2.3.2 else instruction

- The else instruction allows to make the distinction between the case in which the condition is met and another in which it is not. For example (note the indentation!):

[87]:
```python
a = 11
if a <= 10:
    print('a is lower than 10')
else:
    print('a is higher than 10')
```

```
a is higher than 10
```

1.2.11 2.3.3 elif instruction

- The elif instruction will be useful to define another condition if all the other ones have been considered as false. For instance (note the indentation!):

[88]:
```python
a = 7
if a >= 10:
    print('a is larger than 10')
elif a >= 5:
    print('a is larger than 5 and lower than 10')
```

```
a is larger than 5 and lower than 10
```

1.2.12 2.3.4 Incremental conditions

- It is also possible to apply if and else conditions within another condition (which is quite similar to apply an elif instruction). For instance (pay attention to the indentation):

[89]:
```python
a = 3
if a >= 10:
    print('a is larger than 10')
else:
    if a >= 5:
        print('a is larger than 5 and lower than 10')
    else:
        print('a is lower than 5')
```

```
a is lower than 5
```

1.2.13 2.3.5 Complex conditions

- One last thing about Python's conditional blocks is that we can apply multiple comparators. For example:

[90]:
```python
my_age = 19
if 18 < my_age < 24:
    print('I am between 18 and 24 years old.')
```

```
I am between 18 and 24 years old.
```

- It is of course possible to define more complex cases such as:

[91]:
```python
age_max = 110
if 18 < my_age < age_max < 24 :
    print('The condition is true')
else :
    print('The condition is false')
```

```
The condition is false
```

1.2.14 2.4 Iterative blocks

- An iterative block can be defined using numerous different techniques.
- We start with a for loop already met earlier in this course.

1.2.15 2.4.1 for instruction

- The for instruction allows to iterate some instructions according to a predefined iteration sequence. The keyword in allows to define the iteration sequence. For example

[93]:
```python
for a in (2, 3, 5, 7):
    print('%d is a prime number' % a)
```

```
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
```

- Recall: %d is used for formatting strings and acting as a placeholder for a number.

[94]:
```
a=0

for i in range(10):
    a+=i

print(a)
```

```
45
```

- Note that the instruction a+=i is a shortcut often used in Python and is equivalent to the instruction a=a+i.
- Note also that range() is a built-in function of Python very useful in case of loops. range(n) corresponds to the sequence of integers between 0 and n-1. For instance:

[95]:
```
n=5

# sequence of integers between 0 and 4:
x=range(5)

print(x[0])
print(x[1])
print(x[2])
print(x[3])
print(x[4])
```

```
0
1
2
3
4
```

- Of course, it is possible to display the elements of range(n) (or any other sequence) using a for loop! For instance:

[96]:
```
n=5
x=range(5)

for i in range(5):
    print(x[i])
```

```
0
1
```

```
2
3
4
```

- More generally, the instruction range(a,b,c) corresponds to the sequence of integers between a and b-1 in steps of c. The parameter c is optional and set to 1 by default. For example:

[97]:
```python
a,b = -1,2

# sequence of integers between -1 and 1:
x=range(-1,2)

print(x[0])
print(x[1])
print(x[2])
```

```
-1
0
1
```

- Note that the length (that is, the number of elements) of range(-1,2) is equal to 3. It is then possible to display the elements of range(-1,2) by means of a for loop like this:

[98]:
```python
a,b = -1,2

# sequence of integers between -1 and 1:
x=range(-1,2)

for i in range(3):
    print(x[i])
```

```
-1
0
1
```

- Another example:

[99]:
```python
a,b,c = -2,5,2

for i in range(a,b,c):
    print(i)
```

```
-2
0
2
4
```

1.2.16 2.4.2 while instruction

- Another way to compute an iterative block is to apply a while instruction. It allows to compute an instruction while a certain condition is considered as true. For instance:

[100]:
```
a = 1
while a < 5:
    print(a)
    a+=1
```

```
1
2
3
4
```

- Remark: pay attention to increment the variable used for the while instruction (the variable a in the exemple above) in order to avoid infinite loop! Indeed, if you forget the instruction a+=1in the example above, the value of a is never modified and thus the condition a<5 is always satisfied. Note that in a for loop such a problem doesn't occur thanks to the predefined iteration sequence which serves to modify the value of the variable.

1.2.17 2.4.3 Differences

- for is very different from while considering the following facts:
  - from a conceptual perspective: the end of the for loop is anticipated and you will know in advance the number of iterations;
  - from a practical perspective: a for loop allows to iterate on values and to apply computations on it whereas while allows to repeat an algorithm while a condition is true.

1.2.18 2.4.4 break instruction

- The instruction break allows to stop the iteration immediately or to reduce the number of iterations according to certain circumstances. For instance:

[101]:
```
c = 0
for b in range(4,12):
    print('b is equal to %d' % b)
    c += 2
    print('c is equal to %d' % c)
    if b < c:
        break
```

```
b is equal to 4
c is equal to 2
b is equal to 5
c is equal to 4
b is equal to 6
c is equal to 6
b is equal to 7
c is equal to 8
```

- As you can see, the for loop is not executed all the way through (that is, the range(4,12) is not entirely browsed) due to the break instruction.

1.2.19 2.4.5 continue instruction

- The instruction continue will allow to skip an iteration in order to go directly to the next one. For instance:

[102]:
```
for a in range(-5,5):
    if a < 0:
        continue
    print(a)
```

```
0
1
2
3
4
```

- As you can see, the instruction print(a) is skipped if the value of the variable a is lower than 0.

1.2.20 2.5 Functions

- A function consists in a block of code devoted to execute a particular task.
- Actually, we have already encounter several functions earlier in this course such as the function print allowing the developer to display something on the screen.
- Any function can be called by the developer by means of its name.
- A very interresting and powerful fact is that, as a developer, you are allowed to create your own functions!
- It can be performed by using the keyword def: this is the signal for Python that you start to create your own function.
- Let us consider a first example. Suppose that you want to create a function called "hello_user" saying "Hello user!".
- You can do that by defining your function like this:

[128]:
```
def hello_user():
    print("Hello user!")
```

- Pay attention to the structure of the code:
  - the definition of the function starts with the keyword def ;
  - there are opened and closed parentheses followed by a colon right after the name of the function. The parenthesis can contain arguments of the function (i.e. elements passed to the function and that can be used inside the function). When there is no argument, we have nevertheless to use empty parenthesis ;

  - the block of code to be executed once the function is called is defined by means of the indentation system.

- When the code above is executed, it seems that nothing happens.
- Indeed, the code above creates the function hello_user but in order to use this function, you have to call it.
- Calling a function is very simple, it suffices to type its name followed by opened and closed parentheses

[129]:
```
hello_user()
```

```
Hello user!
```

- Of course, the function hello_user is not very useful, in particular because it doesn't depend on any variable.
- Let us then create (as a second example) another function called "square" that will compute the square of a number of your choice.
- You can do that by defining your function like this:

[130]:
```
def square(n):
    n*n
```

- Pay attention to the structure of the code: it is very similar to the funtion hello_user but here the function square depends on a parameter (the argument) called n.
- It is important to know that the choice for the name of the parameter is free, it only serves to define what the function is supposed to do with this parameter.
- Let us now try our brand new function by calling it:

[131]:
```
square(3)
```

- As you can see, the function square is applied to the number 3 which is supposed to give 9… but nothing happens!
- Let us try this way:

[107]:
```
print(square(3))
```

```
None
```

- That doesn't work either… What's the problem?
- The problem comes from the fact that we want to get an information back of the function: we want to access to the answer computed by the function.
- This is the purpose of the keyword return.
- When some computations are performed within the function block, you must specify the quantity to which you want to have access as a result of the function.
- Let us then modify the definition of our function square as follows:

```
[132]: def square(n):
           return(n*n)

       square(3)
```

```
[132]: 9
```

- So it works!
- It is worth mentioning that the keyword return is considered by Python as the last instruction of the function. It means that if there is another instruction after the line starting with the keyword return, Python will skip it. For instance:

```
[133]: def twice(n):
           return(2*n)
           print("It works!")

       twice(3)
```

```
[133]: 6
```

- Note that you can then use the output of your functions and assign it to continue your computations. For instance:

```
[110]: a = square(4)
       b = a + 3
       print(b)
```

```
19
```

- Of course, you are not limited to functions with only one variable.
- You can create a multivariables function by simply adding other variables separated with commas between the parenthesis. For instance:

```
[136]: # function that will compute the area of a rectangle

       def rect_area(a,b = 1):
           return(a*b)

       # applying the function to a rectangle of sides 2 and 3
       rect_area(2)
```

```
[136]: 2
```

```
[ ]:
```

- You can also combine functions with conditions and/or loops!
- For instance, let us define a function computing the absolute value of a number.

```
[112]: # definition of the function

       def abs_val(x):
           if x >= 0:
               return(x)
           else:
               return(-x)

       # test of the function for the value -3

       abs_val(-3)
```

```
[112]: 3
```

1.3 3. Libraries manipulations examples

1.3.1 3.1 First examples

- Below a first introductive example to manage both library import, allias, and call of functions

```
[ ]:
```

```
[140]: df = pd.DataFrame([1,2,3])
```

```
[141]: df
```

```
[141]:    0
       0  1
       1  2
       2  3
```

```
[ ]:
```

```
[70]: import pandas as pd
      import numpy as np
```

```
[143]: from pandas import pivot_table as pd_pivot_table
```

```
[144]: df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
                                "bar", "bar", "bar", "bar"],
                          "B": ["one", "one", "one", "two", "two",
                                "one", "one", "two", "two"],
                          "C": ["small", "large", "large", "small",
                                "small", "large", "small", "small",
                                "large"],
                          "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
                          "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
       df
```

[144]:
```
     A    B      C  D  E
0  foo  one  small  1  2
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  5
4  foo  two  small  3  6
5  bar  one  large  4  6
6  bar  one  small  5  8
7  bar  two  small  6  9
8  bar  two  large  7  9
```

[71]:
```
table =  pd_pivot_table(df, values='D', index=['A', 'B'],
                     columns=['C'], aggfunc=np.sum)
table
```

[71]:
```
C        large  small
A   B
bar one    4.0    5.0
    two    7.0    6.0
foo one    4.0    1.0
    two    NaN    6.0
```

1.4 4. Exercises

See other files

[ ]:
```
```