1 ITFP 2.1 Introduction to Python

Keep under you pillow the Python Library Reference https://docs.python.org/3/library/index.html

1.1 The interpreter

Each cell of this graphical interface called Jupyter notebook is equivalent to a block of python instructions written inside the Python interpreter. No differences! This is only a beautiful way to gather code, text, output and documentation.

1.2 Variables as labels

In Python, a variable is created assigning a value (a number, a string, a dictionary, a list, an object..) to a label with the = symbol.

- a is the label attached to the object (the number 3).
- b is attached to a string object
- c is a list

In [1]: a = 3
   b = "ITFP"
   c = [2,3,4]

Let’s inspect the variable using the Python global function print()

In [2]: print(a)
   print(b)
   print(c)

3
ITFP
[2, 3, 4]

First point: We don’t have to tell Python the type of the variable a, but it is inferred automatically by the interpreter.

Python is a strongly typed object oriented language: all the variables are objects of a specific type.

You can inspect the type of a variable with the global function type().
In [3]: print( type(a) )
   print( type(b) )
   print( type(c) )
<class 'int'>
<class 'str'>
<class 'list'>

In [4]: # We cannot mix different type of objects of course.---> this is a comment
   a + b

---------------------------------------------------------------------------
TypeError                      Traceback (most recent call last)
<ipython-input-4-7815e8693b57> in <module>
     1 # We cannot mix different type of objects of course.---> this is a comment
     2 a + b
----> 2 a + b

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

This is a Python exception: these are useful because usually are really descriptive and they point you to the exact point where your code has a problem. We will look at them again, but no more segmentation faults

1.3 Numbers
Python uses two main number types: int for integers (like C++) and float for floating point numbers corresponding to double precision type of C++

In [5]: a = 2
   b = 5
   a + b
Out[5]: 7
In [6]: a * b
Out[6]: 10

The power is directly available!
In [7]: a ** 2
Out[7]: 4

N.B.: in Python 3 the division between two integers returns a float. The behaviout in Python 2 is different! Be careful!
In [8]: a / b
Out[8]: 0.4
1.4 Math functions

A lot of modules are available in Python, containing functions and classes. As in other languages, they have to be explicitly included in the current script or Python session.

The keyword to include modules is import. Let’s do an example using the math module.

```
In [9]: import math
   : math.sqrt(a)
Out[9]: 1.4142135623730951
In [10]: sqrt(a)
```

```
NameError Traceback (most recent call last)
<ipython-input-10-55c08d4e5fa4> in <module>
----> 1 sqrt(a)
NameError: name 'sqrt' is not defined
```

```
In [11]: from math import sqrt
   : sqrt(a)
Out[11]: 1.4142135623730951
```

You can also import all the labels available inside a module with the __from module import __ syntax.

```
In [12]: from math import *
   : pi  # è una variabile dentro math
Out[12]: 3.141592653589793
```

1.5 The help() command and Python global commands

In Python language there are a lot og global function that can be used without any module import. They are basic and fundamental features of the language. For a complete list look here (https://docs.python.org/3/library/functions.html)

For example we have already met the type and print functions. Another useful one if the help() function which returns the documentation for every module or function in the language.

```
In [13]: help(math.sin)
Help on built-in function sin in module math:

    sin(x, /)  
    Return the sine of x (measured in radians).
```
1.6 The dir() function

Talking about ways to discover features and functions available in the Python language, you can use the `dir` command to output the list of available labels of an object (what's inside that)

```
In [ ]: dir(math)
```

2 Strings

A string is a portion of text, a list of characters. Python has a really flexible library for the use of strings, a lot more useful for text manipulation than other languages.

```
In [15]: s = "HI! I'm a ITFP student"  #double " 
   t = 'A lot of fun'  #single ' 
   u = """A lot of fun!  
   """  #three ' to start a multiline string

In [16]: print(u)
A lot of fun!

Let's check some quick string manipulations...

In [17]: s + "! " + t
Out[17]: "HI! I'm a ITFP student! A lot of fun"

In [18]: s.upper()
Out[18]: "HI! I'M A ITFP STUDENT"

In [19]: t.split(" ")
Out[19]: ["A", 'lot', 'of', 'fun']

For example we can use the count method of string objects to count the occurrences of a substring

```
In [20]: s = "Di te che spendi stipendi stipati in posti stupendi"
   print(s.count("s"))
   print(s.count("st"))
   print(s.count("sti"))
```
There are also *strange* functions like `istitle()`. Guess what it does..

**In [21]:**
```
s1 = "It Tools For Physicists"
s2 = "IT Tools For Physicists"
print(s1, "is title?", s1.istitle())
print(s2, "is title?", s2.istitle())
```

```
It Tools For Physicists is title? True
IT Tools For Physicists is title? False
```

Notice that here we are using the `print` function, using more than one arguments. All the arguments are printed on the same line with a space. Different print statements output on a new line.

You can also change the separator of `print` function

**In [22]:** `help(print)`

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='
', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

**In [23]:** `print("IT", "Tools", "For", "Physicists", sep="-")`

```
IT-Tools-For-Physicists
```

We will return later on the *named* argument that we have used here.

### 2.1 String format

We you want to print numbers or other *printable* objects inside a string, the `format()` command is really handy: it converts automatically the object to string.
In [24]: a = 3.4
    print("I have " + str(a) + " apples")
    print("I have {} apples".format(a))

I have 3.4 apples
I have 3.4 apples

You can also add formatting options

In [25]: help(str.format)

Help on method_descriptor:

format(...)  
S.format(*args, **kwargs) -> str

    Return a formatted version of S, using substitutions from args and kwargs.  
    The substitutions are identified by braces ('{'} and '}'.

In [26]: print("I have {:.5f} apples".format(math.pi))

I have 3.14159 apples

3  Lists

The list is the most used data type in Python, they are ubiquitous. The can contain arbitrary objects inside them. They are really a generic container, and they can contain nested structures.

In [27]: l = [1, 2, "ciao"]
    m = [1, "ciao", 2, [2,3,4], l ]
    print(m)  # the print function works of course

[1, 'ciao', 2, [2, 3, 4], [1, 2, 'ciao']]

You can access list elements in the usual indexing manner, but there are also more sofisticated ways in Python.

In [28]: l[0]

Out[28]: 1

For example you can access to elements counting from the last one.

In [29]: m[-3]

Out[29]: 2
3.1 Slicing

If you want to extract sublists you have to use slicing

In [30]: s = "abcdefghil"
   l = list(s)  # A string is a list of characters -- more later
   print(l)

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l']

Slicing works using more indexes separated by a semicolon: Elements are extracted from the first index to the second one (excluded).

In [31]: l[0:5]
Out[31]: ['a', 'b', 'c', 'd', 'e']

In [34]: l[1:4]
Out[34]: ['b', 'c', 'd']

In [33]: l[0:6:3]
Out[33]: ['a', 'd']

You can also omit one of the two numbers in the slice: in this case it corresponds to 0.

In [35]: l[:4]
Out[35]: ['a', 'b', 'c', 'd']

In [38]: l[-3:]
   # or "until the end of the string"
Out[38]: ['h', 'i', 'l']

The slice is a brand new list, you can assign to it a label and use it.

In [39]: m = l[1:4]
   print(m)
   print(m[0])

['b', 'c', 'd']
b
3.2 Generate list of numbers

A global command exists to generate list of integer numbers: \texttt{range()}. It is a special function, called \textit{generator} (more in the next tutorial): it generated a list of number one at a time, so if we want a complete list in one go we have to explicitly convert it to list.

\begin{verbatim}
In [40]: list(range(10))
Out[40]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [41]: a = list(range(10))
a
Out[41]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [42]: b = list(range(4,10))
print(b)
[4, 5, 6, 7, 8, 9]
In [43]: c = list(range(0, 100, 10))  # every 10 elements
print(c)
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
In [44]: d = list(range(100, 0, -10))  # decreasing
print(d)
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
\end{verbatim}

4 Manipulate a list

Lists are modifiable objects: elements can be added, removed, modified.

\begin{verbatim}
In [45]: a = [ 1, 2, "ciao" ]
In [46]: a[1] += 1  # modify an element
   print(a)
[1, 3, 'ciao']
In [47]: a[1] = 4  # replace an element
   print(a)
[1, 4, 'ciao']
\end{verbatim}
In [48]: a.append("new element")  # Add a new element at the end of the list
   print(a)
[1, 4, 'ciao', 'new element']

In [49]: a.insert(1, "2ř position") # Add a new element in the second position
   print(a)
[1, '2ř position', 4, 'ciao', 'new element']

I can remove an element knowing it's value using the function \texttt{remove}

In [50]: help(list.remove)
Help on method_descriptor:
remove(self, value, /)
   Remove first occurrence of value.
   Raises ValueError if the value is not present.

In [51]: a.remove("ciao")
   print(a)
[1, '2ř position', 4, 'new element']

I can also remove an element using the position index:

In [52]: help(list.pop)
Help on method_descriptor:
pop(self, index=-1, /)
   Remove and return item at index (default last).
   Raises IndexError if list is empty or index is out of range.

In [53]: e = a.pop(2)  # Remove the third element
   print(e)
   print(a)
4
[1, '2ř position', 'new element']
For cycle

The for statement iterates a block of code for a certain number of times. In Python a block of code is not delimited by {} parenthesis as in C++, but is simply indented by 4 spaces (please avoid tabs)

```python
In [55]: for i in range(5):
        j = i + 2
        print(j)
```

```
2
3
4
5
6
```

You exit from the repeated block of code going back of 4 spaces. As strange as simple.

```python
In [56]: s = "I prefer C"
    for i in range(10):
        s += "+
        s+= " than Python"

    print(s)
```

```
I prefer C++++++++++ than Python
```

The command range(10) is used to generate a list of 10 elements on which the variable i is iterated. In general in Python you can always iterate with a for cycle on a list.

```python
In [14]: l = ["Fisica1", "Lab1", "Lab2", "Fisica2"]
    for i in l:
        print("I love: "+format(i))
```

```
I love: Fisica1
I love: Lab1
I love: Lab2
I love: Fisica2
```
5.1 Example: Fibonacci sequence

In [57]: n = 50
sequence = [0,1]
   for i in range(2,n): # This is going to be a problem if we ever set n <= 2!
       sequence.append(sequence[i-1]+sequence[i-2])
   print(sequence)

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, ... 102334155, 165580141, 267914296, 433494437, 701408733, 1134903170, 1836311903, 2971215073, 4807526976, 7778742049]

6 More on strings and lists

The global command len() returns the "length" of an object in general. Since strings are lists of characters we can use this method also on strings

In [81]: help(len)

Help on built-in function len in module builtins:

len(obj, /)
   Return the number of items in a container.

In [82]: a = [2,3,4,5,65]
   len(a)

Out[82]: 5

In [83]: a = "ciao ciao"
   len(a)

Out[83]: 9

   In fact you can consider a string as list in different contexts. For example you can do a for loop on a string or use the slicing.

In [84]: for i in a:
   print(i)

c
i
a
o

c
i
a
o
In [85]: print(a)
    a[1:7:1]

ciao ciao

Out[85]: 'iao ci'

6.1 Join list of strings
The elements of a list can be joined together with a string using the join method of string objects

In [89]: help(str.join)

Help on method_descriptor:

join(self, iterable, /)
    Concatenate any number of strings.
    The string whose method is called is inserted in between each given string.
    The result is returned as a new string.
    Example: '.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

In [86]: b = ['ciao', 'come', 'va', '?']
    b

Out[86]: ['ciao', 'come', 'va', '?']

In [87]: '-'.join(b)

Out[87]: 'ciao-come -va-?'

7 If - else
The if - else construct permits to execute a block of code only if the specifies condition is True.
    First of all we have to introduce bool objets: they can have only two values: True or False.

In [58]: a = True
    b = False

In [59]: a

Out[59]: True

In [60]: type(a)

Out[60]: bool
In Python all the usual comparison operators, ==, >=, >, <, <= are available

```
In [61]: f = 3
   print(f == 2)  # Comparison operators
False
```

```
In [62]: if f == 2:
   print("Eh si ")
else:
   print("Oh no!")
Oh no!
```

```
In [63]: f = "CIIiao"
   
   if f ==2:
      print("SI")
   elif f=="ciao":
      print("è una stringa")
   else:
      print("BOH")
BOH
```

We can group different logic statements with keyword **and**, **or** instead of && || of C++. Logical negation can be expressed both with the keyword **not** for clarity, or with ! symbol

```
In [64]: a = 3
   b = "ciao"
   if (f == 2 and a !=1 ) or len(b) == 4:
      print("OK")
OK
```

### 8 Dictionaries

A dictionary is a data structure based on a map of key:value. Usually the key is a string, but you can use whatever object you want. A dictionary is defined using {} parentheses.

```
In [65]: di = { 3: "ciao" }
In [66]: a = {
   "a" : 2,
   "b": [2,3,4,5],
   "c": { "a": [2,3,4],
          "e": "ciao"}
   }
```
8.1 Dictionaries manipulation

In [67]: print(a["c"])

{'a': [2, 3, 4], 'e': 'ciao'}

In [68]: a["c"]['a'][2] = 3333

In [69]: d[1]

Out[69]: 90

From nested dictionaries you can get keys one after another.

In [70]: a["c"]['a'][1]

Out[70]: 3

9 Example: Lab2 measurements

In [77]: lab2 = {
        "V": [10, 100, 1000],
        "I": [0.1, 0.3, 0.4]
    }

Let’s calculate the resistance

\[ R = \frac{V}{I} \]

N.B.: you can use LaTex in jupyter notebooks

In [78]: R = []

    for i in range(len(lab2["V"])):
        R.append(lab2["V"][i] / lab2["I"][i])
    lab2["R1"] = R

In [79]: lab2

Out[79]: {'V': [10, 100, 1000],
          'I': [0.1, 0.3, 0.4],
          'R1': [100.0, 333.33333333333337, 2500.0]}