Lecture#1

May 2, 2023

1 Python for Signal Processing

Danilo Greco, PhD - danilo.greco@uniparthenope.it - University of Naples Parthenope

1.0.1 Lecture 1

This lecture will provide an overview on: 1. Python installation on Linux, Mac OS and Windows 2. introduction on Jupyter Notebooks and Google Colaboratory for interactive rapid prototyping of python applications 2. basic programming concepts 3. main language statements 4. exceptions, functions, arguments and parameters 5. strings, lists, tuples and dictionaries

2 Python installation

The Python language can be easily installed both in Windows, Mac OS and in Linux.

It can be downloaded from the official web site and installed by following these instructions, depending on your target platform.

Official documentation for the latest version is available on this web site.

Additional libraries can be used to expand the native features of the language; these software modules can be installed by running:

```
pip install
```

or

```
python -m pip install
```

followed by the required module name.

Beginners are often tempted to install Anaconda, which is definitely easier to setup but generates a lot of troubles for unsatisfied dependencies when updating or installing packages (therefore it is not recommended).

Eventually, if you're familiar with Docker, you can download a ready made python image and run the interpreter inside a container by typing the following command:

```
docker pull python
```

3 What is Jupyter Notebook?

The Jupyter Notebook is an open source web application that you can use to create and share documents that contain - live code, - equations, - visualizations, - text.

As a client-server application, the Jupyter Notebook allows you to edit and run your notebooks via a web browser, even without internet access.

There are two components: the Python kernel and the dashboard. These lessons are delivered through Jupyter Notebooks.

Additional kernels are available for other programming languages which can be installed following these instructions.

4 Install

It can be installed by executing the following command:

pip install jupyter

or

python -m pip install jupyter

Easier installation can be done through Anaconda, a popular data science platform that can be downloaded here: by installing Anaconda the tool is already available along with many other preinstalled scientific libraries.

It can be started by running the following command in a terminal shell:

jupyter notebook

This will start up Jupyter and your default browser should start (or open a new tab) to the following URL:

http://localhost:8888/tree

5 Basic usage

Each notebook can be organized in independent cells that can be of two main types:

5.0.1 markdown cells

containing text, html code, LaTeX mathematical formulas

HTML code can be used to format text or include images such as: >

which produces:

Mathematical expressions can be written inline with other text, like $e^{i\pi} + 1 = 0$, with >$e^{i\pi} + 1 = 0$

or on a single line:

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!} x^i$$

by typing

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!} x^i$$

5.0.2 code cells

containing python code to be executed by pressing Shift+Enter or the Run button on the toolbar.

This is an example of code cell:

[1]:
```
import time
for i in range(8):
    print(i)
    time.sleep(0.5)
```

```
0
1
2
3
4
5
6
7
```

Organizing the code in cells is useful to control the execution and selectively execute different cells depending on the specific task.

6 Useful features

If you don't remember the usage of a function, it is possible to recall the related python help by using the ? symbol. For instance:

[2]:
```
?range
```

or, after importing a particular library:

[4]:
```
import numpy as np

?np.sqrt
```

Another command, useful for code optimization, is %time which reports the execution time of a line of code or %%time for the entire cell.

[5]:
```
%time for i in range(5): print('Hello world!')
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
CPU times: total: 0 ns
Wall time: 1.13 ms
```

[6]:
```
%%time
for i in range(5):
    print('Hello world!')
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
CPU times: total: 0 ns
Wall time: 1.02 ms
```

To test multiple runs of a piece of code, we can use the %%timeit magic command, as of the example below:

[7]:
```
%%timeit -n 100
print('Hello world!')
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
4.76 µs ± 1.08 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[8]:
```
%%timeit -n 100
msg = 'Hello world!'
print(msg)
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
5.2 µs ± 1.79 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

It is also possible to execute shell commands by prepending the exclamation mark (!):

[9]:
```
!pip install numpy
```

```
Requirement already satisfied: numpy in
c:\users\grcdnl71d14d969b\anaconda3\lib\site-packages (1.23.5)
```

[11]:
```
#!ls -la
!dir
```

```
 Il volume nell'unit… C Š Windows
 Numero di serie del volume: ACAF-A8C2
```

```
 Directory di C:\Users\GRCDNL71D14D969B\Desktop\Intelligent signal
processing\OneDrive_1_10-4-2023\Python Course

02/05/2023  17:01    <DIR>          .
10/04/2023  09:24    <DIR>          ..
12/04/2023  17:46    <DIR>          .ipynb_checkpoints
10/04/2023  09:24            96.184 anatomy.png
12/04/2023  10:34            57.989 contours.svg
10/04/2023  09:24         4.075.555 covid19_cases.csv
10/04/2023  09:24            47.282 Deep_Learning-Colab.ipynb
10/04/2023  09:24         2.772.143 diamonds.csv
10/04/2023  22:40             1.228 Homework#2.ipynb
10/04/2023  09:24             1.913 Homework#3.ipynb
10/04/2023  09:24             1.029 Homework#4.ipynb
10/04/2023  09:24               946 Homework#5.ipynb
10/04/2023  22:43             9.353 HW_Solution#2.ipynb
10/04/2023  09:24             7.433 HW_Solution#3.ipynb
10/04/2023  09:24           123.257 HW_Solution#4.ipynb
10/04/2023  09:24           150.824 HW_Solution#5.ipynb
10/04/2023  09:24             4.549 iris.csv
10/04/2023  09:24             3.639 iris_test.csv
10/04/2023  09:24               938 iris_train.csv
10/04/2023  09:24           153.379 KNN.ipynb
02/05/2023  17:01            87.432 Lecture#1.ipynb
12/04/2023  19:06         4.216.375 Lecture#2.ipynb
12/04/2023  17:04            74.893 Lecture#3.ipynb
12/04/2023  10:31           374.542 Lecture#4.ipynb
12/04/2023  14:43         1.884.221 Lecture#5.ipynb
02/05/2023  12:59           520.375 Lecture#6(1).ipynb
12/04/2023  21:39           438.309 Lecture#6.ipynb
10/04/2023  09:24            46.954 lena.jpg
10/04/2023  09:24               450 missing.csv
              26 File     15.151.192 byte
               3 Directory  725.857.447.936 byte disponibili
```

7 Alternatives to local installation

It is possible to use jupyter notebooks even without installing anything on your computer. Best available options on internet are:

- Google Colaboratory (recommended alternative, requires a Google account): provides a customized version of jupyter notebooks which can be executed either on a standard CPU, for low computational requirements, or on GPUs or on TPUs when parallel processing is required. Provides a good quickstart tutorial and can mount Google Drive storage for data persistence.
- Microsoft Azure Notebooks (likely to be discontinued): notebooks can be run on a VM or a shared cluster computing environment

8 Language introduction

Python is an interpreted programming language but additional tools allow script compilation to obtain native executables for the target platform.

As every programming language, it has a list of reserved keywords that cannot be used as variable names or identifiers.

The keyword list is accessible through the inline help:

[12]:
```
help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               break               for                 not
None                class               from                or
True                continue            global              pass
__peg_parser__      def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
```

9 Constants, variables and expressions

Unmutable values such as numbers, letters or strings are termed Constants because their value is not meant to change over time.

[13]:
```
print(145)
```

```
145
```

[14]:
```
print(23.6)
```

```
23.6
```

[15]:
```
print('Hello students!')
print("Hello students!")
```

```
Hello students!
Hello students!
```

Please note that string constants are enclosed in single or double quotes.

Variables are named memory locations where one can store data values and later retrieve them using the variable name.

Examples of variables are:

[16]:
```
a = 145
print(a)
```

```
145
```

[17]:
```
b = 23.6
print(b)
```

```
23.6
```

[18]:
```
c = 'Hello students!'
print(c)
```

```
Hello students!
```

Variables can be modified any time by simply setting a new value which overwrites the content of the memory location:

[19]:
```
a = 100  # new value for variable a
print(a)
```

```
100
```

Variable names are chosen by programmers and usually abide by some naming rules defined in companies guidelines or best practices.

According to language naming rules, variables: * must begin with letter or _ (underscore) * must contain only letters, numbers and underscores * are case sensitive

Examples >name, name23, _name are valid names;

$name, 23name, na.me are invalid names;

name, Name, NAME are different names.

Variable names should be able to help us remind what is going to be stored in that memory area. As an example, the first code block does not manifest its purpose, whereas the second one makes it clear to everyone even if the results are exactly the same.

[20]:
```
# bad naming
a = 38
b = 13.50
c = a * b
print(c)

# good naming
worked_hours = 38
hourly_rate = 13.50
amount_due = worked_hours * hourly_rate
print(amount_due)
```

```
513.0
513.0
```

Assignments and expressions The = symbol represents the assignment operation that is copying in the left side variable the value or the expression result on the right side.

For instance,

$$a = 39$$

means that the constant value 39 is assigned to variable a. It can be thought as

$$a \leftarrow 39$$

Similarly, we can assign the result of an expression to a variable, like

$$a = a + 3$$

which means that the original value of a is incremented by 3 and then reassigned to the same variable thus reusing the same memory location

$$a \leftarrow a + 3$$

An expression is a combination of values, operators and functions that should be evaluated before assigning its result to a variable.

[21]:
```
x = a / 24 * (1 + b)     # this is an expression
print(x)
```

```
22.958333333333332
```

Expressions are evaluated according to common mathematical rules and make use of the following symbols to express arithmetical operations:

Operator

Operation

+

addition

-

subtraction

*

multiplication

/

division

**

power

%

reminder

Expressions evaluation is subject to operator precedence rules; from highest to lowest precedence, python considers:

$$parentheses \rightarrow exponentiation \rightarrow multiplication, division\ and\ reminder \rightarrow addition\ and\ subtraction \rightarrow lef$$

The expressions below show the effect of operator precedence:

[22]:
```
x = 1 + 2**3 - 10 / 2 * 3
y = 1 + 2**3 - 10 / (2 * 3)
print(x,y)
```

```
-6.0 7.333333333333333
```

Types In python all variables and constants have a type and the interpreter is able to execute the appropriate actions according to items' type.

Some type conversions are applied implicitly by the language interpreter, like integer to float, but some others require explicit type casting, such as string to number and viceversa.

[23]:
```
x = 3 + 4.3   # adding numerical values with implicit type conversion
print(x)
```

```
7.3
```

[37]:
```
x = '3'+'4.3'   # concatenating strings
print(x)
```

```
34.3
```

[24]:
```
x = '3' + 4.3   # fails because types are not compatible
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 1
----> 1 x = '3' + 4.3

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

[25]:
```
x = float('3') + 4.3
print(x)
```

```
7.3
```

It is possible to display the type of a variable by executing type() on the desired element.

[26]:
```
type(x), type('3'), type(3)
```

[26]:
```
(float, str, int)
```

The user can assign values to program variables at runtime by means of the input() function, which always returns a string and therefore requires type conversion if the expected value is a number.

[27]:
```
val = input('Give me a number: ')
print(type(val))
```

```
Give me a number: 34
<class 'str'>
```

10 Conditional Execution

The execution of a program is often altered by a condition being true or false, that is a boolean expression whose result affects the program flow.

Boolean expressions use comparison operators to look at the variables without changing their content. These operators are

Operator

Meaning

<

less than

<=

less than or equal to

==

equal to

=

greater than or equal to

greater than

!=

not equal to

The comparison operator for equal to is ==, not to be confused with = which is the assignment operator.

The conditional statement is if-elif-else and it can be used in different forms.

Python has no switch statement: it can be replaced by multi-way decisions.

[29]:
```
# one-way decision
x = 3
```

```
print('start')
if x == 3:
    print('the value is 3')
print('end')
```

```
start
the value is 3
end
```

[31]:
```
# two-way decision
x = 5
print('start')
if x == 3:
    print('the value is 3')
else:
    print('the value is not 3')
print('end')
```

```
start
the value is not 3
end
```

[34]:
```
# multi-way decision
x = 5
print('start')
if x > 3:
    print('the value is greater than 3')
elif x < 3:    # there can be more than one elif
    print('the value is less than 3')
else:
    print('the value is equal to 3')
print('end')
```

```
start
the value is greater than 3
end
```

Multi-way decisions can be tricky: always pay attention to how the tests are set up because some branches may never be executed, as shown in the example below.

[38]:
```
x = 7
if x < 2 :
    print('Below 2')
elif x < 20 :
    print('Below 10')
elif x < 10 :
    print('Below 20')  # this option will never run
else :
    print('Something else')
```

```
Below 10
```

[39]:
```
x = 45
if x < 2 :
    print('Below 2')
elif x < 10 :
    print('Below 10')
elif x < 20 :
    print('Below 20')
else :
    print('Something else')
```

```
Something else
```

11 Indentation

Indentation in Python is mandatory and it is required to define code blocks containing all statements that have to be executed within the same scope. Nesting statements implies an increase in indentation and the definition of a new block.

Indentation can be done with either spaces or the tab key but sometimes they cannot be mixed together in the same file.

[ ]:
```
x = 5
print('start')
if x > 2:
    print('the value is greater than 2')        # block 1
    print('the values is '+str(x))              # block 1
    if x > 4:                                   # block 1
        print('the value is greater than 4')    # block 2
else:
    print('the value is less than 2')           # block 3
print('end')
```

12 Try-except

This block is very useful when a piece of code is somehow dangerous and program execution has to continue nonetheless.

If the code in the try block works, then the except is skipped, otherwise an exception is raised and trapped in the except branch.

[40]:
```
val = '145'

try:
    v = int(val)
    v += 2    # v = v + 2
except:
```

```
    v = 'not an integer'

print(v)
```

```
147
```

[41]:
```
val = 'hello'

try:
    v = int(val)
    v += 2
except:
    v = 'not an integer'

print(v)
```

```
not an integer
```

If necessary, one can trap only specific errors and ignore all others. In the following example, we want to handle the division by zero error only. Therefore, the error in value conversion can still interrupt the execution.

[42]:
```
val = 'hello'

try:
    v = int(val)/0
    v += 2
except ZeroDivisionError:
    v = 'division by zero is forbidden'

print(v)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[42], line 4
      1 val = 'hello'
      3 try:
----> 4     v = int(val)/0
      5     v += 2
      6 except ZeroDivisionError:

ValueError: invalid literal for int() with base 10: 'hello'
```

13 Functions

Functions are self-contained modules that can accomplish a specific task: they can take some data as an input and return a result after processing.

Functions are useful to * increment code reuse * reduce code complexity by dividing it into logical chunks * provide a library of tools to be used across different applications

In Python we have:

- **built-in** functions, provided as part of the language itself, e.g. print(), type();
- **user-defined** functions, written by programmers to create reusable code snippets.

Please note that **built-in** functions are treated as reserved keywords and therefore cannot be used as variable names.

A function can be defined through the def keyword, may have arguments and may return one or more values. The definition of a function does not execute its code.

def my_func(param1, param2, …): # this is the function body and it MUST be indented # with respect to function definition keyword

```
    ... do some processing ...

    # return results
    return output_values
```

The variables between brackets are placeholders for the function inputs and are termed **parameters**; when a function is invoked, or called, these placeholders are replaced by the **arguments** and the function code is executed. Each time a function is called, we can pass different arguments.

arg1 = 'hello' arg2 = 12.4 … other arguments …

```
# assign return value to a variable
ret = my_func(arg1, arg2, ...):
```

A function may have zero or more parameters, which in python can also be optional. Moreover, the output of a function can be used as argument to another one.

Examples

[43]:
```
m = max('This is a message')
print(m)
```

```
s
```

[44]:
```
arg = 'This is another message'
print(max(arg))
```

```
t
```

[45]:
```
# a function with an optional parameter
def power(num, exp=2):
    return num**exp
```

[48]:
```
print(power(3,0.5))
```

```
1.7320508075688772
```

REMEMBER: *parameters* are used in function definition, whereas *arguments* are assigned when the function is called.

It is possible to define a function does not return a value: in this case, it is termed **procedure**.

14 Iterations

In order to implement the repeated execution of a piece of code python provides the following iteration statements.

while <conditions>: Is should be used when the number of iterations is not known upfront and the execution depends on one or more conditions: they are called **indefinite** loops and the execution continues until the condition becomes **False**.

[49]:
```
while True:    # this may lead to infinite loops
    str = input('Type something: ')
    print('You typed '+str)
    if str == 'stop':
        print('Stopping loop')
        break
```

```
Type something: Danilo
You typed Danilo
Type something: 13
You typed 13
Type something: stop
You typed stop
Stopping loop
```

[50]:
```
while True:
    str = input('Type something: ')
    if str == 'skip':
        continue   # skips all subsequent statements

    print('You typed '+str)
    if str == 'stop':
        print('Stopping loop')
        break
```

```
Type something: Danilo
You typed Danilo
Type something: skip
Type something: stop
You typed stop
Stopping loop
```

[51]:
```
loop = True    # boolean condition

while loop:
```

```
    str = input('Type something: ')
    print('You typed '+str)
    if str == 'stop':
        print('Stopping loop')
        loop = False
```

```
Type something: Danilo
You typed Danilo
Type something: stop
You typed stop
Stopping loop
```

for <value> in <set of values>: This statement is used any time the number of iterations is finite: it is a **definite** loop that iterates through the members of a set.

The iteration variable is sequentially assigned the elements of the list and its content can be used within the code block, which is executed as many times as the list size.

[54]:
```
for i in [5,4,3,2,1]:
    print(i)
print('Boom!!!')
```

```
5
4
3
2
1
Boom!!!
```

[55]:
```
for i in range(1,10):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

[56]:
```
for s in ['Honda', 'Yamaha', 'Aprilia', 'Ducati']:
    print(s)
```

```
Honda
Yamaha
Aprilia
Ducati
```

In Python, you can slice a list to extract a subset of its elements. The syntax for slicing a list is my_list[start:end], where start is the index of the first element to include in the slice, and end is the index of the first element to exclude from the slice. If start is omitted, it defaults to 0. If end is omitted, it defaults to the length of the list.

In the given code, my_list[1:] is a slice of my_list that includes all elements from the second element (index 1) to the end of the list. This means that v will take on the values of each element in my_list, except for the first element (which is assigned to mx outside the loop).

The loop then iterates through each of these values, comparing each value to the current maximum value mx and updating mx if the current value is greater than mx

[57]:
```
# find the maximum of a list of values
my_list = [3,41,58,32,5,68,34,21]

mx = my_list[0]
for v in my_list[1:]:
    if v > mx:
        mx = v

print('The largest values is',mx)
```

```
The largest values is 68
```

15 Strings

A string is a sequence of characters containing any printable symbol, defined between quotes or double-quotes.

Strings can also contain numbers, in which case they must be converted into numerical variables before any computation. The + operator on strings means concatenation.

[60]:
```
s = 'Hello'
t = '12.75'

r = s+t
print(r)

print(float(t)+1)
print(int(t[:2]))   # this works because an integer number is selected
print(int(t[3:]))   # this works because an integer number is selected

print(int(t))  # this fails because t is a floating point value
```

```
Hello12.75
13.75
12
75
```

```
ValueError                                Traceback (most recent call last)
Cell In[60], line 11
      8 print(int(t[:2]))   # this works because an integer number is selected
      9 print(int(t[3:]))   # this works because an integer number is selected
---> 11 print(int(t))

ValueError: invalid literal for int() with base 10: '12.75'
```

Each character can be addressed individually like a list element, using an index between square brackets, but it cannot be modified: only the whole string can be replaced.

The index values are either integers or integer expressions, start at zero and end at string length minus one. String length is returned by the len built-in function.

List slicing rules apply using the *colon* operator. The basic syntax for slicing is: list[start:stop[:step]] where * start is the first element to be selected * stop is the first element to be excluded * step is the increment between two selected positions

[61]:
```
t[2] = 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[61], line 1
----> 1 t[2] = 3

TypeError: 'str' object does not support item assignment
```

[62]:
```
print(len(r))
```

```
10
```

[64]:
```
# looping over a string
ix = 0

while ix < len(s):
    print(r[ix])
    ix += 1
```

```
H
e
l
l
o
```

[65]:
```
# a better way to loop over a string
for char in s:
    print(char)
```

```
H
e
l
l
o
```

The input function is used to read user data: it returns a string value to be parsed and converted as needed. This allows for better control over user input or possible errors.

The in keyword can also be used as a logical operator to check whether a string is contained into another:

[66]:
```
print(s)
'll' in s
```

```
Hello
```

[66]:
```
True
```

Strings are compared in alphabetical order according to the selected charset (ASCII, UTF-8, etc.), using the comparison operators described above.

The string class defines many built-in functions that can be used for string processing.

[67]:
```
type(s)
dir(s)
```

[67]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
```

```
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'maketrans',
'partition',
'removeprefix',
'removesuffix',
'replace',
'rfind',
'rindex',
'rjust',
'rpartition',
'rsplit',
```

```
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

Inline **help** is available for every class method by prepending the ? symbol to class or variable method:

[68]:
```
?str.find
# or ?s.find
```

For instance, if we want to find the first occurrence of a substring in a text, the find class method should be used as follows:

[69]:
```
text = 'this is my sample message'

pos = text.find('my')
print(pos)

pos = text.find('x')  # if substring is not found, -1 is returned
print(pos)

addr = 'john.doe@the.university.edu'
splitat = addr.find('@')
uname = addr[:splitat]
domain = addr[splitat+1:]
print(uname)
print(domain)


# or else
parts = addr.split('@')
print(parts)
```

```
8
-1
john.doe
the.university.edu
['john.doe', 'the.university.edu']
```

16 Lists

Lists are a collection of elements associated to a single variable.

Each element can be any python object and even another list. Its elements can be addressed with slicing as already explained with strings.

A list can also be empty: this is useful for initialization.

Examples

[ ]:
```
l1 = [11,32,45,1,6,78]
l2 = ['cat','mouse','dog','horse']

l3 = [l1,l2]
print(l3)
```

List l3 is a bi-dimensional object whose elements can be accessed as follows:

[ ]:
```
l3[0][2], l3[1][1]
```

Lists are **mutable** objects, hence we can change any item as needed. Most frequently, lists are assigned **by reference** therefore any changes to a list item may be reflected into other objects.

[ ]:
```
l1[1] += 3
print(l3)
```

Similarly to strings, the + operator concatenates two or more lists into a new one. Moreover, the in operator can be used to verify the presence of a value in a list.

[ ]:
```
l4 = l1 + l2
print(l4)

print('bee' in l4)
```

[ ]:
```
# available list methods
dir(list())
```

Lists can also be created dynamically by adding elements to an empty list:

[ ]:
```
lx = list()    # or lx = []

lx.append(3)
lx.append('good')

print(lx)
```

17 Dictionaries

A Dictionary is a collection of labelled items organized in key-value pairs, such as:

a_dict = {'key1': value1, 'key2': 'value2', ... }

They are the most powerful data collection in python and are supported, with alternative names, in many other programming languages. Dictionary elements are addressed through their label and can be modified as needed.

Examples

[ ]:
```
dc = dict()

dc['crypto'] = 'Ethereum'
dc['price'] = 2000.0000
dc['24h Change %'] = 3

# print the dictionary
print(dc)

# print all keys
print(dc.keys())

# print all values
print(dc.values())

# print all dictionary pairs
print(dc.items())
```

[ ]:
```
dc['price'] = 2100.0
print(dc)
```

[ ]:
```
# list all available methods
dir(dc)
```

[ ]:
```
# word occurrence count
words = ['this','is','my','hat','my','black','hat']

woc1 = dict()
for word in words:
    if word in woc1:
        woc1[word] += 1
    else:
        woc1[word] = 1

print(woc1)
```

[ ]:
```
# a better way to count occurences
woc2 = dict()
for word in words:
    woc2[word] = woc2.get(word,0) + 1

print(woc2)
```

18 Tuples

A Tuple is a collection of items that cannot be altered once created. It is an **immutable** list.

a_tuple = (item1, item2, …)

Note that lists are declared with square brackets, whereas tuples make use of round brackets. Tuple elements can be copied into variables with a single assignment for all variables.

Examples

[ ]:
```
tu = (2.13,'dog', 102)
print(tu)

print(tu[1])

tu[1] = 'tiger'
```

[ ]:
```
dir(tu)
```

[ ]:
```
a,b,c = tu
print(a)
print(b)
print(c)
```

19 Any Questions?

20 Homeworks

- find average of a list of values
- find the smallest value of a list
- solve quadratic equation