Python Libraries For Data Science

Last lecture reminder

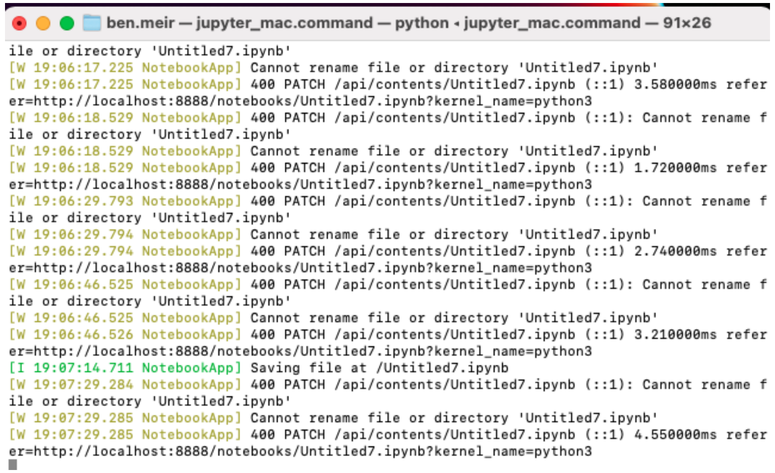
We learned about:

- What is Data science
- Structured Data vs Unstructured Data
- The Data science role in companies and organizations
- Other different Data related roles in companies and organizations
- What are python libraries
- Introduction to Anaconda & Jupyter Notebook

Working with Jupyter Notebook

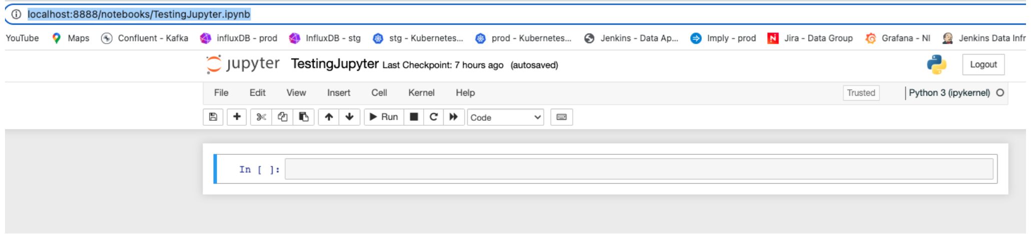
Jupyter Notebook is running with 2 components:

- The Jupyter server which is your computer CMD / Terminal. If we will close this CMD window the notebook itself will be closed.
- The Notebook UI which running on http://localhost:8888/ and can be opened via our web browser.



ben.meir — jupyter_mac.command — python ◂ jupyter_mac.command — 91×26

```
ile or directory 'Untitled7.ipynb'
[W 19:06:17.225 NotebookApp] Cannot rename file or directory 'Untitled7.ipynb'
[W 19:06:17.225 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1) 3.580000ms refer
er=http://localhost:8888/notebooks/Untitled7.ipynb?kernel_name=python3
[W 19:06:18.529 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1): Cannot rename f
ile or directory 'Untitled7.ipynb'
[W 19:06:18.529 NotebookApp] Cannot rename file or directory 'Untitled7.ipynb'
[W 19:06:18.529 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1) 1.720000ms refer
er=http://localhost:8888/notebooks/Untitled7.ipynb?kernel_name=python3
[W 19:06:29.793 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1): Cannot rename f
ile or directory 'Untitled7.ipynb'
[W 19:06:29.794 NotebookApp] Cannot rename file or directory 'Untitled7.ipynb'
[W 19:06:29.794 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1) 2.740000ms refer
er=http://localhost:8888/notebooks/Untitled7.ipynb?kernel_name=python3
[W 19:06:46.525 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1): Cannot rename f
ile or directory 'Untitled7.ipynb'
[W 19:06:46.525 NotebookApp] Cannot rename file or directory 'Untitled7.ipynb'
[W 19:06:46.526 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1) 3.210000ms refer
er=http://localhost:8888/notebooks/Untitled7.ipynb?kernel_name=python3
[I 19:07:14.711 NotebookApp] Saving file at /Untitled7.ipynb
[W 19:07:29.284 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1): Cannot rename f
ile or directory 'Untitled7.ipynb'
[W 19:07:29.285 NotebookApp] Cannot rename file or directory 'Untitled7.ipynb'
[W 19:07:29.285 NotebookApp] 400 PATCH /api/contents/Untitled7.ipynb (::1) 4.550000ms refer
er=http://localhost:8888/notebooks/Untitled7.ipynb?kernel_name=python3
```



localhost:8888/notebooks/TestingJupyter.ipynb

TestingJupyter Last Checkpoint: 7 hours ago (autosaved)

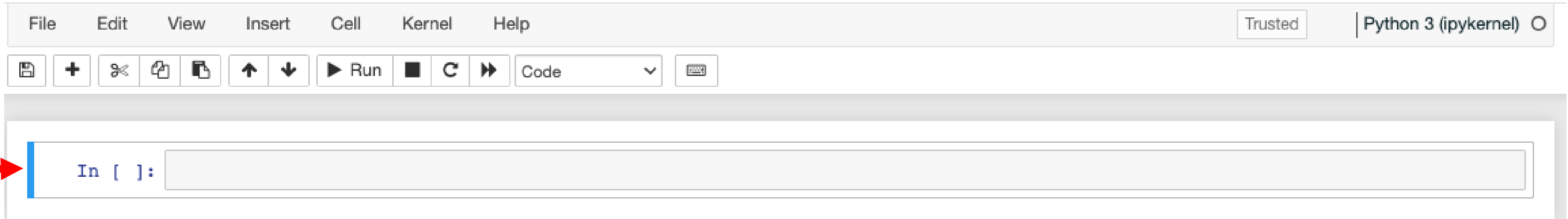
Working with Jupyter Notebook

In the notebook we have 2 main modes the we can use:

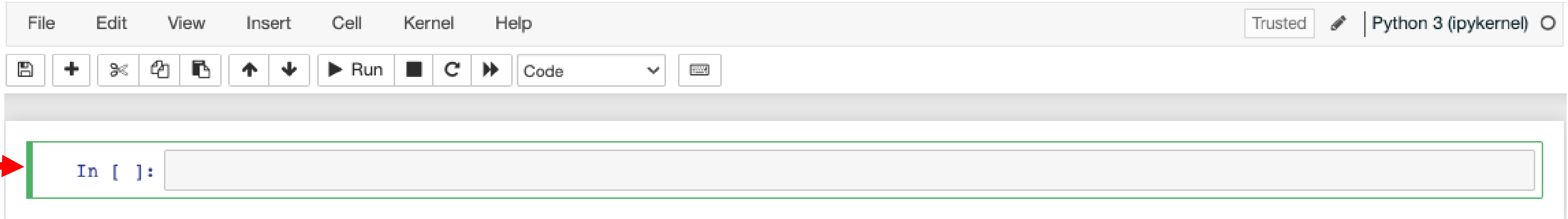
- **Edit mode** → In this mode we can modify our existing code, add / remove lines of code. The edit mode can be recognized by the green border.
- **Command mode** → In this mode we can move between notebook lines but we can't change content of a specific cell. The command mode can be recognized by the blue border.

From command mode → edit mode (press enter), From edit mode → command mode (press esc)

Command Mode



Edit Mode



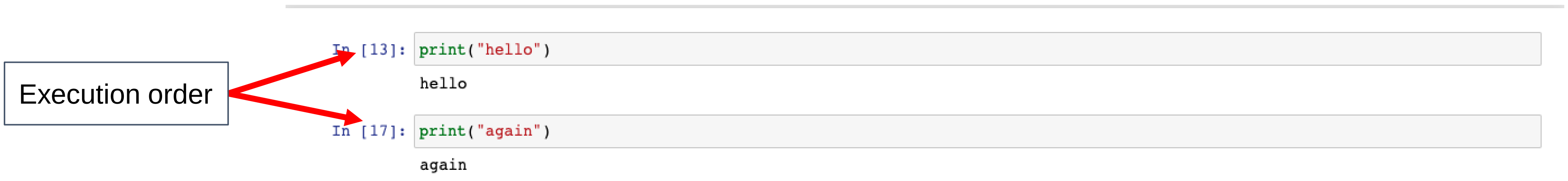
Jupyter Notebook - Run Cells

We can run the code inside an existing cell using the notebook menu or using the shortcut:

- **Cell** → **Run Cells** → Will run the cell that we are on and all the cells above it
- **Cell** → **Run Cells and Select Below** → Will active like “Run Cells” but also will select the next cell below the cell that we are on. If we don’t have a cell below it will insert a new one.
- **Cell** → **Run Cells and Insert Below** → Will active like “Run Cells” but also will insert a new cell below the cell that we are on.

In addition we can use the keyboard shortcut to run the line → **CTRL + ENTER**

On the left side of each cell we can see a counter that notify the execution order. The number represent the execution time this cell has been executed.



Execution order

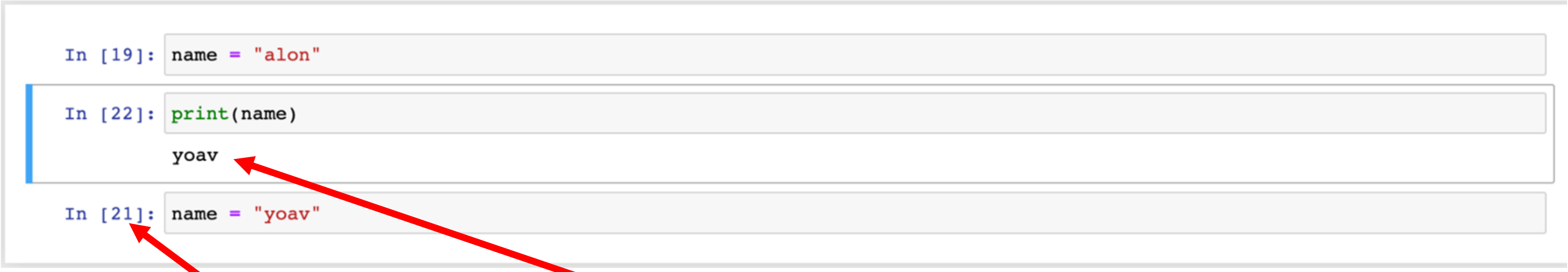
```
In [13]: print("hello")
         hello
In [17]: print("again")
         again
```

Jupyter Notebook - Cell Execution Order

We need to understand that Jupyter Notebook works like a Python interpreter, meaning that once it run a specific line of code this code will affect other cells execution regardless the position of the cell.

The execution order is what determine and not the cells order.

For example:



```
In [19]: name = "alon"

In [22]: print(name)

         yoav

In [21]: name = "yoav"
```

We can also look at the execution order counter to see why we are getting the results we see

We created a new variable with the value “alon” than in the 3th cell we changed the variable value to “yoav” and print it in the second cell.
We can see that the value in the variable was changed according to the execution order and not the cell order

Jupyter Notebook - Insert / Remove Cells

In order to insert a cell we can choose from two options:

- Insert a new cell below the chosen cell, in the menu go to **Insert** → **Insert Cell Below**
- Insert a new cell above the chosen cell, in the menu go to **Insert** → **Insert Cell Above**

We can also use shortcut to insert a new cell by selecting a cell in command mode and press **B**.

In order to remove existing cell we can select the cell \ cells we want to delete

and in the menu go to **Edit** → **Delete Cells**

We can also use the shortcut to delete selected cell by pressing **D twice**.

When deleting with the shortcut we must be in edit mode.

Jupyter Notebook - Markdown Cells

One of the main features when using a Jupyter Notebook is the ability to add markdown text (similar to HTML text) in addition to the Python code itself.

This feature provide us the ability to add explanation and pictures to our Python code so it will be more understandable.

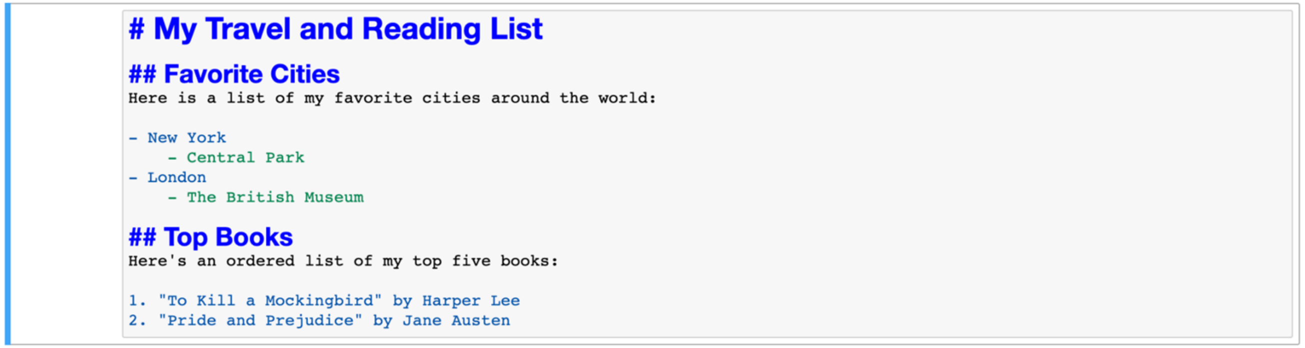
In order to add markdown code we need to select a cell and change its type to “Markdown”.

We can do it by selecting a cell and go in the menu to **Cell → Cell Type → Markdown**

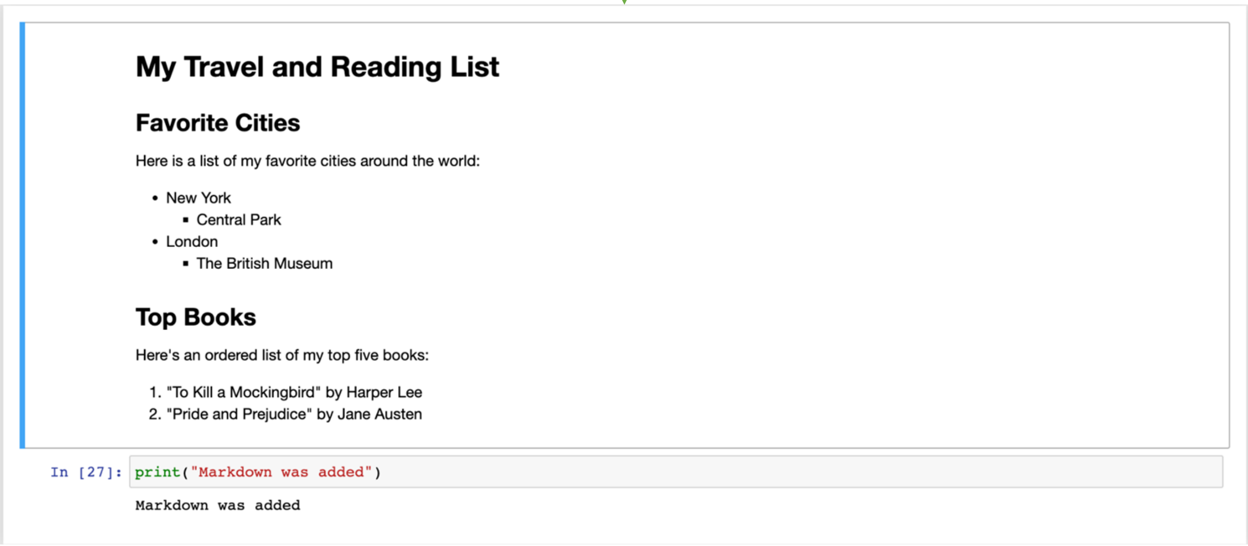
Note: In order to see the markdown text in the presentation form we need to execute the markdown cell type like with the code cell types.

Jupyter Notebook - Markdown Cells

For example → Let’s see how markdown text is look like before and after execution:



```
# My Travel and Reading List
## Favorite Cities
Here is a list of my favorite cities around the world:

- New York
    - Central Park
- London
    - The British Museum

## Top Books
Here's an ordered list of my top five books:

1. "To Kill a Mockingbird" by Harper Lee
2. "Pride and Prejudice" by Jane Austen
```



My Travel and Reading List

Favorite Cities

Here is a list of my favorite cities around the world:

- New York
  - Central Park
- London
  - The British Museum

Top Books

Here's an ordered list of my top five books:

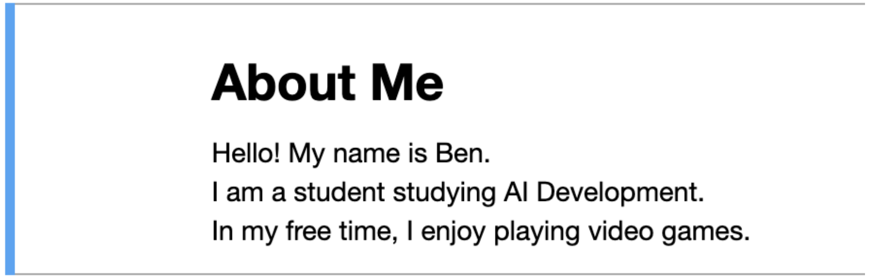
1. "To Kill a Mockingbird" by Harper Lee
2. "Pride and Prejudice" by Jane Austen

```
In [27]: print("Markdown was added")

Markdown was added
```

Class Exercise - Jupyter Notebook Basic

Instructions:

- Launch Jupyter notebook from the Anaconda navigator.
- Create a new Python3 Jupyter Notebook instance.
- Insert 4 new cells to the notebook and implement the following:
- The first cell should be a markdown cell with the following details - change the relevant details according your own :



About Me

Hello! My name is Ben.
I am a student studying AI Development.
In my free time, I enjoy playing video games.

- In the next 3 cells print each row from the markdown cell using Python code.
- Execute all 4 cells together.

ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

Class Exercise Solution - Jupyter Notebook Basic

NumPy

Introduction to NumPy Library

NumPy, which stands for Numerical Python, is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
It is particularly useful in scientific computing contexts, as it offers efficient operation on data structures often used in machine learning, data science, and other complex computations.

NumPy commonly usages are:

- **Mathematical and Logical Operations** → It is used to perform logical and mathematical operations on arrays and matrices.
- **Operations related to linear algebra** → determinant, rank, and solving linear equations.
- **Random Simulation** → It is capable of simulating random data which is often useful in statistical analysis and machine learning techniques.
- **Image processing** → Numpy's multi-dimensional array is also used in image processing by transforming images into arrays.

Introduction to PIP

PIP, which stands for "Pip Installs Packages" or "Preferred Installer Program", is a package management system used by Python to install and manage software packages written in Python. With pip we will be able to install other Python external libraries and use them in your Python programs.

In order to install library with pip we need to open our computer CMD and run this command:

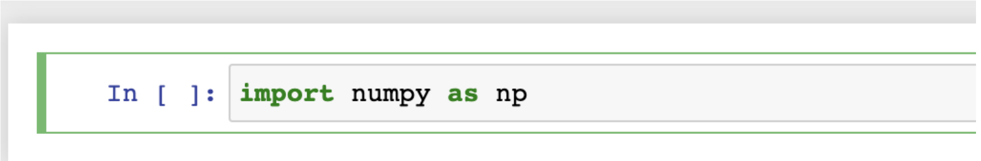
```
pip install ${Python library name}
```

Once the installation ended successfully we will be able to import the library we installed and use it.

Installing & Importing NumPy

In order to work with the NumPy library we will first need to install and import it by running those commands:

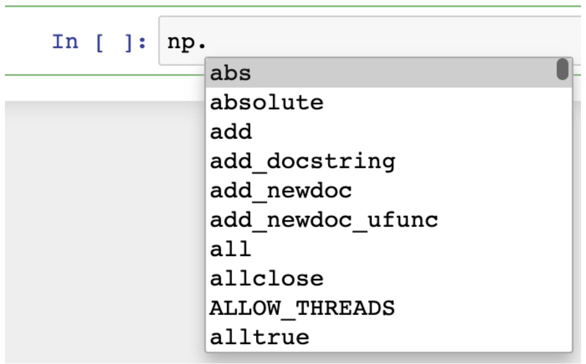
- In the computer CMD → **pip install numpy**
- In Jupyter Notebook cell → **import numpy as np**



```
In [ ]: import numpy as np
```

What we did was to import all the numpy functions into a variable called np.

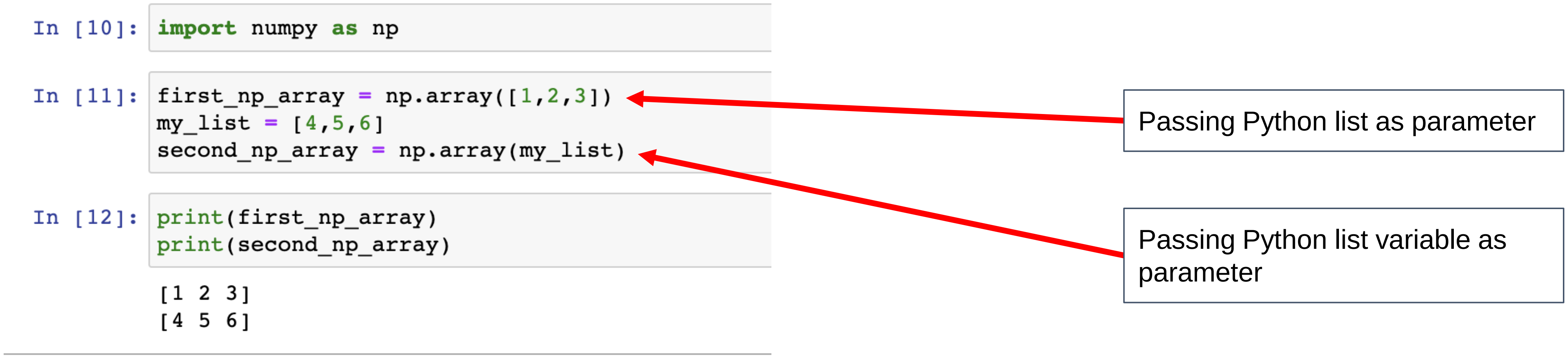
By pressing np. + TAB we will be able to see all the different functions that numpy has.



```
In [ ]: np.
        abs
        absolute
        add
        add_docstring
        add_newdoc
        add_newdoc_ufunc
        all
        allclose
        ALLOW_THREADS
        alltrue
```

NumPy Array Functions

np.array() → The array() function create new Numpy array by taking a Python list as a parameter. We can pass Python list variable as the parameter as well.

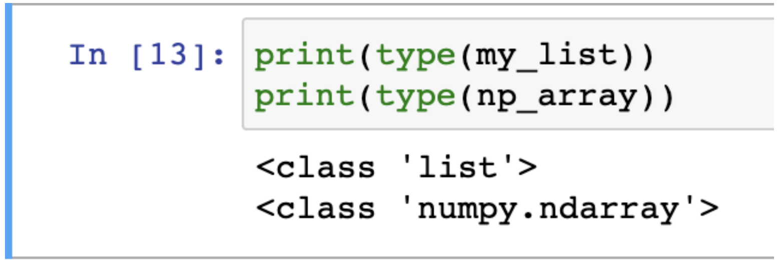


```
In [10]: import numpy as np

In [11]: first_np_array = np.array([1,2,3])
         my_list = [4,5,6]
         second_np_array = np.array(my_list)

In [12]: print(first_np_array)
         print(second_np_array)

         [1 2 3]
         [4 5 6]
```

Passing Python list as parameter

Passing Python list variable as parameter

We can also print the Numpy array type and see that its of type **numpy.ndarray**



```
In [13]: print(type(my_list))
         print(type(np_array))

         <class 'list'>
         <class 'numpy.ndarray'>
```

NumPy Arrays vs Python Lists

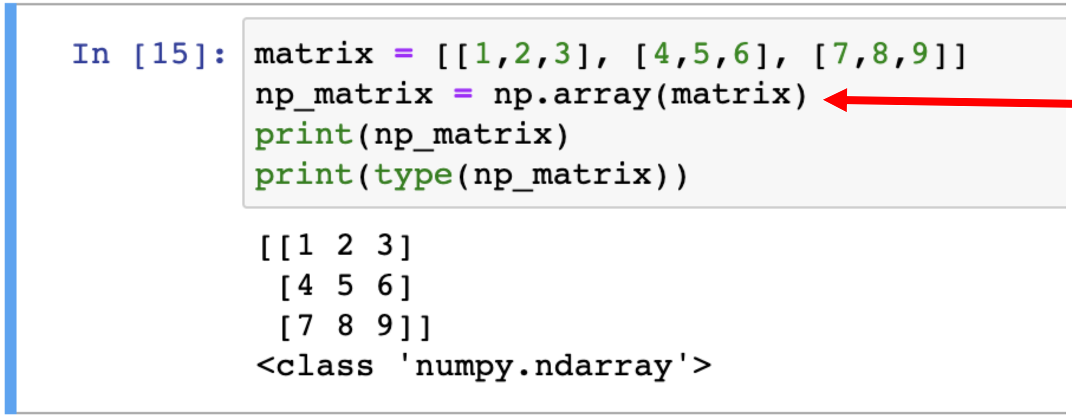
Arrays in Numpy are very similar to lists in Python but can be much more efficient and with more abilities than the standard Python lists.

The main differences between Numpy arrays and Python lists are:

- **Speed** → NumPy arrays are more compact and faster than lists in Python, especially for operations involving large amounts of numeric data. Numpy uses less memory to store data and it provides a fast way to operate on this data.
- **Functionality** → NumPy arrays come with a wide range of functions that are not available with lists. For instance, you can carry out arithmetic operations (addition, multiplication, etc.) on the entire array which is not possible with lists. Furthermore, functions for statistical analysis, linear algebra, and Fourier transforms are also available.
- **Memory** → Numpy arrays also use much less memory as compared to lists. This can be very important when handling large data sets.
- **Compatibility** → Many scientific and mathematical Python-based packages, e.g. SciPy, Matplotlib, and Pandas, use NumPy arrays.

NumPy Array Functions

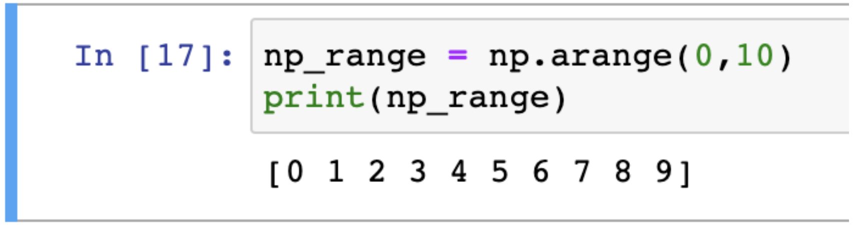
We can also use np.array() to convert matrix into Numpy arrays and it will work the same:



```
In [15]: matrix = [[1,2,3], [4,5,6], [7,8,9]]
         np_matrix = np.array(matrix)
         print(np_matrix)
         print(type(np_matrix))

         [[1 2 3]
          [4 5 6]
          [7 8 9]]
         <class 'numpy.ndarray'>
```

Passing Python matrix as parameter

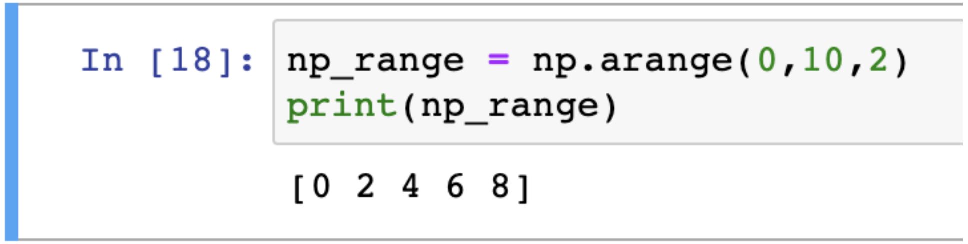
np.arange() → The arange() function allow us to create a new Numpy array with the range of numbers we passed as parameter. For example - np.arange(0,10) → will create a new Numpy array with the numbers 0 - 9 (the first parameter is included and the second parameter is not included).



```
In [17]: np_range = np.arange(0,10)
         print(np_range)

         [0 1 2 3 4 5 6 7 8 9]
```

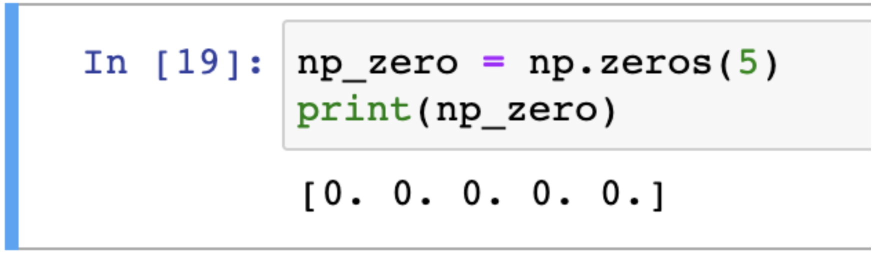
NumPy Array Functions

np.arange() → We can also pass a third parameter that will determine the “jumps” between the numbers in the range. For example - np.arange(0,10, 2) → will create a new Numpy array with the numbers 0 - 9 with “jumps” of 2 between each number in the range.



```
In [18]: np_range = np.arange(0,10,2)
         print(np_range)

         [0 2 4 6 8]
```

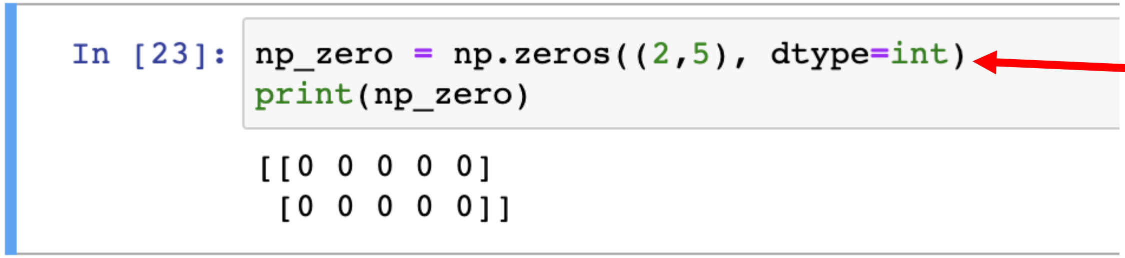
np.zeros() → The zeros() function will create an array of zeros, the parameter we pass to this function will determine the size of the array. For example - np.zeros(5) → will create an array with 5 zeros inside.



```
In [19]: np_zero = np.zeros(5)
         print(np_zero)

         [0. 0. 0. 0. 0.]
```

NumPy Array Functions

np.zeros() → We can also create multiple dimensional arrays populated with zeros by passing a second parameter. For example - np.zeros((2, 5)) → will create a 2 dimensional array with 5 zeros in each row. Also by passing **dtype** parameter we can determine the type of element Numpy is putting inside the array

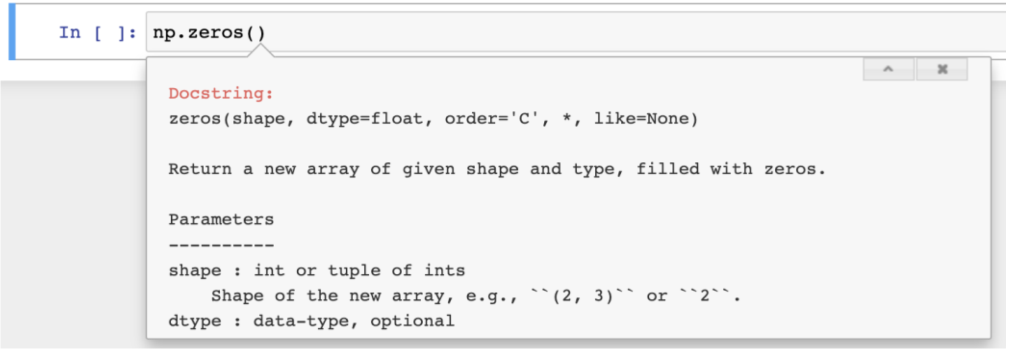


```
In [23]: np_zero = np.zeros((2,5), dtype=int)
         print(np_zero)

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

We are passing dtype=int so the numbers in the array will be of type int

Function description → By pressing **SHIFT + TAB** while selecting the function () we can see the explanation

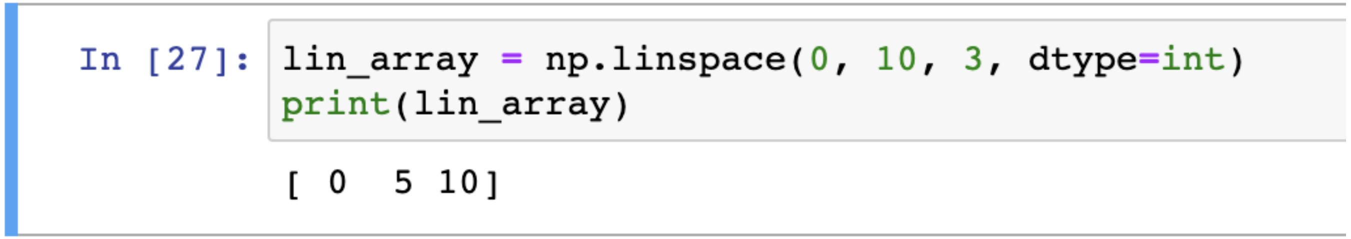
has:



```
In [ ]: np.zeros()

Docstring:
zeros(shape, dtype=float, order='C', *, like=None)

Return a new array of given shape and type, filled with zeros.

Parameters
----------
shape : int or tuple of ints
    Shape of the new array, e.g., ``(2, 3)`` or ``2``.
dtype : data-type, optional
```

NumPy Array Functions

np.linspace() → The linspace() function will return Numpy array with the amount of numbers requested between the given range with a similar space between them. For example - np.linspace(0, 10, 3) → will create an array of 3 numbers between the range 0 - 10 that has similar space between them all.



```
In [27]: lin_array = np.linspace(0, 10, 3, dtype=int)
         print(lin_array)

         [ 0  5 10]
```

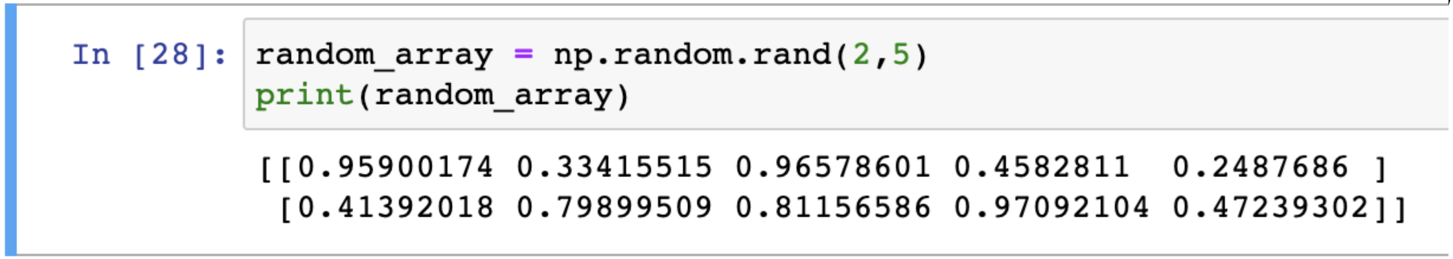
np.random.rand() → The rand() function will return array of random decimal numbers between 0 -1

with

unit probability. We can also determine what will be the size of the array by passing it as parameter.

s and 5 columns with random

numbers

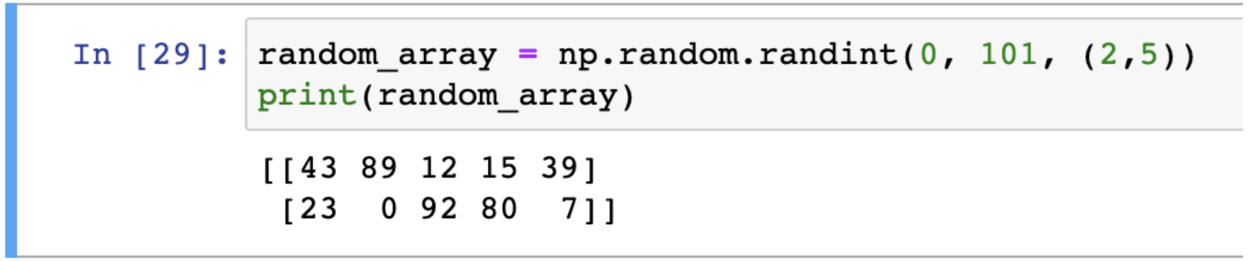


```
In [28]: random_array = np.random.rand(2,5)
         print(random_array)

         [[0.95900174 0.33415515 0.96578601 0.4582811  0.2487686 ]
          [0.41392018 0.79899509 0.81156586 0.97092104 0.47239302]]
```

NumPy Array Functions

np.random.randint() → Working the same as the rand() function but will generate integers type numbers with unit probability from a given range.

For example - np.random.randint(0, 101, (2,5)) → will create a matrix of 2 rows and 5 columns with random numbers between 0 and 101.

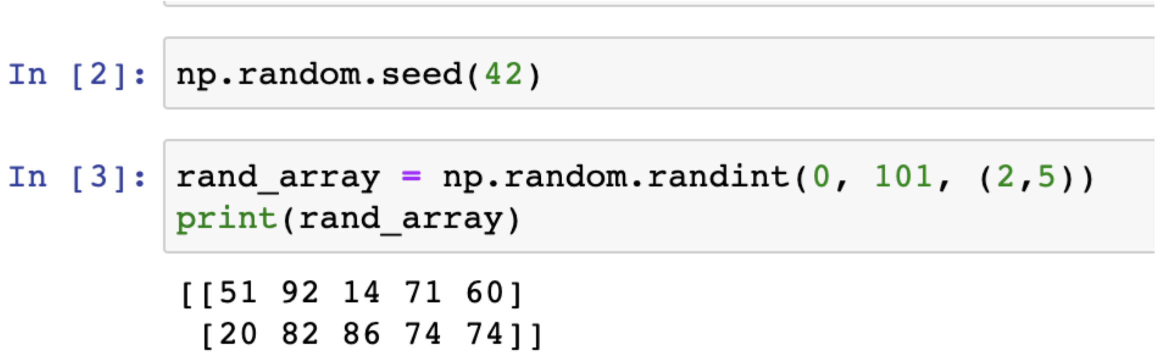


```
In [29]: random_array = np.random.randint(0, 101, (2,5))
         print(random_array)

         [[43 89 12 15 39]
          [23  0 92 80  7]]
```

Random seed → A random seed refers to the initial configuration that is used to generate a sequence of random numbers in a pseudo-random number generator (PRNG), a type of algorithm used in computer simulations. Random seeds are used to initialize the algorithm, and a specific seed value will always produce the same sequence of random numbers.

NumPy Array Functions

np.random.seed() → The seed() function allow us to determine the seed value we want Numpy to use in order to generate random numbers. If we will use the same seed value the random numbers Numpy generates will always be the same.



```
In [2]: np.random.seed(42)
```

```
In [3]: rand_array = np.random.randint(0, 101, (2,5))
        print(rand_array)
```

```
[[51 92 14 71 60]
 [20 82 86 74 74]]
```

Note: Using the same seed will always generate the same random numbers but the numbers will change in each execution, meaning the first execution will always generate the same numbers and than the second, the third and so on.