



# Python Libraries

## For Data Science



**ECOM SCHOOL**

המכללה למקצועות הדיגיטל וההייטק

# Last lecture reminder



We learned about:

- Introduction to Jupyter Notebook & Anaconda
- Basic functionality when working with Jupyter Notebook
- How to install and import Python libraries
- Introduction to Numpy Python library
- Numpy arrays vs Python lists
- Different Numpy arrays functions
- What is random seed and for what it used for

# Numpy Array Methods

Until now we saw different functions available in the Numpy library. Now we will see methods available as part of the Numpy array type.

**reshape()** → The reshape() function allow us to change the shape of the given array into a different shape. In case the reshaping is not possible we will get exception.

For example - `numpy_array.reshape(5, 5)` → will change the `numpy_array` value into matrix of 5X5.

```
In [6]: numpy_array = np.arange(0,25)
print(numpy_array)
print(numpy_array.reshape(5,5))

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24]
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

We didn't got an exception because array of 25 elements can be fit into 5X5 matrix



# Numpy Array Methods

**max()** → Return the max element in the Numpy array.

**min()** → Return the min element in the Numpy array.

**argmax()** → Return the index of the max element in the Numpy array.

**argmin()** → Return the index of the min element in the Numpy array.

**shape** → Return the current array shape as tuple, shape its a property and not a method so it takes no arguments.

```
In [11]: numpy_array = np.arange(0,25)
         reshaped_array = numpy_array.reshape(5,5)
         print(reshaped_array.max())
         print(reshaped_array.min())
         print(reshaped_array.argmax())
         print(reshaped_array.argmin())
         print(reshaped_array.shape)

24
0
24
0
(5, 5)
```



# Working with Numpy Array

[] → In order to get elements from Numpy array we can use [] and pass the index in the array we want to get its value. It works the same as with Python lists.

```
In [12]: numpy_array = np.arange(0,25)
         print(numpy_array[7])
```

7

If we want to get more than specific element we can select it as well using []:

```
In [13]: numpy_array = np.arange(0,25)
         print(numpy_array[0:7])
```

[0 1 2 3 4 5 6]

By passing [0:7] we are requesting the get all elements in the indexes between 0 - 7 (not included). We are getting back a new array with the elements we requested



# Working with Numpy Array

Additional abilities using [] in Numpy arrays:

```
In [15]: numpy_array = np.arange(0,25)
print(numpy_array[7:])
numpy_array[0:5] = 100
print()
print(numpy_array)
```

```
[ 7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]

[100 100 100 100 100  5  6  7  8  9 10 11 12 13 14 15 16 17
 18 19 20 21 22 23 24]
```

Select all elements from the 7 index until the end of the array

Select all elements from the 0 - 4 indexes and change their values to be 100

[] → Selecting element by index from a Numpy matrix also works the same as with Python matrix.

The first number we pass to [] will be the row index and the second number will be the column index.

```
In [17]: numpy_array = np.arange(0,25)
reshaped_array = numpy_array.reshape(5,5)
print(reshaped_array)
print()
print(reshaped_array[2][4])
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

Selecting the element in the third row and the 5th column

# Working with Numpy Array

**copy()** → Because Numpy arrays are memory objects, if we will want to make changes on a duplicate array but keep the original array as it is we will need to use the copy() method. It's not enough to save the array to a new variable because every change we will make on the new variable (in case we didn't use copy() method) will be affected also on the original one. Using the copy() method can also be called **deep copy**.

For example → Duplicate Numpy array without using copy() vs With using copy():

```
In [21]: original_array = np.arange(0,25)
new_array = original_array
new_array[0:10] = 100
print(original_array)
```

[100 100 100 100 100 100 100 100 100 100 10 11 12 13 14 15 16 17  
18 19 20 21 22 23 24]

Without using copy() the changes on the new\_array were affected also the original\_array

```
In [20]: original_array = np.arange(0,25)
new_array = original_array.copy()
new_array[0:10] = 100
print(original_array)
```

[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23  
24]

With using copy() the changes on the new\_array did not affect the original\_array

# Numpy Array - Conditional Selection

**[condition]** → By providing a true-false condition inside the [], we apply filtering on our Numpy Array.

Only elements that match the condition will return.

value. It works the same as with Python lists.

```
In [9]: np_array = np.arange(0,25)
        filtered_array = np_array[np_array > 4]
        print(filtered_array)

[ 5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

We filter all elements that are greater than 4

We can also apply complex conditions using & (AND), | (OR) -

```
In [10]: np_array = np.arange(0,25)
         filtered_array = np_array[(np_array > 4) & (np_array < 10)]
         print(filtered_array)

[5 6 7 8 9]
```

We filter all elements that are greater than 4 and less than 10





# Numpy Array - Arithmetics

One of the most useful features in Numpy library is applying arithmetic actions on Numpy arrays.

Meaning, by using Numpy arrays we can apply mathematics and statistics calculations on the elements inside the array. This is very useful because with regular Python lists we would have creating function that will apply this logic manually using condition and loops while with Numpy we don't need to write any additional logic.

```
In [12]: np_array = np.arange(0,10)
np_array = np_array + 5
print(np_array)

[ 5  6  7  8  9 10 11 12 13 14]
```

We added 5 to each element inside the array by using the '+' operator

```
In [14]: first_array = np.arange(0,10)
second_array = np.arange(0,10) + 10
print(first_array)
print(second_array)
print(first_array + second_array)

[0 1 2 3 4 5 6 7 8 9]
[10 11 12 13 14 15 16 17 18 19]
[10 12 14 16 18 20 22 24 26 28]
```

If we have two different arrays with the same size, we can add them together using the '+' operator

# Numpy Array - Arithmetics

**sum()** → Return the sum of all elements in the Numpy array.

**mean()** → Return the mean of all elements in the Numpy array.

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

          45
          4.5
```

In 2 dimensional array we can apply those arithmetic methods on a specific row or on a specific column.

**axis = 0** → Calculate the sum by rows, meaning Numpy will sum for each column the entire rows values for that column, when providing axis = 0 we will get the sum of every column.

**axis = 1** → Calculate the sum by columns, meaning Numpy will sum for each row the entire columns values for that row, when providing axis = 1 we will get the sum of every row.



# Numpy Array - Arithmetics

For example →

```
In [25]: np_array = np.arange(0,25)
matrix = np_array.reshape((5,5))
print(matrix)
print()
print("Columns sum is:")
print(matrix.sum(axis=0))
print("Rows sum is:")
print(matrix.sum(axis=1))
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

Columns sum is:

```
[50 55 60 65 70]
```

Rows sum is:

```
[ 10  35  60  85 110]
```

axis = 0 will return an array with the sum of each column

axis = 1 will return an array with the sum of each row





# Class Exercise - Numpy Array

## Instructions:

Implement the following requirements using Numpy array, Python and Jupyter:

- Create a 2-dimensional array with 5 rows and 5 columns. Populate this array with random integers that range from 0 to 100. Make sure your seed set to 101.
- Calculate the following statistics for each row:
  - The sum of the elements
  - The minimum value among the elements
  - The maximum value among the elements
  - The mean (average) of the elements
  - The median of the elements
- Create a 1-dimensional NumPy array. This new array should contain all elements from the previous 2-dimensional array that are greater than 10.



# Class Exercise Solution - Numpy Array



# Pandas



**ECOM SCHOOL**

המכללה למקצועות הדיגיטל וההייטק

# Introduction to Pandas Library

**Pandas** (Python Data Analysis Library) is an open-source data manipulation and analysis library for the Python programming language. It supports structured data operations and manipulations which are pivotal to data processing and analysis tasks. The library provides flexible data structures that enable a fast and efficient manipulation of structured data.

Pandas is especially effective in data preprocessing, data wrangling, and working with data structures like series, dataframe, and panels.

Pandas commonly usages are:

- **Data Wrangling and Cleaning** → Pandas offers extensive capabilities to perform data cleaning operations, reshaping and pivoting of data sets, handling missing data, and merging and joining of datasets.

# Introduction to Pandas Library

Pandas commonly usages are:

- **Statistical Analysis** → It provides built-in methods for descriptive statistics – mean, median, mode, variance etc. and for correlation analysis, covariance, and data summary.
- **Data Exploration** → Pandas enables quick data visualization for better understanding and intuitive data exploration.
- **Working with Different Data Sources** → Pandas supports wide range of data sources to read data from like Excel files, CSV files, SQL databases, HDF format files etc.

In addition to these, it also supports time-series functionality, handling of missing data, and operations like slicing, indexing, and subsetting large datasets.



# Installing & Importing Pandas

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

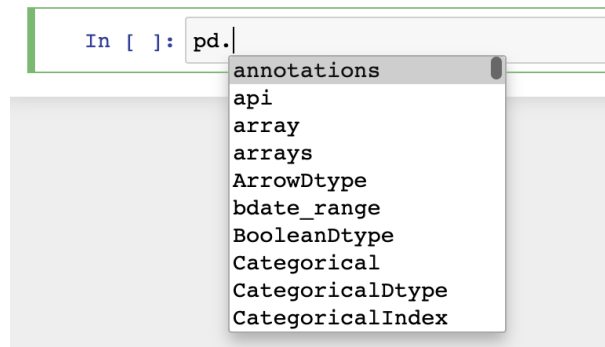
- In the computer CMD → **pip install pandas**
- In Jupyter Notebook cell → **import pandas as pd**

```
In [2]: import pandas as pd
```

What we did was to import all the pandas functions into a variable called pd.

By pressing pd. + TAB we will be able to see all the different functions that pandas

has.



# Pandas Series Functions

**Pandas Series** → A Pandas Series is a one-dimensional array-like object that can hold any data type (integers, strings, floating point numbers, Python objects, etc.). It is a single dimension labelled array capable of holding any type of data.

With Pandas Series we can also create label indexing and not just numeric indexing like with regular lists.

**pd.Series()** → The Series method allow us to create new Pandas Series and pass the data we want to store.

```
In [5]: pd_series = pd.Series([10, 100, 15, 35])
print(pd_series)
print(type(pd_series))

0    10
1   100
2    15
3    35
dtype: int64
<class 'pandas.core.series.Series'>
```

We can pass Python list as the Series data source

When printing Pandas Series we will always see the index of each element at the right column and the element value on the left column



# Pandas Series Functions

**Pandas Series label indexing** → In order to create label indexing we can pass to the Series index parameter with the labels we want to use as index.

```
In [6]: data = [1776, 1867, 1821]
label_index = ['USA', 'CANADA', 'MEXICO']
pd_series = pd.Series(data=data, index=label_index)
print(pd_series)
```

```
USA      1776
CANADA   1867
MEXICO   1821
dtype: int64
```

We are passing an index parameter with list of strings that will be our Series indexes

Now instead of numeric indexing we have label indexing

In Pandas Series numeric and label indexing work the same, meaning we can't have the same label indexing twice and we can get the data according to its matched label.

```
In [8]: data = [1776, 1867, 1821]
label_index = ['USA', 'CANADA', 'MEXICO']
pd_series = pd.Series(data=data, index=label_index)

print(pd_series['USA'])
print(pd_series[0])
```

```
1776
1776
```

We can get data using the label index or using the default numeric index

# Pandas Series Functions

Pandas Series allows us to also pass Python dictionary as the data source. When providing a dictionary the Series will automatically use the dictionary key as the index and the dictionary value as the data for that index.

```
In [9]: data = [1776, 1867, 1821]
label_index = ['USA', 'CANADA', 'MEXICO']

countries_dict = {'USA': 1776, 'CANADA': 1867, 'MEXICO': 1821}
pd_series = pd.Series(countries_dict)

print(pd_series)

USA      1776
CANADA   1867
MEXICO   1821
dtype: int64
```

We are passing Python dictionary as the Series data source

**series.keys()** → Will return index object, that act like Python list, contain all the Series indexes.

```
In [20]: print(pd_series.keys())
print(pd_series.keys()[0])
print(type(pd_series.keys()))

Index(['USA', 'CANADA', 'MEXICO'], dtype='object')
USA
<class 'pandas.core.indexes.base.Index'>
```

# Pandas Series Functions

**Series addition operator** → We can concat two or more different Pandas Series with the + operator.

Pandas will take values for the same index and add them together, for indexes that are not overlapping, Pandas will populate them with NaN values. This is because Pandas doesn't know how to add missing value with existing values.

```
In [21]: first_series = pd.Series({'Japan': 100, 'China': 450, 'USA': 250})
second_series = pd.Series({'Brazil': 70, 'China': 350, 'USA': 200})

concat_series = first_series + second_series
print(concat_series)
```

```
Brazil    NaN
China     800.0
Japan     NaN
USA       450.0
dtype: float64
```

We are using the + operator for concat 2 different Series together

For indexes that are not overlapping between the Series, Pandas will give NaN as the value



# Pandas Series Functions


**Series.add()** → The add() method allow us to concatenate between Series as well, we can also pass **fill\_value** parameter that will determine what should be the default value Pandas will populate non overlapping indexes instead of NaN.

```
In [22]: first_series = pd.Series({'Japan': 100, 'China': 450, 'USA': 250})
second_series = pd.Series({'Brazil': 70, 'China': 350, 'USA': 200})

concat_series = first_series.add(second_series, fill_value=0)

print(concat_series)

Brazil    70.0
China    800.0
Japan    100.0
USA     450.0
dtype: float64
```



Because we passed to the add() method the fill\_value parameter with 0 value, Pandas can now successfully add missing values together so we won't see NaN on non overlapping indexes now

**dtype** → Return the type of the values in the Pandas Series.

**astype()** → Convert the type of the values in the Pandas Series to the new type. If the convert can't be done Pandas will throw an error.

```
In [35]: print(concat_series.dtype)
print()
print(concat_series.astype(int))

float64

Brazil    70
China    800
Japan    100
USA     450
dtype: int64
```



# Class Exercise - Pandas Series

## Instructions:

Implement the following requirements using Pandas Series, Python and Jupyter:

- Create 3 Pandas Series stores that represent store stock inventory.  
Each store should have its corresponding items it sells and each item should have its corresponding price.  
For example - Tel-Aviv Store → Sells Computer → For price 1500
- Create at least 3 stores with at least 5 different items and prices
- Write a python function that get list of items to buy and return a dictionary contain 2 things:  
One → The total price the buyer will need to pay for the order  
Two → A dictionary contain each item and the option stores to buy it, if a requested item is not available in any store your series should put nan instead.



# Class Exercise Solution - Pandas Series

