Python

PROGRAMMING FOR EVERYONE

Eng. Omar Shaqra

Course objective

By the end of this course, You will be able to:

- Understand the role of Python libraries in data analysis and real-world applications
- Work efficiently with **NumPy** for numerical operations and array-based computing
- Use **Pandas** to load, clean, manipulate, and analyze structured data
- Create clear and meaningful visualizations using **Matplotlib**
- Build advanced and attractive statistical plots with **Seaborn**
- Apply these libraries together to solve real business and data problems
- Read, understand, and write data-related Python code with confidence

NumPy

1. The Foundation of Numerical Computing in Python
2. Essential for Data Science, Machine Learning, Scientific Computing & More
3. NumPy = Numerical Python
4. The most important library for numerical and scientific work in Python
5. Powers almost every major data & ML library

NumPy

Why do we use NumPy instead of regular Python lists?

- Much **faster** for large amounts of data
- Uses contiguous memory + fixed data types (no pointers like Python lists)
- Allows **vectorized** operations → no explicit loops needed
- Cleaner, more readable, and mathematically natural code

```python
# Pure Python lists (slow for big data)
a = [1, 2, 3, 4, 5]
b = [10, 20, 30, 40, 50]

print(a+b)
c = []
for i in range(len(a)):
    c.append(a[i] + b[i])
```

```python
# NumPy – clean & fast
import numpy as np

a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])

c = a + b            # No loop!
print(c)             # → [11 22 33 44 55]
```

NumPy

Installation

- NumPy is not part of standard Python
- pip install numpy / conda install numpy
- In Jupyter / Colab / many online environments → already installed

```
import numpy as np
print(np.__version__)          # example output: 1.26.4
```

```
import numpy as np

# Create array from a list
a = np.array([10, 20, 30, 40, 50])

print(a)                # [10 20 30 40 50]
print(type(a))          # <class 'numpy.ndarray'>
```

Creating Arrays & Understanding Shape

Creating NumPy Arrays – Most Common Methods

```
(function) def arange(
    start: _IntLike_co,
    stop: _IntLike_co,
    step: _IntLike_co = ...,
```

```
(function) def linspace(
    start: _ArrayLikeFloat_co,
    stop: _ArrayLikeFloat_co,
    num: SupportsIndex = ...,
```

```
# 1. From list
scores = np.array([85, 92, 78, 95])

# 2. From range-like
b = np.arange(10)          # 0, 1, 2, ..., 9

# 3. Zeros and ones (very useful!)
zeros = np.zeros(5)        # [0. 0. 0. 0. 0.]
ones  = np.ones(4)         # [1. 1. 1. 1.]

# 4. Simple sequence with steps
c = np.arange(0, 20, 5)    # [ 0  5 10 15]
```

```
# better control - number of points
y = np.linspace(0, 1, 11)      # [0.  0.1 0.2 ... 1. ]
```

Creating Arrays & Understanding Shape

Arrays filled with constants

```
zeros = np.zeros(6)                # 1D: [0. 0. 0. 0. 0. 0.]
zeros2d = np.zeros((3, 4))         # 3 rows × 4 columns

ones = np.ones((2, 5))             # 2 × 5 array of 1s

full = np.full((3, 3), 7)          # all elements = 7
```

Identity matrix & diagonal

```
I = np.eye(4)                      # 4×4 identity matrix
D = np.diag([10, 20, 30])          # diagonal matrix
```

Creating Arrays & Understanding Shape

Important Array Attributes

```
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

print(arr.shape)      # (3, 3)   → (rows, columns)
print(arr.ndim)       # 2        → number of dimensions
print(arr.size)       # 9        → total number of elements
print(arr.dtype)      # int64 (or int32 depending on system)
```

shape → (3 rows, 3 columns)
ndim → 2 (it's a 2-dimensional array / matrix)
size → 3 × 3 = 9 elements

```
#EX:
np.zeros(8)            # shape: (8,)    ndim: 1
np.full((4, 5), 99)    # shape: (4, 5)   ndim: 2
np.linspace(0, 5, 10) # shape: (10,)   ndim: 1
```

Indexing & Slicing in NumPy Arrays

Basic Indexing (like lists, but more powerful)

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50, 60, 70])

print(arr[0])      # 10
print(arr[3])      # 40
print(arr[-1])     # 70   (last element)
print(arr[-3])     # 50   (third from the end)
```

2D array example

```
mat = np.array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12]])

print(mat[0, 2])     # 3    → row 0, column 2
print(mat[2, 3])     # 12
print(mat[1, -1])    # 8     (last column of row 1)
```

Indexing & Slicing in NumPy Arrays

Syntax: array[start:stop:step]

```
a = np.arange(20)           # [ 0  1  2  ... 19]

print(a[2:8])               # [2 3 4 5 6 7]
print(a[5:])                # [ 5  6 ... 19]   (from index 5 to end)
print(a[:7])                # [0 1 2 3 4 5 6]  (from start to 6)
print(a[::2])               # [ 0  2  4  6 ... 18]  (every second element)
print(a[::-1])              # [19 18 ... 1  0]     (reverse the array)
```

2D slicing

```
print(mat)
# Rows 0 to 1 (inclusive), columns 1 to 3 (not including 3)
print(mat[0:2, 1:3])
# All rows, only columns 0 and 2
print(mat[:, [0, 2]])
# Last two rows, first three columns
print(mat[-2:, :3])
```

Indexing & Slicing in NumPy Arrays

Slicing creates a VIEW (very important!)

- Shallow Copy vs Deep Copy
  - **Shallow Copy:**
    - Copies the outer list only.
    - Inner objects (like nested lists) are shared.
  - **Deep Copy:**
    - Copies everything recursively.
    - Completely **independent** of the original.

```
TowDi = [[1,2],[3,4]]
x = TowDi.copy()
x[0][0]= 99
print(TowDi) # [[99, 2], [3, 4]]
```

```
print("view")
b = mat[1:3, 1:4]     # this is a view, not a copy
b[0, 0] = 99

print(mat)            # original matrix is changed!

c = mat[1:3, 1:4].copy()
c[0, 0] = 777         # does NOT affect mat
```

```
import copy

list1 = [[1, 2], [3, 4]]
deep = copy.deepcopy(list1)
deep[0][0] = 99

print(list1)  # [[1, 2], [3, 4]] 👉 not affected
```

Basic Operations & Broadcasting

Element-wise Operations

```python
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

print(a + b)        # [11 22 33 44]
print(a - b)        # [-9 -18 -27 -36]
print(a * b)        # [ 10  40  90 160]
print(a / b)        # [0.1 0.1 0.1 0.1]
print(a ** 2)       # [ 1  4  9 16]
print(a % 3)        # [1 2 0 1]

print(a + 100)      # [101 102 103 104]
print(a * 5)        # [ 5 10 15 20]
print(a > 2)        # [False False  True  True]
```

Basic Operations & Broadcasting

What is Broadcasting?

- NumPy automatically "stretches" smaller arrays so operations can be performed between arrays of different shapes — without copying data.

Broadcasting Rules

- Dimensions are compared from the right (trailing dimensions)
- Two dimensions are compatible when:
  - they are **equal**, or
  - one of them is **1**

Fail Example

```
A.shape = (3, 4)
B.shape = (2, 4)
```

```python
# Example 1: scalar + array
a = np.array([[1, 2, 3],
              [4, 5, 6]])
print(a + 10)
# [[11 12 13]
#  [14 15 16]]

# Example 2: row vector + matrix
row = np.array([100, 200, 300])
print(a + row)
# [[101 202 303]
#  [104 205 306]]

# Example 3: column vector
col = np.array([[1000],
                [2000]])
print(a + col)
# [[1001 1002 1003]
#  [2004 2005 2006]]
```

Basic Operations & Broadcasting

Common Broadcasting Patterns

```python
# Normalize rows (subtract mean of each row)
X = np.random.rand(5, 4)
X_centered = X - X.mean(axis=1, keepdims=True)

# Add different value to each column
adds = np.array([10, 20, 30, 40])
X_adjusted = X + adds
```

```python
# مصروف 3 أيام (صفوف) × 4 أنواع (أعمدة)
# أكل ، قهوة ، مواصلات ، ترفيه
X = np.array([
    [50, 10, 20, 20],
    [80, 20, 30, 10],
    [40, 10, 20, 30]
])
print("البيانات الأصلية:")

row_means = X.mean(axis=1, keepdims=True)
    # axis=0 → go down rows (column-wise)
    # axis=1 → go across columns (row-wise)
    # keepdims=True  keep it 2D
print("\nمتوسط كل يوم:")
X_centered = X - row_means
print("\nالصرف بالنسبة لمتوسط اليوم:")
print(X_centered)
```

Statistical & Aggregation Functions

Most Important Aggregation Methods

```
import numpy as np

data = np.array([23, 45, 12, 67, 89, 34, 56, 78, 91,

print(data.sum())          # total = 505
print(data.mean())         # average ≈ 50.5
print(data.median())       # 50.5
print(data.std())          # standard deviation
print(data.var())          # variance
print(data.min())          # 10
print(data.max())          # 91
print(data.argmin())       # index of minimum → 9
print(data.argmax())       # index of maximum → 8
```

Operation	axis=0 means	axis=1 means
sum / mean / max	per column	per row
Useful for	summarizing features	summarizing samples

Working with axis

```
# 2D example
scores = np.array([
    [85, 92, 78, 88],     # student 1
    [64, 70, 82, 91],     # student 2
    [95, 88, 76, 93]      # student 3
])

print(scores)

# axis=0 → along columns (per subject)
print("Mean per subject:", scores.mean(axis=0))
# [81.333 83.333 78.666 90.666]

# axis=1 → along rows (per student)
print("Mean per student:", scores.mean(axis=1))
# [85.75  76.75  88.  ]

print("Total per student:", scores.sum(axis=1))
print("Best score in each subject:", scores.max(axis=0))
```

Statistical & Aggregation Functions

```
[10, 12, 23, 34, 45, 56, 67, 78, 89, 91]
```

- Number of elements = 10 → even number → median = average of 5th and 6th elements:

$$\text{median} = \frac{45 + 56}{2} = 50.5$$

```
print(data.std())  # standard deviation
```

- **Standard deviation** measures how spread out the numbers are from the mean.
- Formula:

$$\text{std} = \sqrt{\frac{\sum (x_i - \text{mean})^2}{N}}$$

- Gives an idea of **how much numbers deviate from 50.5**.
- ✅ Output: a number around `28.5` (depends on exact calculation).

```
print(data.var())  # variance
```

- **Variance** = standard deviation squared.
- Formula:

$$\text{var} = \frac{\sum (x_i - \text{mean})^2}{N}$$

- It's literally the **average squared deviation from the mean**.
- ✅ Output: around `812.25` (because std ≈ 28.5 → 28.5^2 ≈ 812).

Normalization vs Standardization

Normalization (Min-Max Scaling)

- Transforms data to a **fixed range**, usually [0,1] or [-1,1].

```
# Normalize to [0,1]
normalized = (data - data.min()) / (data.max() - data.min())
```

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

Example:

Original	Normalized
10	0
20	0.25
30	0.5
40	0.75
50	1

Normalization vs Standardization

Standardization (Z-score Scaling)

- Centers data to **mean = 0** and **standard deviation = 1**.

```
# Standardize (z-score)
z_scores = (data - data.mean()) / data.std()
```

$$X_{\text{std}} = \frac{X - \text{mean}}{\text{std}}$$

Example:

Original	Standardized
10	-1.26
20	-0.63
30	0
40	0.63
50	1.26

Reshaping, Transposing & Flattening

Changing the Shape of Arrays Reshape • Transpose • Flatten

1. reshape()

```
import numpy as np

a = np.arange(12)          # [ 0  1  2  3  4  5  6  7  8  9 10 11]

print(a.reshape(3, 4))
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

print(a.reshape(2, -1))    # -1 يعني "احسبها أنت"
# [[ 0  1  2  3  4  5]
#  [ 6  7  8  9 10 11]]

print(a.reshape(3, 2, 2))
# 3 blocks, each 2×2
```

Note : reshape return a view not a copy
It affects the original data

Reshaping, Transposing & Flattening

2. Transpose
   - Swap rows and columns

```
mat = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

print(mat.T)
# or mat.transpose()
# [[1 4 7]
#  [2 5 8]
#  [3 6 9]]
```

3. Flattening
   - Convert multi-dimensional array to 1D

```
print(mat.flatten())      # always returns a copy
# [1 2 3 4 5 6 7 8 9]

print(mat.ravel())        # usually returns a view (faster)
# [1 2 3 4 5 6 7 8 9]

print(mat.reshape(-1))    # another common way
```

Boolean Indexing & Fancy Indexing

Selecting Data with Conditions

1. Boolean Indexing

- Use a boolean mask (True/False array) to select elements

```
import numpy as np

scores = np.array([85, 92, 67, 45, 98, 73, 88, 55])

# Create a boolean mask
mask = scores >= 80
print(mask)
# [ True  True False False  True False  True False]

# Use the mask to filter
print(scores[mask])
# [85 92 98 88]

# One-liner (very common pattern)
print(scores[scores >= 80])
# [85 92 98 88]
```

Modifying with boolean indexing

```
# Replace low scores with 50
scores[scores < 60] = 50
print(scores)
# [85 92 67 50 98 73 88 50]
```

2D example

```
data = np.array([
    [23, 45, 12, 67],
    [89, 34, 56, 78],
    [91, 10, 33, 82]
])

# All values > 70
print(data[data > 70])
# [89 78 91 82]

# Rows where the first column > 50
print(data[data[:, 0] > 50])
# [[89 34 56 78]
#  [91 10 33 82]]
```

Boolean Indexing & Fancy Indexing

2. Fancy Indexing

- Select elements using arrays of indices (integer arrays)

```
arr = np.arange(20) * 10
# [  0  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150 160 170 180 190]

# Select specific positions
indices = [2, 5, 8, 11]
print(arr[indices])
# [ 20  50  80 110]

# Can also use in 2D
mat = np.arange(20).reshape(4, 5)
print(mat)

rows = [0, 2, 3]
cols = [1, 3, 4]
print(mat[rows, cols])
# [ 1 13 19 ]
```

Random Numbers & Random Sampling

1. Setting the Random Seed

```python
import numpy as np

np.random.seed(42)          # same seed → same random numbers every time
```

2. Most Commonly Used Random Functions

```python
# Uniform random numbers [0.0, 1.0)
print(np.random.rand(5))          # 1D array of 5 numbers
print(np.random.rand(3, 4))       # 3×4 array

# Standard normal (Gaussian) distribution mean=0, std=1
print(np.random.randn(6))
print(np.random.randn(4, 3))

# Random integers (inclusive low, exclusive high)
print(np.random.randint(1, 100, size=10))      # 10 numbers between 1 and 99
print(np.random.randint(0, 10, size=(3,5)))    # 3×5 matrix
```

Random Numbers & Random Sampling

3. Other Useful Random Functions

```
# Random choice from a list/array
names = ['Ali', 'Sara', 'Omar', 'Lina', 'Khaled']
print(np.random.choice(names, size=3, replace=True))      # with replacement
print(np.random.choice(names, size=3, replace=False))     # without replacement

# Shuffle an array in place
arr = np.arange(10)
np.random.shuffle(arr)
print(arr)          # order is randomly changed

# Random permutation (returns new array)
perm = np.random.permutation(10)
print(perm)
```

lab

1. Create a 1D NumPy array containing the numbers from 10 to 50 (inclusive) with a step of 5.
2. Create a 4×6 array filled with zeros, then fill its third row (index 2) with the value 99.
3. Given the array:

   `arr = np.array([12, 45, 7, 19, 88, 3, 56, 91, 24, 67])`

   Print all elements that are greater than 50.
4. Create a 5×5 identity matrix, then change its main diagonal (from top-left to bottom-right) to the values [10, 20, 30, 40, 50].
5. Let `x = np.arange(1, 21).reshape(4, 5)`

   Extract the following using slicing:

   a. The last two rows

   b. The third column

   c. The 3×3 submatrix in the top-right corner
6. Given two arrays:

   `a = np.array([1, 2, 3, 4])` `b = np.array([[10], [20], [30]])`

   Compute `a + b` and explain the shape of the result.
7. Normalize the following array to the range [0, 1]:

   `data = np.array([150, 220, 90, 300, 180, 45])`

lab

8. Standardize (z-score) the same array from question 7.

9. Given the 2D array:

```python
mat = np.array([
    [4,  8,  1, 12],
    [7,  3,  9,  5],
    [11, 2,  6, 10],
    [15, 0, 14, 13]
])
```

   a. Compute the mean of each row

   b. Compute the sum of each column

   c. Find the maximum value in the entire matrix and its position (row, column)

10. Reshape the array `np.arange(36)` into a 3D array of shape (3, 4, 3).

11. Given `arr = np.arange(1, 17).reshape(4, 4)`

    Reverse the order of rows and columns (flip both).

lab

12. Given:

    `ages = np.array([23, 45, 19, 34, 28, 51, 17, 39])` `names = np.array(["Ali", "Sara", "Omar", "Lina", "Khaled", "Nour", "Yara", "Hassan"])`

    Get the names of people whose age is between 25 and 40 (inclusive).

13. Using the same `ages` and `names` arrays:

    Replace all ages less than 20 with 20.

14. Create a 6×6 array filled with random integers between 1 and 100 (inclusive).

    Then set the seed to 123 first.

15. Create an array of 10 random numbers from a normal distribution with mean 100 and standard deviation 15.

16. From the list `colors = ["red", "blue", "green", "yellow", "purple"]`

    Randomly select 3 different colors without replacement.

17. Shuffle the numbers from 1 to 20 in place.