Chapter V

Introduction to Python Libraries

1. Introduction

One of the major strengths of Python is the large number of scientific libraries that extend its capabilities. A library can be described as a collection of pre-written tools and functions that simplify the implementation of complex tasks.

Among the many libraries available in Python, two are particularly important for scientific and mathematical applications:

- **NumPy**: provides efficient tools for handling numerical data and performing operations on arrays, which are closely related to vectors and matrices in mathematics.
- **Matplotlib**: offers powerful capabilities for data visualization, allowing users to represent information through graphs and plots.

For students of mathematics, these libraries are especially valuable because they facilitate the implementation of vector and matrix computations while also enabling the graphical representation of mathematical functions and datasets. Together, they form an essential foundation for scientific computation using Python.

2. NumPy Library:

In Python we have lists that serve the purpose of arrays, but they are slow to process. NumPy acronym of "Numerical Python", is a one of the core libraries used for scientific computing in Python. It provides powerful tools to work with arrays, which are data structures designed to store and manipulate numerical data in a structured way, NumPy aims to provides an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called **ndarray** (n-dimensional array: 1-D Array = Vector, 2-D Array = Matrix …etc.), it provides a lot of functions that make morking with **ndarray** very easy.

In mathematics, arrays are closely related to vectors and matrices. Using NumPy, it becomes possible to perform operations on entire arrays simultaneously rather than element. This approach, known as vectorized computation, makes numerical calculations more efficient and closely aligned with the way mathematical operations on vectors and matrices are typically expressed.

2.1. Creating Arrays:

The central object in NumPy is the ndarray (n-dimensional array).

To use NumPy, we first import the library (Usually imported under the **np** alias):

```
import numpy as np
```

2.1.1. Empty Array: You can use ***np.empty()*** to create array with uninitialized values.

- **Empty 1-D Array:** is similar to a vector.

```
import numpy as np

array_1d = np.empty(5)
print(type(array_1d))
print(array_1d)
print(array_1d.ndim)
```

```
<class 'numpy.ndarray'>
[4.79425883e-288 6.87047687e-320 0.00000000e+000 1.14809090e-056
 0.00000000e+000]
1
```

So, this array contains 5 cells: [?, ?, ?, ?, ?], but these cells are not initialized, they contain random values already present in the memory.
NumPy Arrays provides the **ndim** attribute that returns the dimension of the array:

- **Empty 2-D Array:** Correspond to a matrix.

```
import numpy as np

array_2d = np.empty((3, 4))
print(type(array_2d))
print(array_2d)
```

```
<class 'numpy.ndarray'>
[[4.9e-324 9.9e-324 1.5e-323 2.0e-323]
 [2.5e-323 3.0e-323 4.9e-324 9.9e-324]
 [1.5e-323 2.0e-323 2.5e-323 3.0e-323]]
```

$$A = \begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

In this case, (3,4) means 3 rows and 4 columns.

- **Empty 3-D Array:** Can be seen as a stack of matrices.

```
import numpy as np

array_3d = np.empty((2, 3, 4))
print(type(array_3d))
print(array_3d)
```

```
<class 'numpy.ndarray'>
[[[6.23042070e-307 4.67296746e-307 1.69121096e-306 6.23043429e-307]
  [8.45593934e-307 7.56593017e-307 1.11258854e-306 1.11261502e-306]
  [1.42410839e-306 7.56597770e-307 6.23059726e-307 1.42419530e-306]]

 [[6.89805151e-307 7.56592338e-307 6.89807188e-307 7.56589622e-307]
  [2.22520559e-306 1.11261298e-306 9.79107193e-307 1.42418172e-306]
  [1.37961641e-306 9.45699679e-308 2.22522596e-306 1.42410974e-306]]]
```

In this case, (2,3,4) means : 2 matrices, each matrix has 3 rows and 4 columns.

2.1.2. From a list

- **1-D Array from a list:**

```
import numpy as np

list_1d = [1, 2, 3, 4, 5]
array_1d = np.array(list_1d)

print(array_1d)
```

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

- **2-D Array from a list:**

```
list_2d = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

array_2d = np.array(list_2d)

print(array_2d)
```

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

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

- **3-D Array from a list:**

```
list_3d = [
    [
        [1, 2],
        [3, 4]
    ],
    [
        [5, 6],
        [7, 8]
    ]
]

array_3d = np.array(list_3d)

print(array_3d)
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

2.1.3. With special functions

In **NumPy**, special functions allows us to create arrays quicly without manually writing lists.These functions are particularly in scientifc computing when working with vectors, matrices, or multidensional munerical data.

- **1-D Array:**

```
import numpy as np

# ----- 1D Arrays -----

# 1D array of zeros
array1_zeros = np.zeros(5)

# 1D array of ones
array1_ones = np.ones(5)

# 1D array using arange
array1_arange = np.arange(0, 12, 3)

print("1D Arrays:")
print("Zeros:", array1_zeros)
print("Ones :", array1_ones)
print("Arange:", array1_arange)
```

```
1D Arrays:
Zeros: [0. 0. 0. 0. 0.]
Ones : [1. 1. 1. 1. 1.]
Arange: [0 3 6 9]
```

- **2-D Array:**

```
import numpy as np

# ----- 2D Arrays -----

# 2D array of zeros (3x4 matrix)
array2_zeros = np.zeros((3,4))

# 2D array of ones (3x4 matrix)
array2_ones = np.ones((3,4))

# 2D array using arange and reshape
array2_arange = np.arange(12).reshape(3,4)

print("\n2D Arrays:")
print("Zeros:\n", array2_zeros)
print("Ones:\n", array2_ones)
print("Arange:\n", array2_arange)
```

```
2D Arrays:
Zeros:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
Ones:
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
Arange:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

- **3-D Array:**

```
import numpy as np

# ----- 3D Arrays -----

# 3D array of zeros (2 matrices of 3x4)
array3_zeros = np.zeros((2,3,4))

# 3D array of ones (2 matrices of 3x4)
array3_ones = np.ones((2,3,4))

# 3D array using arange and reshape
array3_arange = np.arange(24).reshape(2,3,4)

print("\n3D Arrays:")
print("Zeros:\n", array3_zeros)
print("Ones:\n", array3_ones)
print("Arange:\n", array3_arange)
```

```
3D Arrays:
Zeros:
 [[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
Ones:
 [[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]]
Arange:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

2.2. Basic Array Operations

There is three basic NumPy array operations: Addition, Multiplication and scalar operations.

2.2.1. Basic operations with 1-D Arrays (Vectors)

```
import numpy as np

# Create two 1-D arrays (vectors)
A = np.array([1, 2, 3, 4])
B = np.array([5, 6, 7, 8])

# Addition
C = A + B

# Element-wise multiplication
D = A * B

# Scalar operations
E = 2 * A
F = A + 10

print("Vector A:", A)
print("Vector B:", B)

print("A + B =", C)
print("A * B =", D)

print("2 * A =", E)
print("A + 10 =", F)
```

```
Vector A: [1 2 3 4]
Vector B: [5 6 7 8]
A + B = [ 6  8 10 12]
A * B = [ 5 12 21 32]
2 * A = [2 4 6 8]
A + 10 = [11 12 13 14]
```

2.2.2. Basic operations with 2-D Arrays (Matrices)

```python
import numpy as np

# Create two 2-D arrays (matrices)
A = np.array([[1, 2, 3],
              [4, 5, 6]])

B = np.array([[6, 5, 4],
              [3, 2, 1]])

# Addition
C = A + B

# Element-wise multiplication
D = A * B

# Scalar operations
E = 3 * A
F = A + 5

print("Matrix A:\n", A)
print("Matrix B:\n", B)

print("A + B:\n", C)
print("A * B:\n", D)

print("3 * A:\n", E)
print("A + 5:\n", F)
```

```
Matrix A:
 [[1 2 3]
 [4 5 6]]
Matrix B:
 [[6 5 4]
 [3 2 1]]
A + B:
 [[7 7 7]
 [7 7 7]]
A * B:
 [[ 6 10 12]
 [12 10  6]]
3 * A:
 [[ 3  6  9]
 [12 15 18]]
A + 5:
 [[ 6  7  8]
 [ 9 10 11]]
```

2.2.3. Basic operations with 3-D Arrays

```python
import numpy as np

# Create two 3-D arrays
A = np.array([[[1, 2], [3, 4]],
              [[5, 6], [7, 8]]])

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

# Addition
C = A + B

# Element-wise multiplication
D = A * B

# Scalar operations
E = 2 * A
F = A + 3

print("Array A:\n", A)
print("Array B:\n", B)

print("A + B:\n", C)
print("A * B:\n", D)

print("2 * A:\n", E)
print("A + 3:\n", F)
```

```
Array A:
 [[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
Array B:
 [[[8 7]
  [6 5]]

 [[4 3]
  [2 1]]]
A + B:
 [[[9 9]
  [9 9]]

 [[9 9]
  [9 9]]]
A * B:
 [[[ 8 14]
  [18 20]]

 [[20 18]
  [14  8]]]
2 * A:
 [[[ 2  4]
  [ 6  8]]

 [[10 12]
  [14 16]]]
A + 3:
 [[[ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]]]
```

3.Matplot Library

Matplotlib is a low-level graph plotting library in python that serves as a visualization utility. It allows you to create graphs, plots, and charts from data so that you can see patterns, trends or relationships. In mathematics, it's like turning numbers into visual shapes – curves, points, or bars, so that understanding functions or datasets becomes much easier.

The most used submodule of Matplot Library is **pyplot**, and it is usually imported under the **plt** alias:

```
import matplotlib.pyplot as plt
```

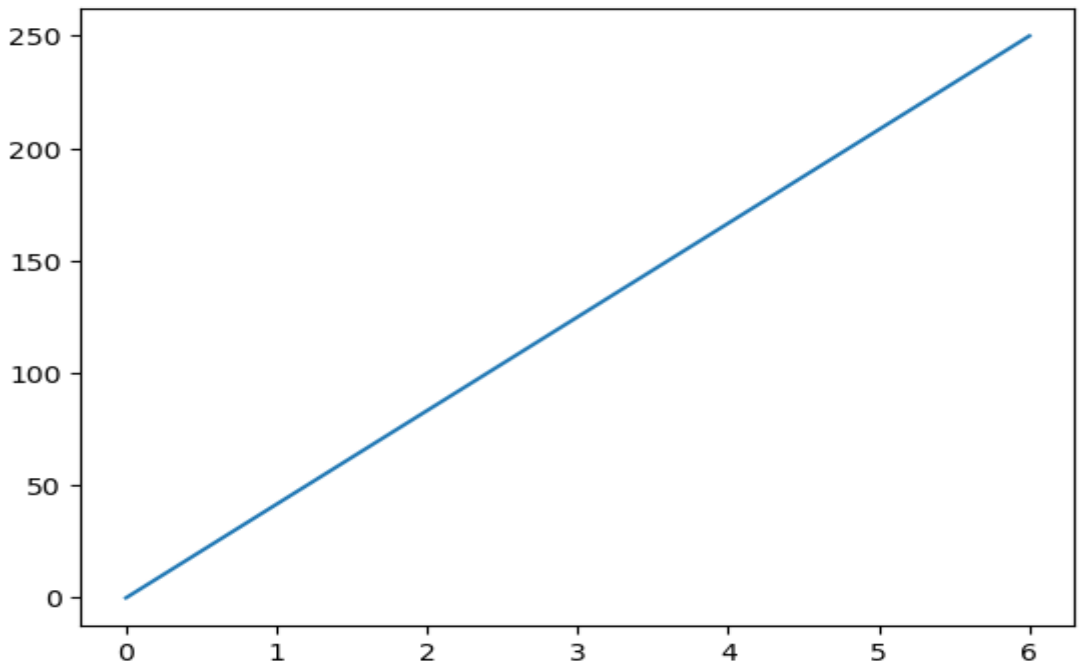
3.1. Line Plot (Function Curve)

As example, if we want to draw a line using two points A(0,0) and B(6,250); we notice that we have two axes: The X-axis [0,6] and the Coordinates axis [0, 250].

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])
print(xpoints)
print(ypoints)
plt.plot(xpoints, ypoints)
plt.show()
```

```
[0 6]
[  0 250]
```

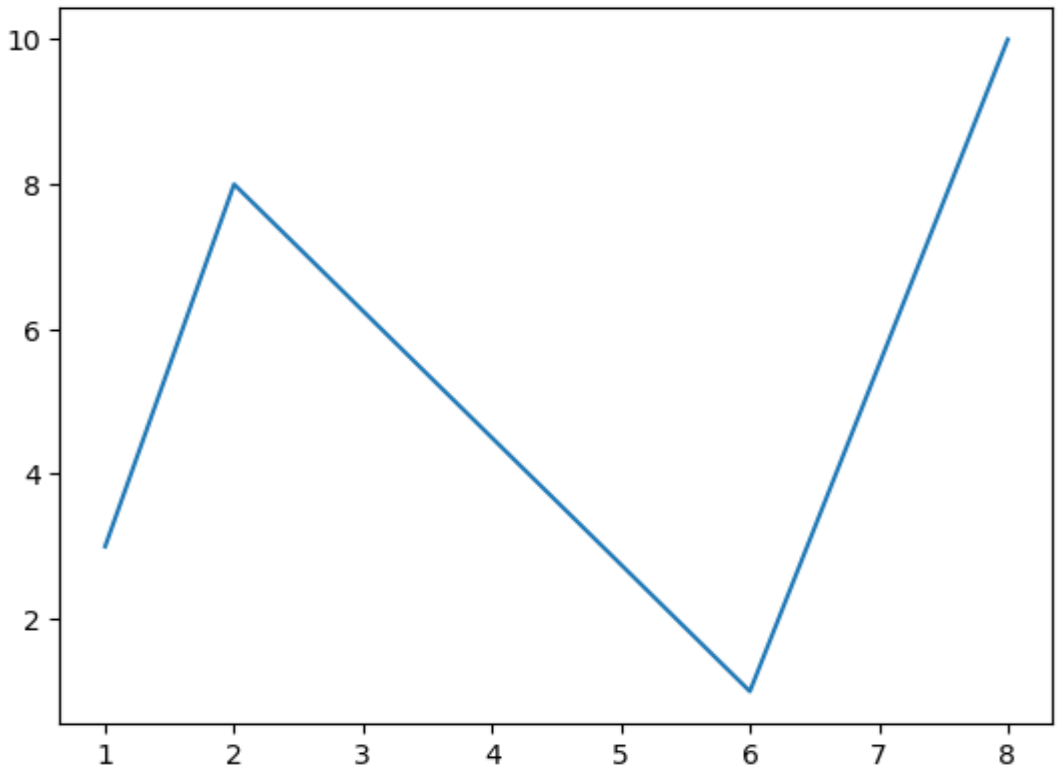


If we have many points, and we want to draw a line from the first point (1,3) to the second point (2,8), then to the third point (6,1)and finally to the fourth point (8,10), we have to do:

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```



We can use the keyword argument ***marker*** to emphasize each point with a specified marker:

```
plt.plot(xpoints, ypoints, marker = 'o')
```

You can use : o / + / * or any other symbol.

You can also personalize other parameters as line and color : *marker|line|color*

- Line: '-' : solid line, ':' : Dotted line, '-.' : Dashed/Dotted line
- Color r: red, b: blue, g: green ….

There are many other parameters that you can use like: marker size : ms, markeredgecolor : mec, markerfacecolor : mfc,

❖ **Create Labels for a plot:**

The sub-library Pyplot, offer two functions ***xlabel()*** and ***ylabel()*** to set labels for x- and y-axis, and another function to set a title.

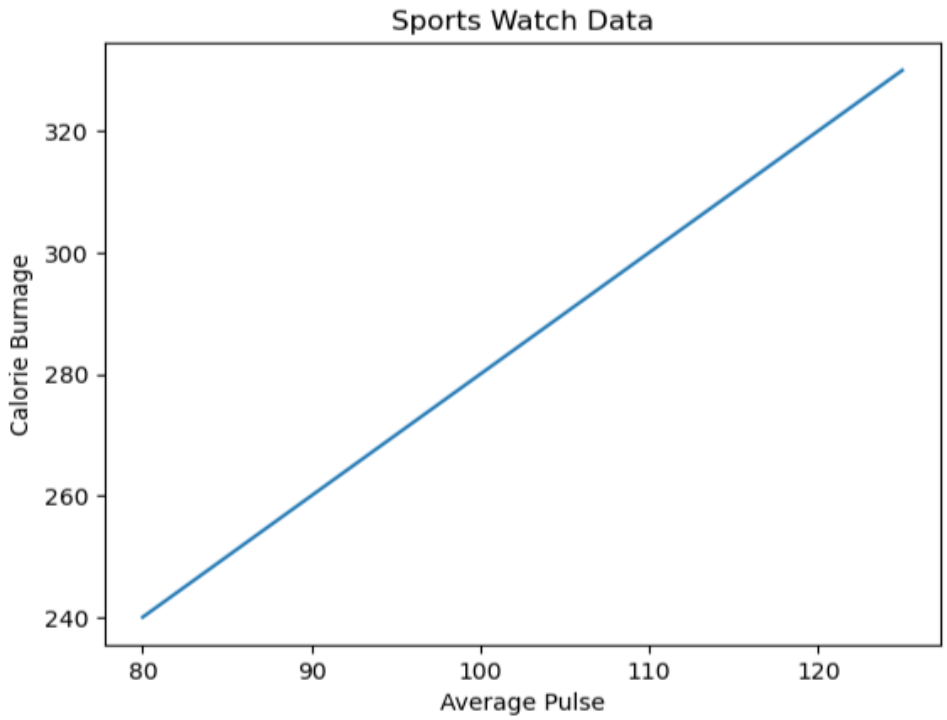
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```



3.2. Scatter Plot (Points)

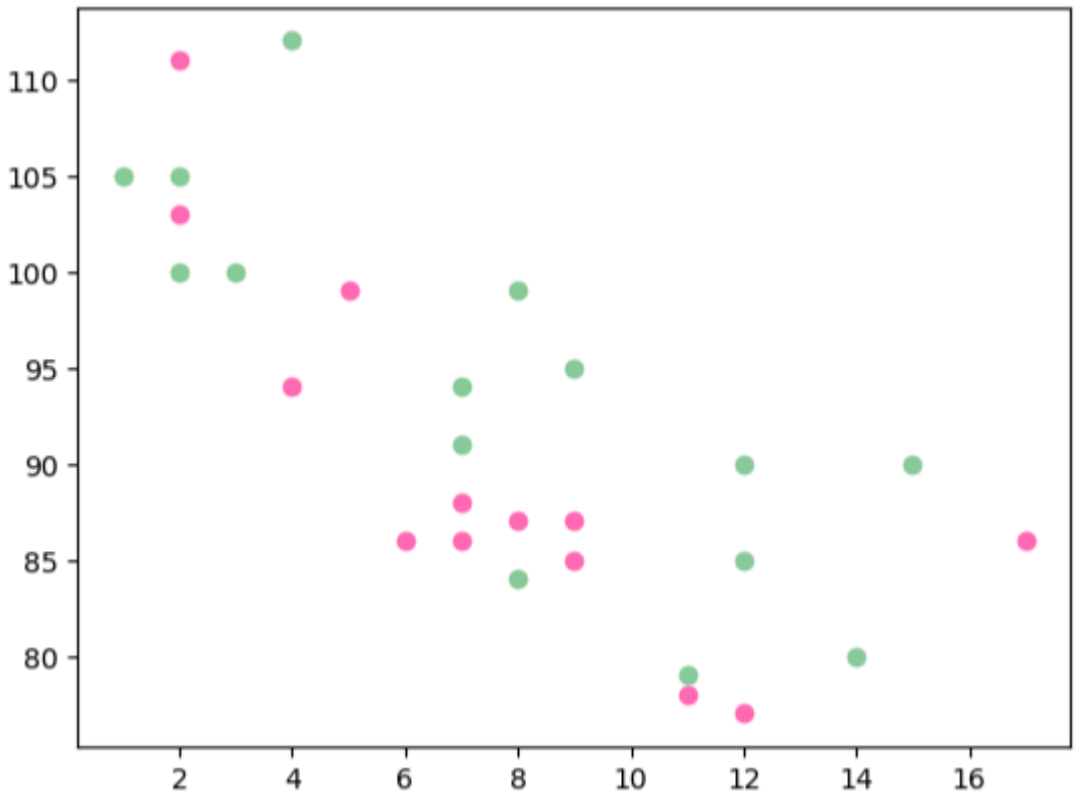
The scatter() function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y, color = 'hotpink')

A = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
B = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(A, B, color = '#88c999')

plt.show()
```



3.3. Histogram

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate random data
data = np.random.randn(1000)

# Plot histogram
plt.hist(data, bins=20, color="skyblue")

plt.title("Histogram of Random Data")
plt.xlabel("Value")
plt.ylabel("Frequency")

plt.show()
```