1.Getting Familiar with NumPy

NumPy (Numerical Python) is a foundational library in the Python ecosystem that enables efficient numerical computations and handling of large datasets. Its core functionality revolves around the ndarray object, which is a powerful, N-dimensional array that supports a wide range of mathematical operations. Unlike traditional Python lists, NumPy arrays are more memory-efficient and allow for fast, vectorized operations without the need for explicit loops. This makes NumPy an essential tool for scientific computing, data analysis, and machine learning.

install numpy:

In [35]:
```
# import numpy package
import numpy as np
```

creation of arrays:

Array Creation: Understanding how to create arrays is fundamental. Arrays can be created from lists or tuples using np.array(), or initialized with default values like zeros, ones, or sequences.

In [2]:
```
#creation of array from a list
array = np.array([0,2,4,6,8])
array
```

Out[2]:
```
array([0, 2, 4, 6, 8])
```

In [3]:
```
#creation of arrays using built-in functions
np.zeros((2,2)) #creation of an array filled with zeros
```

Out[3]:
```
array([[0., 0.],
       [0., 0.]])
```

In [4]:
```
np.ones((3,4)) #creation of an array filled with ones
```

Out[4]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [5]:
```
np.arange(0,30,3) #creates an array with evenly spaced values within a specified range
```

Out[5]:
```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])
```

In [6]:
```
np.empty((2,2))
```

Out[6]:
```
array([[0., 0.],
       [0., 0.]])
```

In [7]:
```
np.random.randn(3,3) #array with random values
```

Out[7]:
```
array([[ 0.25033476,  0.84031634, -0.31005456],
       [ 0.23435519, -0.75849843,  0.00503297],
       [ 0.52051578, -0.18185237,  1.57077125]])
```

In [8]:
```
np.random.rand(3,3) #array with random values between 0 and 1
```

Out[8]:
```
array([[0.43746843, 0.6061039 , 0.88101444],
       [0.673657  , 0.6813042 , 0.76539053],
       [0.45493033, 0.15423337, 0.77726153]])
```

Basic operations of numpy:

NumPy allows for element-wise operations, such as addition, subtraction, multiplication, and division, to be performed directly on arrays.

In [9]:
```
#addition of arrays
array1 = np.array([1,3,5,7,9])
array+array1
```

Out[9]:
```
array([ 1,  5,  9, 13, 17])
```

In [10]:
```
#multiplication of arrays
array*array1
```

Out[10]:
```
array([ 0,  6, 20, 42, 72])
```

In [11]:
```
#subtraction of arrasy
array1-array
```

Out[11]:
```
array([1, 1, 1, 1, 1])
```

In [12]:
```
#division
array/array1
```

Out[12]:
```
array([0.        , 0.66666667, 0.8       , 0.85714286, 0.88888889])
```

In [13]:
```
#finding square root of array numbers
array**0.5
```

Out[13]:
```
array([0.        , 1.41421356, 2.        , 2.44948974, 2.82842712])
```

Understanding array properties:

Important properties like shape, size, dtype, and ndim provide insight into the array's structure, dimensions, and data type, which is crucial for effective data manipulation.

In [14]:
```
array1.shape #indicates the size of each dimension
```

Out[14]:
```
(5,)
```

In [15]:
```
array.ndim #no.of dimensions in the array
```

Out[15]:
```
1
```

In [16]:
```
array.size #total no.of elements in the array
```

Out[16]:
```
5
```

In [17]:
```
array.dtype #indicates the datatype of the elements in the array
```

Out[17]:
```
dtype('int32')
```

2.DATA MANIPULATION:

Data manipulation involves transforming and preparing data for analysis or modeling. With NumPy, this includes operations such as indexing, slicing, reshaping, and applying mathematical transformations to arrays. These capabilities allow you to efficiently handle and modify datasets, which is a common requirement in data preprocessing tasks. NumPy's ability to easily access and manipulate specific elements or subsets of data makes it an invaluable tool for data scientists and analysts.

In [18]:
```
#array creation
arr = np.array([[1,2,3],[4,5,6]])
arr
```

Out[18]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Indexing and Slicing:

NumPy allows for precise selection of elements or subarrays, enabling targeted data manipulation. Indexing can retrieve individual elements, while slicing can extract subarrays.

In [19]:
```
#indexing
print("element at (1,1) is",arr[1,1])
```

```
element at (1,1) is 5
```

In [20]:
```
#slicing
print("row 1 elements are:",arr[:1])
```

```
row 1 elements are: [[1 2 3]]
```

Reshaping:

The reshape() function changes the shape of an array without altering its data, which is useful for preparing data for specific algorithms or visualizations.

In [21]:
```
#reshaping
arr.reshape(6,1)
```

```
Out[21]: array([[1],
                [2],
                [3],
                [4],
                [5],
                [6]])
```

3.DATA AGGREGATION

Data aggregation involves summarizing data to extract meaningful insights. NumPy provides various functions to compute summary statistics, such as mean, median, standard deviation, and sum. These functions help in understanding the central tendencies, variability, and overall distribution of your data. Grouping data based on certain criteria and then performing aggregation is a common task in data analysis.

```
In [22]: data = np.array([1,3,2,6,4,3,6,4,3,7])
         data
```

```
Out[22]: array([1, 3, 2, 6, 4, 3, 6, 4, 3, 7])
```

Summary Statistics:

Calculating metrics such as mean (average), median (middle value), standard deviation (measure of spread), and sum (total).

```
In [23]: # to find mean of the data
         mean = np.mean(data)
         mean
```

```
Out[23]: 3.9
```

```
In [24]: # To find the median of the data
         np.median(data)
```

```
Out[24]: 3.5
```

```
In [25]: # to find standard deviation of the data
         np.std(data)
```

```
Out[25]: 1.8138357147217055
```

```
In [26]: # sum of elements in the data
         np.sum(data)
```

```
Out[26]: 39
```

4.DATA ANALYSIS

Data analysis with NumPy focuses on deriving insights and patterns from data:

```
In [27]: array
```

```
Out[27]: array([0, 2, 4, 6, 8])
```

```
In [28]: array1
```

```
Out[28]: array([1, 3, 5, 7, 9])
```

```
In [29]: corr_coef = np.corrcoef(array,array1)[0,1]
         corr_coef
```

```
Out[29]: 0.9999999999999999
```

correlation:

Measuring the strength and direction of the relationship between two datasets using correlation coefficients.

```
In [30]: #correlation
         np.corrcoef(array,array1)[0,1]
```

```
Out[30]: 0.9999999999999999
```

```
In [31]: data
```

```
Out[31]: array([1, 3, 2, 6, 4, 3, 6, 4, 3, 7])
```

outliers:

Identifying values that significantly differ from the rest of the data, often by comparing them to statistical measures like mean and standard deviation.

```
In [32]: mean = np.mean(data)
std_dev = np.std(data)
z_scores = (data - mean) / std_dev
# Identify outliers (z-score > 3 or < -3)
outliers = data[np.abs(z_scores) > 3]
print(f"Number of outliers: {len(outliers)}")
```

```
Number of outliers: 0
```

percentiles

Determining the position of a value within a dataset relative to other values, useful for understanding data distribution.

```
In [33]: # Calculate the 25th, 50th (median), and 75th percentiles
percentiles = np.percentile(data, [25, 50, 75])
print(f"25th percentile: {percentiles[0]}")
print(f"50th percentile (median): {percentiles[1]}")
print(f"75th percentile: {percentiles[2]}")
```

```
25th percentile: 3.0
50th percentile (median): 3.5
75th percentile: 5.5
```

5. Application in Data Science

NumPy is a powerful tool for data science professionals due to its efficient handling of numerical data and operations. Here's how it benefits data science tasks:

1.Performance Efficiency:

Speed: NumPy operations are implemented in C, which makes them much faster than equivalent operations in pure Python. This is particularly important when dealing with large datasets or performing complex calculations. Vectorization: NumPy allows for vectorized operations, meaning computations can be applied across entire arrays without explicit loops. This speeds up the processing and makes the code cleaner and more efficient.

2.Memory Efficiency:

Storage: NumPy arrays consume less memory compared to traditional Python lists because they store data in contiguous blocks of memory and use a more efficient data type.

Data Handling: With NumPy, you can perform operations on large datasets without running into memory issues, thanks to its efficient data storage and manipulation techniques.

3.Advanced Mathematical Functions:

NumPy provides a wide range of mathematical functions and operations, including linear algebra, Fourier transforms, and random number generation. This makes it suitable for complex numerical analysis.

4.Integration with Other Libraries:

NumPy is often used in conjunction with other libraries like pandas for data manipulation, scikit-learn for machine learning, and TensorFlow or PyTorch for deep learning. This integration is seamless due to NumPy's array structure being a common standard in the Python data science ecosystem.

Advantages Over Traditional Python Data Structures

Arrays vs. Lists: Unlike Python lists, NumPy arrays have a fixed size and type, which allows for more efficient memory usage and faster computation. Lists are more flexible but less performant for numerical operations.

Element-wise Operations: NumPy supports element-wise operations, which are not natively supported by Python lists. This feature allows for concise and efficient mathematical computations on entire datasets.

Broadcasting: NumPy's broadcasting mechanism allows for operations on arrays of different shapes without the need for explicit replication of data, simplifying code and improving performance.

Real-World Examples

1.Machine Learning:

Data Preparation: NumPy is used for preprocessing data, such as scaling features, transforming datasets, and handling missing values. Libraries like scikit-learn often use NumPy arrays for their underlying data structures.

Algorithm Implementation: Many machine learning algorithms, such as linear regression, logistic regression, and neural networks, rely on matrix operations and linear algebra, which are efficiently handled by NumPy.

2.Financial Analysis:

Risk Metrics: NumPy helps calculate financial metrics such as volatility, Value at Risk (VaR), and portfolio returns. Efficient computation of these metrics is crucial for real-time financial analysis and decision-making.

Time Series Analysis: NumPy facilitates the handling of large financial datasets and performing time series analysis to forecast trends and make investment decisions.

3.Scientific Research:

Simulations: Scientists use NumPy for numerical simulations, such as solving differential equations and running experiments. Its ability to handle large matrices and perform complex calculations makes it ideal for research applications.

Data Analysis: Researchers analyze experimental data, perform statistical tests, and visualize results using NumPy. Its support for high-performance operations is essential for processing large datasets and deriving meaningful insights.

In [ ]: