## Getting familiar with Numpy:

NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing and handling large datasets.

## Exploring Numpy:

- Creating Arrays

One of the key features of NumPy is its N-dimensional array object, or ndarray,which is a fast, flexible container for large datasets in Python.

```
In [1]: import numpy as np
```

```
In [4]: a = np.array([1, 2, 3, 4])
        print(a)
```

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

- Creating 2D array

```
In [5]: b = np.array([[1, 2, 3], [4, 5, 6]])
        print(b)
```

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

## Properties of array:

- Shape:Returns a tuple representing the dimensions of the array.

```
In [6]: print(b.shape)
```

```
(2, 3)
```

- dtype: Returns the data type of the array's elements.

```
In [7]: print(a.dtype)
```

```
int32
```

- ndim: Returns the number of dimensions.

```
In [8]: print(b.ndim)
```

```
2
```

- size: Returns the total number of elements.

```
In [9]: print(b.size)
```

```
6
```

## Array Operations

NumPy supports a wide range of mathematical operations

- Element wise operations

```
In [11]: arr1= np.array([1, 3, 5, 7])
         arr2= np.array([2, 4, 6, 8])
         arr=arr1 + arr2
         print(arr)
```

```
[ 3  7 11 15]
```

```
In [13]: arr1= np.array([3,5,7,9])
         arr2= np.array([2, 4, 6, 8])
         arr=arr1 - arr2
         print(arr)
```

```
[1 1 1 1]
```

In [14]:
```
arr1= np.array([3,5,7,9])
arr2= np.array([2, 4, 6, 8])
arr=arr1*arr2
print(arr)
```

```
[ 6 20 42 72]
```

- Comparisons between arrays of the same size yield boolean arrays:

In [5]:
```
arr1 = np.array([[1.,3.,5.],[4.,7.,11.]])
arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
arr2 > arr1
```

Out[5]:
```
array([[False,  True, False],
       [ True, False,  True]])
```

## Data Manipulation

- Data manipulation in NumPy encompasses a range of techniques for reshaping, slicing, combining, and otherwise transforming arrays to suit specific needs.

1.Array Creation: We start by creating a simple 1D array using np.arange().

In [8]:
```
print("Array Creation:")
arr = np.arange(12)
print("Original array:")
print(arr)
```

```
Array Creation:
Original array:
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

2.Reshaping Arrays:Reshaping arrays in NumPy involves changing the shape of an array without modifying its data.

In [9]:
```
print("\nReshaping Arrays:")
reshaped_arr = arr.reshape((3, 4))
print("Reshaped array (3x4):")
print(reshaped_arr)
```

```
Reshaping Arrays:
Reshaped array (3x4):
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

3.Indexing

In [13]:
```
arr=np.arange(10)
arr[9]
```

Out[13]:
```
9
```

4.Slicing

In [15]:
```
print("\nSlicing:")
slice_arr = reshaped_arr[1:3, 1:4]
print(slice_arr)
```

```
Slicing:
[[ 5  6  7]
 [ 9 10 11]]
```

5.Flattened Array:The 2D array is then flattened back into a 1D array.

In [16]:
```
print("\nFlattening Arrays:")
flattened_arr = reshaped_arr.flatten()
print("Flattened array:")
print(flattened_arr)
```

```
Flattening Arrays:
Flattened array:
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

6.Transposing:

In [17]:
```
print("\nTransposing:")
transposed_arr = reshaped_arr.T
print("Transposed array:")
print(transposed_arr)
```

```
Transposing:
Transposed array:
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

7.Concatenation

In [19]:
```
print("\nConcatenation:")
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
concatenated_arr = np.concatenate((a, b))
print("Concatenated array:")
print(concatenated_arr)
```

```
Concatenation:
Concatenated array:
[1 2 3 4 5 6]
```

## Data Aggregation

Data aggregations are essential for summarizing, analyzing, and understanding datasets. They allow you to condense large amounts of data into meaningful statistics and insights.Data Aggregaation inccludes

1.Summarization of Data

2.Data Exploration and Insight

3.Data Cleaning and Validation

4.Data Reporting and Visualization

In [20]:
```
data = np.array([
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15]
])
print("Summary Statistics:")
mean = np.mean(data)
print("Mean:", mean)
median = np.median(data)
print("Median:", median)
std_dev = np.std(data)
print("Standard Deviation:", std_dev)
total_sum = np.sum(data)
print("Sum:", total_sum)
```

```
Summary Statistics:
Mean: 8.0
Median: 8.0
Standard Deviation: 4.320493798938574
Sum: 120
```

## Data Analysis

1. Finding Correlations

Correlation measures the strength and direction of the linear relationship between two variables.

In [2]:
```
data1 = np.array([10, 20, 30, 40, 50])
data2 = np.array([5, 15, 25, 35, 45])
correlation_matrix = np.corrcoef(data1, data2)
print("Correlation coefficient matrix:")
print(correlation_matrix)
```

```
Correlation coefficient matrix:
[[1. 1.]
 [1. 1.]]
```

2. Calculating Percentiles

Percentiles indicate the value below which a given percentage of observations fall.

In [3]:
```
data = np.array([10, 20, 30, 40, 50])
p30= np.percentile(data, 30)
```

```
p40= np.percentile(data, 40)
p60= np.percentile(data, 60)
print("30th Percentile:", p30)
print("40th Percentile (Median):", p40)
print("60th Percentile:", p60)
```

```
30th Percentile: 22.0
40th Percentile (Median): 26.0
60th Percentile: 34.0
```

- Numpy's efficiency in handling in datasets

NumPy is widely recognized for its efficiency in handling large datasets due to its optimized data structures and operations. Here are key reasons why NumPy excels in managing and processing large amounts of data:

1. Efficient Data Storage Contiguous Memory Allocation: NumPy arrays are stored in contiguous blocks of memory, which enhances cache efficiency and speeds up access times. This contrasts with Python lists, which can be scattered in memory and have more overhead.

Homogeneous Data Types: NumPy arrays require all elements to be of the same type, allowing for more efficient memory storage and processing compared to Python's heterogeneous lists.

2. Vectorized Operations Element-wise Operations: NumPy supports vectorized operations that perform operations on entire arrays without the need for explicit loops. This leverages low-level optimizations and accelerates computation significantly.

3. Optimized Mathematical Functions Built-in Functions: NumPy provides a wide range of mathematical functions that are implemented in C and are highly optimized. Operations like summing, mean calculation, and standard deviation are faster than custom implementations in pure Python.

4. Broadcasting Flexible Array Operations: Broadcasting allows NumPy to perform operations on arrays of different shapes by automatically expanding their dimensions. This eliminates the need for explicit loops and makes operations more memory-efficient.

5.Efficient Array Operations In-place Operations: NumPy operations can often be performed in-place, reducing the need for additional memory allocation.

## Application in Data Science

- Advantages of using Numpy over traditional Python data structures for numerical computations.

1.Performance and Speed: Through vectorized operations and efficient memory usage.

2.Homogeneous Data Types: For consistent and efficient data handling.

3.Advanced Mathematical Functions: Implemented in optimized C code.

4.Array Broadcasting: Facilitates operations on arrays of different shapes.

5.Efficient Data Manipulation: Advanced slicing, indexing, and array operations.

6.Memory Mapping: For handling large datasets efficiently.

7.Integration with Scientific Libraries: Seamless integration with a broader data science ecosystem.

8.Reduction of Boilerplate Code: Simplified and more concise code for complex operations.

- Real world examples

1. Machine Learning

- Data Preparation:

Feature Engineering: NumPy is used for manipulating and transforming features in datasets. For instance, you can use NumPy to normalize or standardize data, which is a common preprocessing step in machine learning pipelines.

Batch Processing: When training models, especially neural networks, data is often processed in batches. NumPy arrays facilitate efficient batch processing and data manipulation.

Example:

Deep Learning Frameworks

2. Financial Analysis a. Portfolio Management:

Risk Assessment: NumPy is used to compute various risk metrics, such as standard deviation, value-at-risk, and portfolio variance. For instance, calculating the covariance matrix of asset returns helps in understanding the risk and correlation between different assets. b. Time Series Analysis:

Trend Analysis: NumPy’s functions for rolling windows and moving averages are employed to analyze financial time series data, such as stock prices or economic indicators.

Example:

Algorithmic Trading

3. Scientific Research a. Simulation and Modeling:

Numerical Simulations: In fields like physics or biology, NumPy is used to run simulations and solve differential equations. For instance, simulating the behavior of particles in a fluid involves using NumPy arrays to handle large-scale numerical computations. b. Data Analysis:

Statistical Analysis: Researchers use NumPy to perform various statistical analyses, such as hypothesis testing or regression analysis. NumPy’s statistical functions help in analyzing experimental data and deriving meaningful insights.

Example:

Climate Modeling