1) Introduction of nympy in python:

- numpy is a powerfull library for numberical computing taht provided support for array,matrices and a wide range of mathematical function to operand on these data structure.
- It is a fundamental package for statistics computing in python
- import numpy as np
- It is used to perform different mathematical operation on N-Dimentional Array

In [2]:
```
import numpy as np
```

Features of numpy:

- It is time efficients.
- It has so many mathematical operations.
- To solve probability.
- Linear algebra.
- matrix related problem.
- Geometry problems.
- Random number generation.

In [3]:
```
# It has so many  mathematical operation:
np.log10(10),np.mean([18,20,34,8.67]),np.median([13,1,2,45,4]),np.sin(23),np.sqr
```

Out[3]:
```
(1.0, 20.1675, 4.0, -0.8462204041751706, 4.0)
```

* Role of NumPy in Data Science, Machine Learning, and Scientific Computing

Data Science:

NumPy simplifies data preprocessing and numerical analysis, enabling tasks like handling missing data, normalizing datasets, and computing statistical measures. Libraries like Pandas are built on NumPy, inheriting its powerful capabilities.

Machine Learning:

NumPy arrays form the foundation for manipulating tensors, which are central to machine learning frameworks like TensorFlow and PyTorch. It helps in implementing algorithms from scratch and understanding their underlying mathematics.

Scientific Computing:

Researchers use NumPy for simulations, modeling, and solving large-scale problems in physics, biology, chemistry, and more. Its tools for Fourier transforms, linear algebra, and

random number generation make it ideal for scientific research.

2) Arrays in NumPy

- A numpy array is homogeneous data typr array which means it of data a one particulaer kind of data type to an entire array
- The ndarray (short for N-dimensional array) is a container for:
- Homogeneous data (all elements must be of the same data type, such as integers, floats, or strings).
- Multi-dimensional structures (1D, 2D, 3D, or more).
- Fast, vectorized operations, eliminating the need for Python loops in most cases.

In [4]:
```
import numpy as np
array=np.array(0)
print(array)
type(array)
```

```
0
```

Out[4]: numpy.ndarray

In [5]:
```
# It is a homogeneous data type
a=np.array(['ram',10])
print(a)
```

```
['ram' '10']
```

* Creating arrays

- NumPy provides several ways to create arrays with different values and structures. Let's explore some of the most commonly used methods:

In [6]:
```
# 1. Creating Arrays with np.array()
# You can create an array directly from a Python list or tuple using np.array().

# 1D array
array_1d = np.array([1, 2, 3, 4, 5])
print("1D Array:", array_1d)

# 2D array
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:\n", array_2d)
```

```
1D Array: [1 2 3 4 5]
2D Array:
 [[1 2 3]
 [4 5 6]]
```

In [7]:
```
# 2. Creating Arrays with np.zeros()
# Creates an array filled with zeros. You need to specify the shape.

# 1D array of zeros
zeros_1d = np.zeros(5)
print("1D Zeros Array:", zeros_1d)
```

```
# 2D array of zeros
zeros_2d = np.zeros((2, 3))  # 2 rows, 3 columns
print("2D Zeros Array:\n", zeros_2d)
```

```
1D Zeros Array: [0. 0. 0. 0. 0.]
2D Zeros Array:
 [[0. 0. 0.]
 [0. 0. 0.]]
```

In [8]:
```
# 3. Creating Arrays with np.ones()
# Creates an array filled with ones. Similar to np.zeros(), you specify the shap

# 1D array of ones
ones_1d = np.ones(4)
print("1D Ones Array:", ones_1d)

# 2D array of ones
ones_2d = np.ones((3, 3))
print("2D Ones Array:\n", ones_2d)
```

```
1D Ones Array: [1. 1. 1. 1.]
2D Ones Array:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

In [9]:
```
# 4. Creating Arrays with np.arange()
# Creates an array with evenly spaced values between a start and stop value (sim

# Array from 0 to 9
array_range = np.arange(0, 10)
print("Range Array:", array_range)
```

```
Range Array: [0 1 2 3 4 5 6 7 8 9]
```

In [10]:
```
# 5. Creating Arrays with np.linspace()
# Creates an array with evenly spaced numbers over a specified range. Unlike np.

# Array of 5 evenly spaced numbers between 0 and 1
linspace_array = np.linspace(0, 1, 5)
print("Linspace Array:", linspace_array)
```

```
Linspace Array: [0.   0.25 0.5  0.75 1.  ]
```

* Array Indexing and Slicing

NumPy arrays support powerful indexing and slicing mechanisms, which allow you to access or modify subsets of an array.

In [11]:
```
# Indexing:
# For 1D arrays:
array = np.array([10, 20, 30, 40, 50])
print("Element at index 0:", array[0])
print("Element at index 3:", array[3])

# For 2D arrays:
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Element at row 1, col 2:", array_2d[1, 2])
print("First row:", array_2d[0])
```

```
Element at index 0: 10
Element at index 3: 40
Element at row 1, col 2: 6
First row: [1 2 3]
```

In [12]:
```python
# slicing:

# For 1D arrays:
array = np.array([10, 20, 30, 40, 50])
print("Elements from index 1 to 3:", array[1:4])
print("Every other element:", array[::3])
print("Elements in reverse:", array[::-2])

# For 2D arrays:
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Slice:\n", array_2d[1:3, 0:2])
```

```
Elements from index 1 to 3: [20 30 40]
Every other element: [10 40]
Elements in reverse: [50 30 10]
Slice:
 [[4 5]
 [7 8]]
```

3) Array operations:

Array operations in NumPy refer to mathematical and logical operations that can be performed on arrays. NumPy arrays are designed to handle large amounts of numerical data efficiently, and these operations are optimized for performance. These operations include arithmetic operations, element-wise functions, broadcasting, and more.

* Arithmetic Operations on NumPy Arrays

- Arithmetic operations are applied element-wise for NumPy arrays of the same shape.

In [13]:
```python
import numpy as np

# Define two arrays
array1 = np.array([10,20,30])
array2 = np.array([40,50,60])

# Addition
print("Addition:", array1 + array2)  # [5 7 9]

# Subtraction
print("Subtraction:", array1 - array2)  # [-3 -3 -3]

# Multiplication
print("Multiplication:", array1 * array2)  # [4 10 18]

# Division
print("Division:", array1 / array2)  # [0.25 0.4 0.5]
```

```
Addition: [50 70 90]
Subtraction: [-30 -30 -30]
Multiplication: [ 400 1000 1800]
Division: [0.25 0.4  0.5 ]
```

* Broadcasting: Operations on Arrays of Different Shapes

- NumPy supports broadcasting, which allows arithmetic operations on arrays of different shapes by stretching the smaller array to match the shape of the larger one.

Broadcasting Rules:

- 1. If the arrays have different numbers of dimensions, the shape of the smaller array is padded with ones on the left.
- 2. Arrays are compatible for broadcasting if, after padding, the dimensions are either the same or one of them is 1.

In [14]:
```
# 1D array and scalar
array = np.array([1, 2, 3])
print("Array + Scalar:", array + 5)

# 2D array and 1D array
matrix = np.array([[1, 2, 3], [4, 5, 6]])
row_vector = np.array([10, 20, 30])
print("Matrix + Row Vector:\n", matrix + row_vector)
```

```
Array + Scalar: [6 7 8]
Matrix + Row Vector:
 [[11 22 33]
 [14 25 36]]
```

* Element-Wise Operations and Functions

- NumPy provides a wide range of universal functions (ufuncs) that perform element-wise operations. These functions are optimized for performance and include mathematical, trigonometric, and exponential functions.

In [15]:
```
array = np.array([16,25,36,49,64])

# Square root
print("Square Root:", np.sqrt(array))

# Exponential (e^x)
print("Exponential:", np.exp(array))

# Sine function
angles = np.array([0, np.pi / 2, np.pi])
print("Sine:", np.sin(angles))

# Logarithm (natural log)
print("Log:", np.log(array))

# Absolute value
negatives = np.array([-1, -2, -3])
print("Absolute Value:", np.abs(negatives))
```

```
Square Root: [4. 5. 6. 7. 8.]
Exponential: [8.88611052e+06 7.20048993e+10 4.31123155e+15 1.90734657e+21
 6.23514908e+27]
Sine: [0.0000000e+00 1.0000000e+00 1.2246468e-16]
Log: [2.77258872 3.21887582 3.58351894 3.8918203  4.15888308]
Absolute Value: [1 2 3]
```

4) Array shape and reshaping :

Shape:

In Python, particularly with NumPy, the array shape refers to the structure of an array—how many elements it has along each dimension or axis. It is a critical concept in working with arrays because it determines how data is organized and manipulated.

- The shape of an array is represented as a tuple of integers, where:
- Each integer specifies the number of elements in a corresponding dimension (axis).
- The number of integers in the shape tuple is equal to the number of dimensions (ndim) of the array.

  Key Points

- 1. A 1D array has a shape of (n,), where n is the number of elements.
- 2. A 2D array has a shape of (rows, columns).
- 3. A 3D array or higher-dimensional array has more entries in the shape tuple, representing the size along each dimension.

Reshape:

- np.reshape() is a function in numpy that allows you to change the shape of an array while keeping the same data.
- It return a new array with the specified a shape without chaning the original arrays data.

In [16]:
```python
# Shape
array = np.array([[1, 2], [3, 4], [5, 6]])
print("Shape of the array:", array.shape)

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

# Reshape into 2x3 matrix
reshaped = array.reshape(2, 3)
print("Reshaped Array Shape:", reshaped.shape)
```

```
Shape of the array: (3, 2)
Reshaped Array Shape: (2, 3)
```

* Checking Array Dimensions with .shape and .ndim in NumPy

In NumPy, arrays have attributes that help you explore their structure. Two important ones are:

- .shape: Returns a tuple representing the dimensions of the array (size along each axis).
- .ndim: Returns the number of dimensions (axes) of the array.

The .shape attribute gives a tuple where:

- 1. Each value represents the size of the array along a specific dimension (axis).
- 2. The length of the tuple is equal to the number of dimensions.

The .ndim attribute tells you

- 1. the number of dimensions (axes) in the array.
- 2. This is useful for understanding the complexity of the array's structure.

In [17]:
```
data = np.random.rand(100, 10)

print("Data Shape:", data.shape)  # (100, 10)
print("Number of Dimensions:", data.ndim)  # 2
```

```
Data Shape: (100, 10)
Number of Dimensions: 2
```

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

# Reshape to 2D
reshaped = array.reshape(2, 3)
print("Reshaped Shape:", reshaped.shape)  # (2, 3)
print("Reshaped Dimensions:", reshaped.ndim)  # 2
```

```
Reshaped Shape: (2, 3)
Reshaped Dimensions: 2
```

* Changing Array Shape in NumPy

NumPy provides several methods to manipulate the shape of an array without altering its data. These include:

- 1. .reshape() - Reshapes an array into a new shape.
- 2. .flatten() - Flattens a multi-dimensional array into a 1D array.
- 3. .transpose() - Transposes the dimensions of an array.

1. Reshaping an Array with .reshape()

- The .reshape() method changes the shape of an array to a specified shape, provided the total number of elements remains the same.
- Syntax:- array.reshape(new_shape)

2. Flattening an Array with .flatten()

- The .flatten() method converts a multi-dimensional array into a 1D array.
- Syntax:

array.flatten()

3. Transposing an Array with .transpose()

- The .transpose() method swaps or reverses the axes of an array, changing its shape accordingly.
- Syntax:

array.transpose(*axes)

In [19]:
```
# 1) reshape()
array = np.array([1, 2, 3, 4, 5, 6])
# Reshape to 2 rows and 3 columns
reshaped = array.reshape(2, 3)
print("Reshaped Array:\n", reshaped)

# 2) flatten()
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
# Flatten to 1D
flattened = array_2d.flatten()
print("Flattened Array:", flattened)

# 3) transpose()
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
# Transpose
transposed = array_2d.transpose()
print("Transposed Array:\n", transposed)
```

```
Reshaped Array:
 [[1 2 3]
 [4 5 6]]
Flattened Array: [1 2 3 4 5 6]
Transposed Array:
 [[1 4]
 [2 5]
 [3 6]]
```

* Concatenating and splitting arrays using np.concatenate(), np.vstack(), np.hstack(), np.split().

1)np.concatenate()

This function joins arrays along an existing axis.Concatenation means combining two or more arrays into one.

- Syntax: np.concatenate((array1, array2, ...), axis=0)
- axis=0 (default): Join along rows (vertically for 2D arrays).
- axis=1: Join along columns (horizontally for 2D arrays).

2) np.vstack()

Stacks arrays vertically (along rows). Equivalent to np.concatenate() with axis=0.

3) 3. np.hstack()

Stacks arrays horizontally (along columns). Equivalent to np.concatenate() with axis=1.

4) np.split()

Splits an array into multiple subarrays along an axis.Splitting breaks an array into multiple subarrays.

- Syntax: np.split(array, indices_or_sections, axis=0)
- indices_or_sections: Defines how to split the array (can be an integer or a list of indices).
- axis=0 (default): Split along rows.
- axis=1: Split along columns.

In [65]:
```
# np.concatenate()¶
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])
result = np.concatenate((array1, array2))
print("Concatenated 1D Array:", result)

# Vertically stack arrays
result_vstack = np.vstack((array1, array2))
print("Vertically Stacked Array:\n", result_vstack)

# Horizontally stack arrays
result_hstack = np.hstack((array1.reshape(-1, 1), array2.reshape(-1, 1)))
print("Horizontally Stacked Array:\n", result_hstack)

# 4) np.split()
# 2D array split
array2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
splits_2d = np.split(array2d, 3)  # Split along rows (default axis=0)
print("Split 2D Arrays (rows):", splits_2d)

# Split along columns
splits_2d_cols = np.split(array2d, 3, axis=1)
print("Split 2D Arrays (columns):", splits_2d_cols)
```

```
Concatenated 1D Array: [1 2 3 4 5 6]
Vertically Stacked Array:
 [[1 2 3]
 [4 5 6]]
Horizontally Stacked Array:
 [[1 4]
 [2 5]
 [3 6]]
Split 2D Arrays (rows): [array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8,
9]])]
Split 2D Arrays (columns): [array([[1],
       [4],
       [7]]), array([[2],
       [5],
       [8]]), array([[3],
       [6],
       [9]])]
```

5) Array manipulation

modify the shape of an array as well as split or join arrays into new arrays. Array manipulation allows you to do tasks such as add, remove, or transform elements in your array.

Common Array Manipulation Techniques in NumPy

- 1. Reshaping Arrays
- 2. Concatenating and Stacking
- 3. Splitting Arrays
- 4. Indexing and Slicing
- 5. Modifying Array Data
- 6. Changing Array Order

* Indexing and slicing: understanding multi-dimensional arrays.

Indexing and Slicing in Multi-Dimensional Arrays (NumPy) In NumPy, arrays can be multi-dimensional (i.e., 1D, 2D, 3D, and beyond), and indexing and slicing allow you to access and modify specific elements or subarrays. For multi-dimensional arrays, NumPy uses multi-indexing to access elements, and slicing can extract subarrays from these multi-dimensional structures.

Let's break down indexing and slicing for multi-dimensional arrays (2D, 3D, etc.) in detail.

1. Indexing in Multi-Dimensional Arrays

In [30]:
```
# A 1D array is a simple array with just one axis. You can access elements using
# Create a 1D array
a = np.array([10, 20, 30, 40, 50])
element = a[2]
print("elements",element)

# 2D Array (Matrix Indexing)
# A 2D array has two axes (rows and columns), and you use two indices: one for t

# Create a 2D array (matrix)
b = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
element = b[1, 2]
print("element",element)
```

```
elements 30
element 60
```

2. Slicing in Multi-Dimensional Arrays

Slicing is used to extract subarrays from the original array, based on specified ranges of indices.

In [31]:
```
# 1D Array Slicing
# Slicing a 1D array works the same way as slicing a list in Python. You can spe
# Create a 1D array
a = np.array([10, 20, 30, 40, 50])
```

```
# Slice from index 1 to 4 (not including 4)
slice_result = a[1:4]
print("slice_result",slice_result)
# Slice with step (every second element)
slice_result = a[::2]
print("slice_result",slice_result)

# 2D Array Slicing
# In a 2D array, you can slice both rows and columns separately, or even slice s
# Create a 2D array
b = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
# Slice rows 0 to 1, and columns 1 to 2
slice_result = b[0:2, 1:3]
print("slice_result",slice_result)
# Slice all rows and columns 1 to 2
slice_result = b[:, 1:3]
print("slice_result",slice_result)


# 3D Array Slicing
# Slicing a 3D array allows you to extract subarrays based on all three dimensio
# Create a 3D array
c = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
# Slice all depth levels, rows 0 to 1, and columns 1 to 2
slice_result = c[:, 0:1, 1:2]
print("slice_result",slice_result)
# Slice all depth levels and all rows and columns
slice_result = c[:, :, :]
print("slice_result", slice_result)
```

```
slice_result [20 30 40]
slice_result [10 30 50]
slice_result [[20 30]
 [50 60]]
slice_result [[20 30]
 [50 60]
 [80 90]]
slice_result [[[2]]

 [[6]]]
slice_result [[[1 2]
  [3 4]]

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

* Masking in NumPy (Logical Indexing Based on Conditions)

Masking (also known as logical indexing) is a powerful feature in NumPy that allows you to select or manipulate elements of an array based on specific conditions. This is done by creating a boolean mask (an array of True and False values) that corresponds to the elements of the array, and then using this mask to filter, select, or modify elements.

How Masking Works

- 1. Create a condition: The condition is applied to the array and produces a boolean array of the same shape.

- 2. Apply the mask: Use the boolean array (mask) to index the original array. Only the elements corresponding to True in the mask are selected.

In [34]:
```
import numpy as np
# Create an array
array = np.array([10, 20, 30, 40, 50])
# Create a mask for elements greater than 25
mask = array > 25
print("mask",mask)
# Use the mask to filter the array
filtered_array = array[mask]
print("filtered_array",filtered_array)
```

```
mask [False False  True  True  True]
filtered_array [30 40 50]
```

* Fancy indexing (using arrays of indices to select elements).

Fancy Indexing in NumPy Fancy indexing in NumPy refers to using arrays of indices (integer arrays) to select elements from a NumPy array. It allows for non-sequential, flexible, and customized selection of elements based on the indices you specify.

Unlike regular slicing, which uses ranges and produces contiguous subarrays, fancy indexing gives you the ability to select elements in any desired order or pattern.

In [38]:
```
# Create a 2D array
array_2d = np.array([[10, 20, 30],
                     [40, 50, 60],
                     [70, 80, 90]])

# Specify row and column indices for each desired element
row_indices = [0, 1, 2]
col_indices = [2, 0, 1]

selected_elements = array_2d[row_indices, col_indices]
print("selected_elements",selected_elements)
```

```
selected_elements [30 40 80]
```

6) Mathematical Functions in Python

Mathematical functions in Python allow you to perform a wide variety of mathematical operations. These functions are primarily available in the math module and NumPy library. They cover basic arithmetic, trigonometric operations, logarithmic and exponential computations, complex number operations, and more.

* Universal Functions (ufuncs) in NumPy

Universal functions (ufuncs) in NumPy are functions that operate element-wise on arrays. They allow for efficient operations on entire arrays without needing explicit loops. These functions are optimized for speed and can handle broadcasting, making them a powerful feature of NumPy.

Common Universal Functions (ufuncs)

Some of the most frequently used ufuncs include:

- 1. np.add(): Performs element-wise addition.
- 2. np.subtract(): Performs element-wise subtraction.
- 3. np.multiply(): Performs element-wise multiplication.
- 4. np.divide(): Performs element-wise division.
- 5. np.sqrt(): Computes the square root element-wise.
- 6. np.exp(): Computes the exponential (e^x) element-wise.
- 7. np.sin(): Computes the sine of elements element-wise.
- 8. np.log(): Computes the natural logarithm element-wise.

In [54]:
```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Using ufuncs
result_add = np.add(a, b)  # Element-wise addition
result_sub = np.subtract(a, b)  # Element-wise subtraction
result_mul = np.multiply(a, b)  # Element-wise multiplication

print(result_add)
print(result_sub)
print(result_mul)
```

```
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
```

* Statistical Functions in NumPy

NumPy provides a collection of powerful statistical functions that allow you to easily calculate key statistics for your arrays. These functions help in analyzing the distribution, spread, and other important characteristics of your data.

Some of the most commonly used statistical functions are:

- 1. np.mean(): Computes the mean (average) of the array.
- 2. np.median(): Computes the median (middle value) of the array.
- 3. np.std(): Computes the standard deviation, which measures the spread of the data.
- 4. np.var(): Computes the variance, which is the square of the standard deviation.
- 5. np.min(): Computes the minimum value in the array.
- 6. np.max(): Computes the maximum value in the array.

In [66]:
```
array = np.array([10,20,30,40,50,60,70,80,90])

# Compute statistics
mean_val = np.mean(array)
median_val = np.median(array)
std_val = np.std(array)
var_val = np.var(array)
min_val = np.min(array)
max_val = np.max(array)
```

```
print(f"Mean: {mean_val}")
print(f"Median: {median_val}")
print(f"Standard Deviation: {std_val}")
print(f"Variance: {var_val}")
print(f"Minimum Value: {min_val}")
print(f"Maximum Value: {max_val}")
```

```
Mean: 50.0
Median: 50.0
Standard Deviation: 25.81988897471611
Variance: 666.6666666666666
Minimum Value: 10
Maximum Value: 90
```

* Linear Algebra Functions in NumPy

NumPy provides a set of functions that are extremely useful for linear algebra operations. These functions allow you to perform matrix and vector operations, which are fundamental in fields like physics, machine learning, and data science. Some of the most commonly used linear algebra functions in NumPy include:

- 1. np.dot(): Computes the dot product of two arrays.
- 2. np.matmul(): Performs matrix multiplication.
- 3. np.linalg.inv(): Computes the inverse of a matrix.
- 3. np.linalg.inv() Computes the eigenvalues and eigenvectors of a matrix.

In [64]:
```
#1) np.dot()
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

dot_product = np.dot(a, b)
print(dot_product)
```

```
32
```

In [65]:
```
#2) np.matmul()
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

matmul_result = np.matmul(A, B)
print(matmul_result)
```

```
[[19 22]
 [43 50]]
```

In [62]:
```
#3) np.linalg.inv()
A = np.array([[1, 2], [3, 4]])

# Compute the inverse of the matrix
A_inv = np.linalg.inv(A)
print(A_inv)
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

In [66]:
```
# 3) np.linalg.inv()
A = np.array([[4, -2], [1, 1]])
```

```
# Compute the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```

```
Eigenvalues: [3. 2.]
Eigenvectors: [[0.89442719 0.70710678]
 [0.4472136  0.70710678]]
```

7) Random Number Generation in NumPy

Random number generation is an essential feature in data analysis, simulations, machine learning, and scientific computations. NumPy provides a dedicated module called numpy.random that offers a wide range of functions to generate random numbers, arrays, and samples from various distributions.

In [41]:
```
a=np.random.rand(12)
print(a)
```

```
[0.88219056 0.23140929 0.4255449  0.24738294 0.73969643 0.05520343
 0.52492787 0.40706852 0.70054017 0.00905697 0.16244168 0.64768197]
```

* Generating Random Numbers Using np.random in NumPy

The numpy.random module provides a variety of functions to generate random numbers. These numbers can be scalars, arrays, or from specific distributions.

In [45]:
```
# 1. Random Numbers Between 0 and 1
# np.random.rand(): Generates random floats uniformly distributed between 0 and

# Single random number
random = np.random.rand()
print(random)

# Array of random numbers
random_array = np.random.rand(3, 4)  # Shape: 3x4
print(random_array)
```

```
0.40199544267987875
[[0.02966098 0.11335349 0.27930595 0.89653191]
 [0.62118663 0.6207262  0.71214537 0.4133355 ]
 [0.75535486 0.69152326 0.64141004 0.53966268]]
```

In [46]:
```
# 2. Random Integers
# np.random.randint(low, high, size): Generates random integers between low (inc

# Single random integer between 1 and 10
random_int = np.random.randint(1, 10)
print(random_int)

# Array of random integers
```

```
random_int_array = np.random.randint(1, 10, size=(2, 3))  # Shape: 2x3
print(random_int_array)
```

```
1
[[8 4 9]
 [4 9 2]]
```

* Seeding Random Number Generation in NumPy (np.random.seed())

- When generating random numbers, the sequence is actually determined by a pseudo-random number generator (PRNG). The PRNG starts with an initial value called the seed. By setting the seed, you can ensure reproducibility of the random number sequence.
- To set the seed, use np.random.seed(seed), where seed is an integer. Once set, all random numbers generated will follow the same sequence until the seed is changed or reset.

This is particularly useful when:

- 1. Debugging code.
- 2. Sharing results with others.
- 3. Ensuring consistent training/testing in machine learning.

In [47]:
```
# Set a seed
np.random.seed(42)

# Generate random numbers
print(np.random.rand(3))
```

```
[0.37454012 0.95071431 0.73199394]
```

* Random sampling, shuffling, and permutations.

1. Random Sampling

np.random.choice() This function allows you to randomly sample elements from a 1D array or range, with or without replacement.

Parameters:

- a: The array or range to sample from.
- size: Number of elements to sample.
- replace: Whether to sample with replacement (default is True).
- p: Probabilities for each element (optional).

In [67]:
```
# Sampling from an array
arr = np.array([10, 20, 30, 40, 50])

# Randomly select 3 elements with replacement
sample = np.random.choice(arr, size=3, replace=True)
print("sample",sample)

# Randomly select 3 elements without replacement
```

```
sample_no_replace = np.random.choice(arr, size=3, replace=False)
print("sample_no_replace",sample_no_replace)
```

```
sample [40 10 40]
sample_no_replace [30 40 10]
```

2. Shuffling

Shuffling is used to rearrange the elements of an array randomly. NumPy provides two main functions for this:

In [68]:
```
# np.random.shuffle()
# Shuffles the array in-place (modifies the original array).

arr = np.array([1, 2, 3, 4, 5])
np.random.shuffle(arr)
print(arr)
```

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

3. Generating Permutations

Permutations are ordered arrangements of elements. Use np.random.permutation() to generate permutations of an array or integers in a range.

In [69]:
```
# Permutation of integers from 0 to 9
perm = np.random.permutation(10)
print("perm",perm)

# Permutation of an array
arr = np.array(['a', 'b', 'c', 'd'])
perm_arr = np.random.permutation(arr)
print("perm_arr",perm_arr)
```

```
perm [4 8 1 3 0 5 2 9 7 6]
perm_arr ['a' 'c' 'b' 'd']
```

8) Broadcasting in NumPy

Broadcasting is a powerful feature in NumPy that allows you to perform arithmetic operations on arrays of different shapes without needing to explicitly reshape or replicate the arrays. It simplifies operations on arrays with different dimensions.

Key Concept

When performing operations between two arrays (e.g., addition, subtraction, etc.), NumPy compares their shapes element-wise. Broadcasting applies when:

- 1. The two arrays have different shapes.
- 2. NumPy automatically "expands" one or both arrays to make their shapes compatible.

Broadcasting Rules

Two arrays are compatible for broadcasting if:

- 1. Their dimensions are equal, or
- 2. One of them is 1.

When dimensions differ, the smaller array is "broadcast" (stretched or repeated) along the dimension(s) of the larger array to match its shape.

Advantages of Broadcasting

- 1. Simplicity: Eliminates the need to manually reshape or replicate arrays.
- 2. Efficiency: NumPy implements broadcasting without creating large intermediate arrays, making it memory-efficient and faster.
- 3. Flexibility: Allows operations between arrays of different dimensions directly.

Common Use Cases

- Adding a scalar to an array.
- Element-wise operations between arrays of different shapes.
- Applying functions across rows or columns.

* Understanding how NumPy handles operations between arrays of different shapes.

When performing operations between arrays of different shapes, NumPy uses broadcasting to make the arrays compatible. Broadcasting automatically expands the smaller array so that it matches the dimensions of the larger array without actually copying data. This process makes operations more memory-efficient and faster.

Let's break down how NumPy handles operations on arrays of different shapes using broadcasting.

One of the arrays has a dimension of size 1 along the axis, in which case it will be stretched to match the other array's size. Steps in Broadcasting:

- Step 1: Compare the shapes of the two arrays, starting from the rightmost dimension.
- Step 2: If the dimensions are the same or one is 1, they are compatible.
- Step 3: If the dimensions are not compatible, NumPy will raise a ValueError.

* Array with Different Shapes

When two arrays have different shapes, NumPy compares them to see if they are compatible according to the broadcasting rules.

In [58]:
```
# Array 1: (3, 1)
arr1 = np.array([[1], [2], [3]])

# Array 2: (1, 3)
```

```
arr2 = np.array([10, 20, 30])

# Broadcasting
result = arr1 + arr2
print(result)
```

```
[[11 21 31]
 [12 22 32]
 [13 23 33]]
```

* Broadcasting Rules

1. If the arrays have different numbers of dimensions, pad the shape of the smaller array with ones on the left (i.e., prepend dimensions of size 1).
2. If the arrays have the same number of dimensions, the sizes of the dimensions must either be:

- The same, or
- One of the dimensions must be 1, in which case the smaller array will be "stretched" to match the larger array along that axis.

3. Arrays with the shape (1, x) or (x, 1) can be broadcast to match arrays with the shape (y, z), where y or z is 1. This means the array with size 1 can be expanded along the axis with size greater than 1.
4. If the shapes are incompatible and do not meet these conditions, NumPy raises a ValueError.

1. Matching Dimensions

The arrays must match along each dimension (or one dimension must be 1). If the dimensions are incompatible, broadcasting fails.

In [63]:
```
arr1 = np.array([1, 2, 3])  # Shape: (3,)
arr2 = np.array([[1, 2], [3, 4]])  # Shape: (2, 2)

# This will raise a ValueError
result = arr1 + arr2

# arr1 has shape (3,), and arr2 has shape (2, 2).
# These shapes are not compatible because neither dimension of the arrays can be
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[63], line 5
      2 arr2 = np.array([[1, 2], [3, 4]])  # Shape: (2, 2)
      4 # This will raise a ValueError
----> 5 result = arr1 + arr2

ValueError: operands could not be broadcast together with shapes (3,) (2,2)
```

2. Aligning Dimensions

When performing operations on arrays with different shapes, NumPy aligns the dimensions starting from the rightmost one (the last axis). If the number of dimensions is

different, it assumes the smaller array has a leading set of dimensions with size 1.

In [64]:
```
arr1 = np.array([1, 2, 3])  # Shape: (3,)
arr2 = np.array([[10], [20], [30]])  # Shape: (3, 1)

# Broadcasting happens here
result = arr1 + arr2
print(result)
```

```
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

In [ ]: