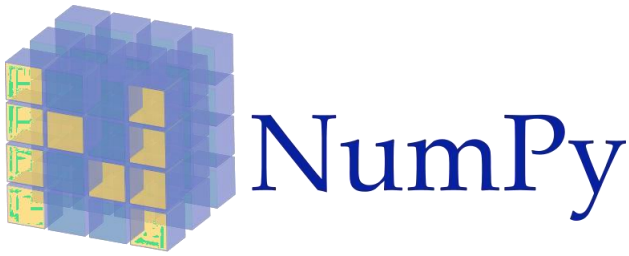
NumPy

Python

AVINASH A S



1. NumPy

NumPy (Numerical Python) is a powerful library for numerical computing in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures efficiently. NumPy is widely used in data science, machine learning, and scientific computing.

Installing NumPy

We can install NumPy using pip, which is the package installer for Python. Open your command prompt or terminal and type:

```
pip install numpy
```

Importing NumPy

Once installed, we can import NumPy into your Python script or interactive session. It is common practice to import NumPy with the alias **np**:

```
import numpy as np
```

Example Code:

```
import numpy as np

# Creating a NumPy array
array = np.array([1, 2, 3, 4, 5])

# Performing a basic operation
sum_array = np.sum(array)

print("Array:", array)
print("Sum of array elements:", sum_array)
```

Output :

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

Sum of array elements: 15
```

2. Basics of NumPy Arrays

Creating Arrays

1. np.array() - The np.array() function creates a NumPy array from a Python list, tuple, or other data structure.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Breakdown:

1. np.array([1, 2, 3, 4, 5]): This creates a NumPy array with the values from the provided Python list.
2. print(arr): Displays the array in the console.

2. np.zeros() - The np.zeros() function creates an array of the given shape with all values set to 0.

```
import numpy as np
zeros_array = np.zeros((2, 3))
print(zeros_array)
```

Breakdown:

1. np.zeros((2, 3)): Creates a 2x3 array (2 rows, 3 columns) with all elements set to 0.
2. print(zeros_array): Prints the created array.

3. np.ones() - The np.ones() function works similarly to np.zeros() but initializes all elements to 1.

```
import numpy as np
ones_array = np.ones((2, 3))
print(ones_array)
```

Breakdown:

1. np.ones((2, 3)): Creates a 2x3 array with all elements initialized to 1.
2. print(ones_array): Displays the array.

4. np.arange() - The np.arange() function creates an array with evenly spaced values within a given range, much like Python's range() function.

```
import numpy as np
arr = np.arange(0, 10, 2)
print(arr)
```

Breakdown:

1. np.arange(0, 10, 2): Creates an array with values starting at 0, stopping before 10, with a step size of 2.
2. print(arr): Displays the array [0, 2, 4, 6, 8].

5. np.linspace() - The np.linspace() function generates an array with a specified number of evenly spaced values between a given start and stop value.

```
import numpy as np
linspace_array = np.linspace(0, 1, 5)
print(linspace_array)
```

Breakdown:

1. np.linspace(0, 1, 5): Generates 5 evenly spaced numbers between 0 and 1.
2. print(linspace_array): Displays the generated array [0.0, 0.25, 0.5, 0.75, 1.0].

Array Attributes

1. **shape -** It returns a tuple representing the number of rows and columns for a 2D array, or the size for each axis in multi-dimensional arrays.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)
```

Breakdown:

- arr = np.array([[1, 2, 3], [4, 5, 6]]): Creates a 2D array with two rows and three columns.
- arr.shape: Returns the shape of the array as (2, 3) (2 rows, 3 columns).
- print(arr.shape): Displays the shape of the array.

2. **Dtype -** It specifies the type of data (e.g., integers, floats, etc.) stored in the array. You can set the dtype when creating the array.

```
import numpy as np
arr = np.array([1, 2, 3], dtype='float64')
print(arr.dtype)
```

Breakdown:

- arr = np.array([1, 2, 3], dtype='float64'): Creates an array with the values [1, 2, 3] and explicitly sets the data type to float64.
- arr.dtype: Returns the data type of the array elements, which is float64.
- print(arr.dtype): Displays the data type of the array.

3. **Size -** The size attribute returns the total number of elements in the array. This is the product of the shape dimensions (i.e., the number of elements in each axis).

```
import numpy as np
arr = np.array([[1, 2], [3, 4], [5, 6]])
print(arr.size)
```

Breakdown:

- arr = np.array([[1, 2], [3, 4], [5, 6]]): Creates a 2D array with 3 rows and 2 columns.
- arr.size: Returns the total number of elements in the array (3 rows × 2 columns = 6 elements).
- print(arr.size): Displays the size of the array.

4. **Ndim -** The ndim attribute gives the number of dimensions (or axes) of the array. This tells you whether the array is 1D, 2D, or has more dimensions.

```
import numpy as np
arr = np.array([1, 2, 3])
print(arr.ndim)
```

Breakdown:

- arr = np.array([1, 2, 3]): Creates a 1D array (a single row of elements).
- arr.ndim: Returns the number of dimensions of the array (in this case, 1D).
- print(arr.ndim): Displays the number of dimensions, which is 1.

3. Array Indexing and Slicing

Array indexing and slicing are allowing you to access and manipulate specific elements, rows, and columns in an array.

Basic Indexing - Indexing in NumPy arrays works similarly to Python lists but with more powerful features, especially when dealing with multidimensional arrays.

- **1D Array Indexing** - We can access individual elements in a 1D NumPy array using their index positions.

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print(arr[0])
print(arr[2])
```

- **2D Array Indexing** - In 2D arrays (matrices), you need to specify both the row and column index.

```
import numpy as np

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

# Accessing individual elements
print(arr[0])  # Output: 10
print(arr[2])  # Output: 30
```

You can also access entire rows or columns using : (colon) notation:

```
import numpy as np

arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing entire row and column
print(arr_2d[0, :])  # Output: [1, 2, 3] (first row)
print(arr_2d[:, 1])  # Output: [2, 5, 8] (second column)
```

Slicing Arrays - Slicing allows us to extract specific parts of an array based on index ranges.

Syntax : `array_name[start:end:step]`

start: The starting index (inclusive)

end: The ending index (exclusive)

step: The step or stride between indices

- **1D Array Slicing**: We can extract a portion of a 1D array by specifying start and end positions.

```
import numpy as np

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

# Slicing with start and end positions
print(arr[1:4])
# Output: [20, 30, 40]

print(arr[:3])
# Output: [10, 20, 30] (up to the third element)

print(arr[::2])
# Output: [10, 30, 50] (every second element)
```

- **2D Array Slicing**: Similarly, we can slice multidimensional arrays by specifying rows and columns.

```
import numpy as np

 arr_2d = np.array([[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]])

# Slicing rows and columns

print(arr_2d[0:2, 1:3])
# Output: [[2, 3], [5, 6]]

print(arr_2d[:, 1:3])
# Output: [[2, 3], [5, 6], [8, 9]] (all rows, second to third column)
```

Slicing creates a **view** of the original array, meaning that changes to the slice will affect the original array.

Boolean Indexing - Boolean indexing allows you to filter elements of an array based on conditions. It returns a new array with only the elements that satisfy the condition.

```
import numpy as np

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

# Boolean condition to filter elements
bool_idx = arr > 3
print(bool_idx)    # Output: [False False False  True  True]
print(arr[bool_idx])  # Output: [4, 5]
```

- **Multiple Conditions:** You can combine conditions using logical operators (&, |, ~):

```
import numpy as np

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

# Using logical operators to combine conditions
print(arr[(arr > 2) & (arr < 5)])  # Output: [3, 4]
print(arr[~(arr % 2 == 0)])  # Output: [1, 3, 5] (not divisible by 2)
```

Fancy Indexing - Fancy indexing refers to indexing using lists or arrays of indices.

- **1D Array Fancy Indexing:**

```
import numpy as np

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

# Fancy indexing using a list of indices
indices = [0, 2, 4]
print(arr[indices])  # Output: [10, 30, 50]
```

- **2D Array Fancy Indexing:** You can also use fancy indexing with 2D arrays by providing a list of row and column indices.

```
import numpy as np

arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Fancy indexing with a list of row and column indices
row_indices = [0, 1, 2]
col_indices = [2, 1, 0]
print(arr_2d[row_indices, col_indices])  # Output: [3, 5, 7]
```

Fancy indexing creates a **copy** of the original array, so changes to the indexed array do not affect the original array.

4. Array Operations

Element-wise Operations (arrays of the same shape)

```
import numpy as np

# Creating two arrays of the same shape
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Element-wise +, -, *, /
print(a + b)  # Output: [5 7 9]
print(a - b)  # Output: [-3 -3 -3]
print(a * b)  # Output: [ 4 10 18]
print(a / b)  # Output: [0.25 0.4  0.5 ]
print(a > b)  # Output: [False False False]
```

Scalar Arithmetic Operations

```
import numpy as np

#Creating Arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print(a + 5)  # Output: [6 7 8]

print(a - 1)  # Output: [0 1 2]

print(a * 2)  # Output: [2 4 6]

print(b / 2)  # Output: [2.  2.5 3. ]
```

Universal Functions (ufuncs)

```
import numpy as np

#Creating Arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = np.add(a, b)
print(c)  # Output: [5 7 9]

c = np.subtract(b, a)
print(c)  # Output: [3 3 3]

c = np.multiply(a, b)
print(c)  # Output: [ 4 10 18]

c = np.divide(b, a)
print(c)  # Output: [4.  2.5 2. ]

angles = np.array([0, np.pi/2, np.pi])
sine_values = np.cos(angles)
print(sine_values)  # Output: [0.0000000e+00 1.0000000e+00 1.2246468e-16]
```

Aggregate Functions(Performs Across the Whole Array**)**

```
import numpy as np

a = np.array([1, 2, 3, 4])

total = np.sum(a)
print(total)  # Output: 10

mean_value = np.mean(a)
print(mean_value)  # Output: 2.5

std_value = np.std(a)
print(std_value)  # Output: 1.118033988749895

min_value = np.min(a)
max_value = np.max(a)
print(min_value, max_value)  # Output: 1 4

cumulative_sum = np.cumsum(a)
print(cumulative_sum)  # Output: [ 1  3  6 10]

product = np.prod(a)
print(product)  # Output: 24
```

5. Reshaping and Resizing

Reshaping Arrays

The **`reshape()`** method gives a new shape to an array without changing its data. You must ensure that the total number of elements remains consistent.

```
import numpy as np

arr = np.arange(12)
print("Original Array:\n", arr)

# Reshaping the array to 3x4
print("\nReshaped Array (3x4):\n", arr.reshape(3, 4))

# Reshaping the array to 2x6
print("\nReshaped Array (2x6):\n", arr.reshape(2, 6))
```

Output:

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

Reshaped Array (3x4):
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Reshaped Array (2x6):
 [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
```

The **`flatten()`** method converts a multi-dimensional array into a one-dimensional array (a flat array). It creates a copy of the original array.

```
import numpy as np

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

# Flattening the array
print("Flattened Array:\n", arr.flatten())
```

Output:

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

The **`ravel()`** method also converts a multi-dimensional array into a one-dimensional array, but unlike flatten(), it returns a view (or reference) of the original array if possible, meaning changes to the result **may affect the original array.**

```
import numpy as np

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

# Raveling the array
ravelled_arr = arr.ravel()
print("Ravelled Array:\n", ravelled_arr)

# Modifying the ravelled array
ravelled_arr[0] = 10
print("\nModified Ravelled Array:\n", ravelled_arr)
print("Original Array after modification:\n", arr)
```

Output:

```
Ravelled Array:
 [1 2 3 4 5 6]
Modified Ravelled Array:
 [10  2  3  4  5  6]
Original Array after modification:
 [[10  2  3]
 [ 4  5  6]]
```

Resizing Arrays

```
import numpy as np

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

# Resizing to a larger array (2x4)
arr.resize(2, 4)
print("Resized Array (2x4):\n", arr)

# Resizing to a smaller array (1x3)
arr.resize(1, 3)
print("\nResized Array (1x3):\n", arr)
```

Output:

```
Resized Array (2x4):
 [[1 2 3 4]
 [5 6 0 0]]


Resized Array (1x3):
 [[1 2 3]]
```

Differences between `reshape()` and `resize()`

- **`reshape()`** creates a new array or view of the original data, but the number of elements must remain the same. It will raise an error if the total number of elements doesn't match.
- **`resize()`** directly modifies the array's shape and can change the total number of elements, either truncating or padding with zeros if necessary.

6. Advanced Array Operations

Broadcasting

Broadcasting allows NumPy to perform element-wise operations on arrays of different shapes. NumPy automatically "stretches" the smaller array along the dimensions with a size of 1 so that they have the same shape.

Broadcasting Rules:

1. If the arrays do not have the same rank (i.e., number of dimensions), prepend 1 to the smaller array's shape until they match in rank.
2. For arrays with the same rank, NumPy compares them element-wise:
    - If the dimensions are equal, proceed.
    - If one of the dimensions is 1, stretch it to match the other dimension.
    - If the dimensions do not match and neither is 1, broadcasting cannot be performed.

Element-wise addition with broadcasting

```
import numpy as np

# Array A of shape (3, 1)
A = np.array([[1],
              [2],
              [3]])

# Array B of shape (1, 4)
B = np.array([[10, 20, 30, 40]])

# Broadcasting A and B for element-wise addition
C = A + B
print(C)
```

Output:

```
[[11 21 31 41]
 [12 22 32 42]
 [13 23 33 43]]
```

Element-wise multiplication with broadcasting

```
import numpy as np

# Array X of shape (2, 3)
X = np.array([[1, 2, 3], [4, 5, 6]])

# Scalar value
Y = 10

# Broadcasting X with scalar Y
Z = X * Y
print(Z)

Z = X / Y
print(Z)
```

Output:

```
[[10 20 30]
 [40 50 60]]
[[0.1 0.2 0.3]
 [0.4 0.5 0.6]]
```

Stacking

vstack() – combines array row wise

Vertical stacking using 1D arrays

```
import numpy as np

# Two 1D arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Stacking them vertically
result = np.vstack((arr1, arr2))
print(result)
```

Output:

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

2D Array Vertical stacking using vstack() - joins arrays row-wise.

```
import numpy as np

# Two 2D arrays
arr1 = np.array([[1, 2, 3],
                 [3, 2, 1]])
arr2 = np.array([[4, 5, 6],
                 [6, 5, 4]])

# Stacking them vertically
result = np.vstack((arr1, arr2))
print(result)
```

Output:

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

hstack() - Horizontal stacking joins arrays column-wise.

Horizontal stacking using 1D arrays

```
import numpy as np

# Two 1D arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Stacking them horizontally
result = np.hstack((arr1, arr2))
print(result)
```

Output:

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

Horizontal stacking using 2D arrays

```
import numpy as np

# Two 2D arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# Stacking them horizontally
result = np.hstack((arr1, arr2))
print(result)
```

Output:

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

Splitting

Splitting divides an array into multiple sub-arrays along a specified axis. The split arrays are smaller versions of the original array.

split()

```
import numpy as np

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

# Splitting into 3 equal parts
result = np.split(arr, 3)
print(result)
```

Output:

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

vsplit()

```
import numpy as np

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

# Splitting vertically into 2 parts
result = np.vsplit(arr, 2)
print(result)
```

Output:

```
[array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]])]
```

hsplit()

```
import numpy as np

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

# Splitting horizontally into 2
parts
result = np.hsplit(arr, 2)
print(result)
```

Output:

```
[array([[1, 2],
        [5, 6]]),
 array([[3, 4],
        [7, 8]])]
```

7. Linear Algebra with NumPy

Dot Product

The **dot product** is a key operation in linear algebra, widely used in fields like physics, statistics, and machine learning. It involves multiplying corresponding elements of two vectors or matrices and summing the products.

Code:

```
import numpy as np

# Define two vectors
vector_a = np.array([1, 2, 3])
vector_b = np.array([4, 5, 6])

# Calculate dot product
dot_product = np.dot(vector_a, vector_b)
print(f"Dot Product: {dot_product}")
```

Output:

```
Dot Product: 32
```

Explanation

In the example, vector_a and vector_b are two vectors. The dot product is calculated as:

```
1*4 + 2*5 + 3*6 = 4 + 10 + 18 = 32
```

Applications

- **Physics:** Used to calculate work done by a force over a distance.
- **Machine Learning:** In measuring similarity between data points, particularly in clustering algorithms.
- **Statistics**: Used in covariance and correlation calculations.

Matrix Multiplication

Matrix multiplication combines two matrices to produce a new matrix. This operation is essential in transforming data across multiple dimensions and is widely used in computer graphics and machine learning.

Code:

```
import numpy as np

# Define two matrices
A = np.array([[1, 2],
              [3, 4]])
B = np.array([[5, 6],
              [7, 8]])

# Calculate matrix multiplication using the @ operator
matrix_product = A @ B

print("Matrix Product:", matrix_product)
```

Output:

```
Matrix Product:[[19 22]
 [43 50]]
```

Explanation: The multiplication of matrices AAA and BBB results in a new matrix:

$$\begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Real-life Application:

- **Computer Graphics:** Used to perform transformations such as rotation, scaling, and translation of objects.
- **Economics:** In modelling the interdependencies between different economic sectors using input-output tables.

Inverse of a Matrix

The **inverse** of a matrix is a matrix that, when multiplied with the original matrix, yields the identity matrix. The inverse is essential for solving systems of linear equations.

Code:

```
import numpy as np

# Define a square matrix
A = np.array([[1, 2], [3, 4]])

# Calculate the inverse of the matrix
inverse_A = np.linalg.inv(A)

print("Inverse of A:\n", inverse_A)
```

Output:

```
Inverse of A:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

Explanation: The inverse of matrix A is calculated such that:

$$A \cdot A^{-1} = I$$

For the given matrix A, the inverse results in:

$$\begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}$$

This inverse can be used to solve linear equations of the form Ax=b by multiplying both sides by A^{-1}.

Real-life Application:

- **Engineering:** In circuit analysis to find currents and voltages in electrical networks.
- **Economics:** To analyze supply and demand equilibrium conditions in markets.

Eigenvalues and Eigenvectors

Eigenvalues and **eigenvectors** are crucial in understanding linear transformations. They help in various applications like stability analysis and dimensionality reduction.

Code:

```
import numpy as np

# Define a square matrix
A = np.array([[4, -2],
              [1, 1]])

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

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

Output:

```
Eigenvalues: [3. 2.]

Eigenvectors:

 [[0.89442719 0.70710678]

 [0.4472136  0.70710678]]
```

Explanation:

For the matrix A, the eigenvalues are 3 and 2, which indicate the scaling factors along their corresponding eigenvectors:

- The first eigenvector scales by 3 when the transformation represented by A is applied.
- The second eigenvector scales by 2.

Real-life Application:

- **Data Science:** In Principal Component Analysis (PCA), eigenvalues and eigenvectors are used to reduce the dimensionality of data while retaining variance, aiding in visualization and analysis.
- **Mechanical Engineering:** Used to analyze the stability of structures and determine natural frequencies in vibration analysis.

8. Random Number Generation

- Random Sampling
    - np.random.rand()
    - np.random.randint()
    - np.random.choice()
- Setting Seed
    - np.random.seed()

9. Sorting and Searching

- Sorting Arrays
    - np.sort()
    - argsort()
- Searching Arrays
    - np.where()
    - np.searchsorted()

10. Advanced Indexing Techniques

- Boolean Indexing
- Integer Array Indexing
- Combining Indexing Techniques

11. Performance Optimization

- Vectorization
- Memory Layout and Strides
- Using numexpr for faster computations

12. Integration with Other Libraries

- Using NumPy with Pandas
- Using NumPy with SciPy
- Using NumPy with Matplotlib