NUMPY

Introduction to NumPy

NumPy (Numerical Python) is a powerful library for numerical computing in Python. It provides support for arrays, matrices, and a collection of mathematical functions to perform operations efficiently.

Installation

If you haven't installed NumPy yet, use the following command:

```
pip install numpy
```

Importing NumPy

You need to import NumPy before using it:

```
import numpy as np
```

---

1. NumPy Arrays

Creating Arrays

NumPy's core data structure is the **ndarray (N-dimensional array)**. You can create arrays using:

import numpy as np

```
# Creating a 1D array
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)

# Creating a 2D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
```

Checking Array Properties

```
print(arr1.ndim)  # Number of dimensions
print(arr2.shape)  # Shape of the array (rows, columns)
print(arr1.size)   # Total number of elements
print(arr1.dtype)  # Data type of elements
```

---

2. Creating Arrays with Default Values

NumPy provides functions to create arrays without manually specifying values.

```
# Creating an array of zeros
zeros = np.zeros((2, 3))
print(zeros)

# Creating an array of ones
ones = np.ones((3, 3))
print(ones)

# Creating an array of random numbers
random_arr = np.random.rand(2, 2)
print(random_arr)

# Creating an array with a range of numbers
arr_range = np.arange(1, 10, 2)  # Start, Stop, Step
print(arr_range)

# Creating an evenly spaced array
linspace_arr = np.linspace(1, 10, 5)  # Start, Stop, Number of values
print(linspace_arr)
```

---

3. Array Indexing and Slicing

You can access elements using indexing and slicing, just like lists.

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

# Accessing elements
print(arr[0])  # First element
print(arr[-1]) # Last element

# Slicing
print(arr[1:4])  # Elements from index 1 to 3
print(arr[:3])   # First 3 elements
print(arr[::2])  # Every second element
```

For **2D arrays**:

```
arr2D = np.array([[1, 2, 3], [4, 5, 6]])

print(arr2D[0, 1])  # Element at row 0, column 1
print(arr2D[:, 1])  # All rows, column 1
print(arr2D[1, :])  # Row 1, all columns
```

---

4. Mathematical Operations

NumPy supports element-wise arithmetic operations.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

print(arr1 + arr2)  # Element-wise addition
print(arr1 - arr2)  # Element-wise subtraction
print(arr1 * arr2)  # Element-wise multiplication
print(arr1 / arr2)  # Element-wise division
Useful functions:
print(np.sum(arr1))  # Sum of elements
print(np.mean(arr1)) # Mean (average)
print(np.max(arr1))  # Maximum value
print(np.min(arr1))  # Minimum value
print(np.sqrt(arr1)) # Square root
print(np.exp(arr1))  # Exponential
print(np.log(arr1))  # Natural logarithm
```

---

5. Reshaping and Transposing

NumPy allows you to reshape and transpose arrays easily.

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

reshaped = arr.reshape((3, 2))
print(reshaped)  # Reshape into 3 rows and 2 columns

transposed = arr.T
print(transposed)  # Transpose (swap rows and columns)
```

This is just the basics! Would you like to explore more advanced NumPy topics like broadcasting, linear algebra, or working with real-world datasets?

6. Broadcasting in NumPy

Broadcasting allows you to perform operations on arrays of different shapes without manually reshaping them.

Basic Example

import numpy as np

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([[1], [2], [3]])

result = arr1 + arr2  # Broadcasts arr1 to match arr2's shape
print(result)
```

Explanation:

- arr1 is a **(1, 3)** shape array
- arr2 is a **(3, 1)** shape array
- NumPy **broadcasts** arr1 to match arr2, so element-wise addition works.

Rules of Broadcasting

1. If arrays have **different dimensions**, NumPy adds dimensions to the smaller array.
2. If a dimension is **1**, it gets stretched to match the other array.
3. If sizes are incompatible, NumPy throws an **error**.

Example of an **error:**

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

print(arr1 + arr2)  # ERROR! Shapes are not compatible
```

---

7. Advanced Indexing & Boolean Masking

Boolean Masking

Used for filtering elements in an array based on conditions.

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

mask = arr > 25  # Condition check
```

```
print(mask)  # [False False True True True]


filtered = arr[mask]  # Apply mask
print(filtered)  # [30 40 50]
```

Fancy Indexing

You can select specific elements using an index list.

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


indices = [0, 3, 4]  # Select specific elements
print(arr[indices])  # [10 40 50]
```

---

8. Stacking & Splitting Arrays

Stacking (Combining arrays)

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


# Vertical stacking
print(np.vstack((a, b)))
# [[1 2 3]
#  [4 5 6]]


# Horizontal stacking
print(np.hstack((a, b)))
# [1 2 3 4 5 6]
```

Splitting (Dividing arrays)

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


split_arr = np.split(arr, 3)
print(split_arr)
# [array([1, 2]), array([3, 4]), array([5, 6])]
```

---

9. Linear Algebra with NumPy

Dot Product (Matrix Multiplication)

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

dot_product = np.dot(A, B)  # Matrix multiplication
print(dot_product)
```

Determinant & Inverse

```
from numpy.linalg import det, inv

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

print(det(matrix))  # Determinant
print(inv(matrix))  # Inverse
```

---

10. Random Number Generation & Statistical Functions

NumPy has powerful random number functions.

Generating Random Numbers

```
# Random number between 0 and 1
print(np.random.rand(3))

# Random integers
print(np.random.randint(1, 10, (2, 2)))
```

Statistical Functions

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

print(np.mean(arr))  # Mean
print(np.median(arr))  # Median
print(np.std(arr))  # Standard deviation
```

---

11. File I/O (Reading & Writing Data)

Saving and Loading NumPy Arrays

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

# Save as .npy file
np.save("my_array.npy", arr)

# Load the saved file
loaded_arr = np.load("my_array.npy")
print(loaded_arr)
```

Reading and Writing CSV Files

```
# Save array as CSV
np.savetxt("data.csv", arr, delimiter=",")

# Load from CSV
loaded_csv = np.loadtxt("data.csv", delimiter=",")
print(loaded_csv)
```

---

12. Performance Optimization (Vectorization)

NumPy is **faster than loops** because it operates on entire arrays at once.

Slow Python Loop

```
numbers = [1, 2, 3, 4, 5]
squared = [x**2 for x in numbers]  # Loop-based
print(squared)
```

Fast NumPy Vectorized Operation

```
arr = np.array([1, 2, 3, 4, 5])
squared = arr**2  # NumPy vectorized operation
print(squared)
```