Introduction to NumPy

What is NumPy?

NumPy (Numerical Python), is a powerful open-source library for the Python programming language that provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays. It was created in 2005 by Travis Oliphant and has since become a fundamental package for scientific computing in Python.

NumPy is widely used in data analysis and manipulation, machine learning and deep learning, scientific simulations and modeling, etc.

In this lecture, we will explore the key features of this library, including ndarrays, mathematical functions and its performance.

NumPy Arrays

A NumPy array, specifically referred to as ***ndarray***, is a powerful N-dimensional array object provided by the NumPy library in Python. It is designed for efficient storage and manipulation of large datasets and is a fundamental data structure for scientific computing in Python.
NumPy arrays are stored in contiguous memory locations, which enhances performance and allows for efficient computation.

Here are the key characteristics of NumPy arrays:

- **Homogeneous Data**: All elements in a NumPy array must be of the same data type, which allows for optimized memory usage and performance.
- **Multi-dimensional**: NumPy arrays can be one-dimensional (vectors), two-dimensional (matrices), or even multi-dimensional (tensors), enabling complex data representations.

NumPy Arrays

NumPy arrays are a more efficient and powerful alternative to Python lists for numerical data processing, for several reasons:

- **Performance**: NumPy arrays are significantly faster than Python lists for numerical operations, due to their optimized implementation.
- **Memory Efficiency**: NumPy arrays consume less memory compared to Python lists. Since they store data of the same type, they require less overhead than lists.
- **Functionality**: NumPy provides a wide range of mathematical functions and operations that can be performed on arrays. This functionality is not natively available with Python lists.
- **Broadcasting**: NumPy supports broadcasting, which allows for arithmetic operations between arrays of different shapes, making it easier to perform calculations without needing to manually reshape data.

Creating NumPy Arrays

To use the NumPy library, we will first import it.

```
import numpy as np
```

There are several ways we can create ndarrays in Python:

1. Using ***np.array()***: This function creates an array from a list, tuple, or any other sequence.
2. Using ***np.zeros()***: This function creates an array of shape *(num_of_rows, num_of_columns)* filled with zeros.

```
# 1. Using np.array()
array_from_list = np.array([1, 2, 3, 4, 5])
print("Array from list:", array_from_list)

# 2. Using np.zeros()
zeros_array = np.zeros((3, 4))  # 3 rows and 4 columns
print("Array of zeros:\n", zeros_array)
```

```
Array from list: [1 2 3 4 5]
Array of zeros:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

Creating NumPy Arrays

3. Using ***np.ones()***: This function creates an array of shape *(num_of_rows, num_of_columns)* filled with ones.
4. Using ***np.arange()***: This function creates an array with a specified range of values. It is similar to Python's *range()* function, and takes *(start, stop, step)* as arguments.

These methods provide a concise way to create NumPy arrays with different initial values and shapes, making it easy to work with numerical data in Python.

```python
# 3. Using np.ones()
ones_array = np.ones((2, 3))  # 2 rows and 3 columns
print("Array of ones:\n", ones_array)

# 4. Using np.arange()
range_array = np.arange(0, 10, 2)  # Start at 0, stop before 10, step by 2
print("Array with range:", range_array)
```

```
Array of ones:
 [[1. 1. 1.]
 [1. 1. 1.]]
Array with range: [0 2 4 6 8]
```

NumPy Array Attributes

Here are the main attributes of a NumPy array:

- **Size**: determines the total number of elements in an array.
- **Shape**: A tuple representing the dimensions of the array. For a 1D array, it will be a tuple with a single value. For a 2D array, it will be a tuple with two values, and so on.
- **dtype**: The data type of the elements in the array, such as *int*, *float*, *bool*, etc.

```python
# Create a 1D array
arr1d = np.array([1, 2, 3, 4, 5, 6])
print("1D Array:", arr1d)
print("Size:", arr1d.size)
print("Shape:", arr1d.shape)
print("Data type:", arr1d.dtype)
```

```
1D Array: [1 2 3 4 5 6]
Size: 6
Shape: (6,)
Data type: int64
```

The ***reshape()*** function is used to change the shape of an array without changing its data. It takes a tuple as an argument specifying the new shape.
Now, we will reshape the array we previously created.

```python
# Reshape the 1D array to a 2D array
arr2d = arr1d.reshape(2, 3)
print("\n2D Array:\n", arr2d)
print("Size:", arr2d.size)
print("Shape:", arr2d.shape)
print("Data type:", arr2d.dtype)
```

```
2D Array:
 [[1 2 3]
 [4 5 6]]
Size: 6
Shape: (2, 3)
Data type: int64
```

Basic Array Operations

1. Arithmetic Operations

Now, we will explore different arithmetic operations that can be applied on NumPy arrays.
For the further examples, we will use the two following arrays:

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

We can apply Addition, Subtraction, Multiplication, Division, etc. Here are a few examples:

```
# Addition
result_add = arr1 + arr2
print("Addition:", result_add)

# Subtraction
result_sub = arr1 - arr2
print("Subtraction:", result_sub)

# Multiplication
result_mul = arr1 * arr2
print("Multiplication:", result_mul)

# Division
result_div = arr1 / arr2
print("Division:", result_div)

# Exponentiation
result_exp = arr1 ** 2
print("Exponentiation:", result_exp)

# Absolute value
result_abs = np.abs(arr1 - 4)
print("Absolute value:", result_abs)
```

```
Addition: [ 5  7  9]
Subtraction: [-3 -3 -3]
Multiplication: [ 4 10 18]
Division: [0.25 0.4  0.5 ]
Exponentiation: [ 1  4  9]
Absolute value: [3 2 1]
```

Basic Array Operations

2. Scalar Operations

Now, we will explore different scalar operations that can be applied on NumPy arrays.
For the further examples, we will use the following array:

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

We can apply Addition, Subtraction, Multiplication, Division, etc. Here are a few examples:

```
# Scalar addition
scalar_add = arr + 10
print("Scalar Addition:", scalar_add)

# Scalar subtraction
scalar_sub = arr - 2
print("Scalar Subtraction:", scalar_sub)

# Scalar multiplication
scalar_mul = arr * 3
print("Scalar Multiplication:", scalar_mul)

# Scalar division
scalar_div = arr / 2
print("Scalar Division:", scalar_div)

# Scalar exponentiation
scalar_exp = arr ** 2
print("Scalar Exponentiation:", scalar_exp)
```

```
Scalar Addition: [11 12 13 14 15]
Scalar Subtraction: [ -1   0   1   2   3]
Scalar Multiplication: [ 3  6  9 12 15]
Scalar Division: [0.5 1.  1.5 2.  2.5]
Scalar Exponentiation: [ 1  4  9 16 25]
```

Basic Array Operations

3. Indexing and Slicing

Basic **indexing** allows us to access individual elements of a NumPy array using their integer indices.

Slicing allows us to extract a range of elements from an array. The syntax for slicing is *array[start:stop:step]*, where:

- start: The starting index (inclusive).
- stop: The ending index (exclusive).
- step: The interval between elements (optional).

```
# Create a 2D array
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing elements in a 2D array
print("Element at (1, 2):", arr2d[1, 2])  # Element in the second row, third column
```

```
Element at (1, 2): 6
```

```
# Create a 1D array
arr1d = np.array([10, 20, 30, 40, 50])
print("Every second element from index 0 to 5:", arr1d[0:5:2])  # Elements at indices 0, 2, 4
```

```
Every second element from index 0 to 5: [10 30 50]
```

Basic Array Operations

4. Boolean Indexing

Boolean indexing in NumPy allows us to select elements from an array based on conditions. We first create a condition that returns a Boolean array, where each element corresponds to whether the condition is True or False for the respective element in the original array. Next, we use this Boolean array to index the original array, resulting in a new array that contains only the elements where the condition is True. Here are a few examples:

```
# Create a 1D array
arr = np.array([10, 20, 30, 40, 50, 60, 70])

# Create a boolean condition: select elements greater than 30
condition = arr > 30

# Print the boolean array
print("Boolean condition (arr > 30):", condition)

# Use boolean indexing to get elements that satisfy the condition
filtered_arr = arr[condition]
print("Filtered array (elements > 30):", filtered_arr)
```

```
Boolean condition (arr > 30): [False False False  True  True  True  True]
Filtered array (elements > 30): [40 50 60 70]
```

Basic Array Operations

4. Boolean Indexing

Here is an example if Boolean indexing for a 2D array:

```python
# Create a 2D array
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Create a boolean condition: select elements greater than 5
condition_2d = arr2d > 5

# Print the boolean array
print("Boolean condition (arr2d > 5):\n", condition_2d)

# Use boolean indexing to get elements that satisfy the condition
filtered_arr2d = arr2d[condition_2d]
print("Filtered array (elements > 5):", filtered_arr2d)
```

```
Boolean condition (arr2d > 5):
 [[False False False]
 [False False  True]
 [ True  True  True]]
Filtered array (elements > 5): [6 7 8 9]
```

Universal functions

Universal functions, commonly referred to as **ufuncs**, are a core feature of the NumPy library. They are designed to perform element-wise operations on NumPy arrays efficiently and flexibly.
A few examples of universal functions include ***np.add()***, ***np.subtract()***, ***np.multiply()***, ***np.sin()***, ***np.tan()***, etc.
Here are a few examples:

```
# Create two arrays for arithmetic operations
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])

# Using ufuncs for addition
result_add = np.add(array1, array2)
print("Addition result:", result_add)
```

```
Addition result: [5 7 9]
```

```
# Create an array of angles in radians
angles = np.array([0, np.pi/2, np.pi])

# Use ufuncs to compute sine and cosine
sine_values = np.sin(angles)
cosine_values = np.cos(angles)

print("Angles (radians):", angles)
print("Sine values:", sine_values)
print("Cosine values:", cosine_values)
```

```
Angles (radians): [0.         1.57079633 3.14159265]
Sine values: [0. 1. 0.]
Cosine values: [ 1.  0. -1.]
```

Array Manipulation

In NumPy, ***np.concatenate()*** and ***np.split()*** are functions used to combine and divide arrays, respectively. These functions are essential for manipulating and organizing data in various applications.

The *np.concatenate()* function is used to join two or more arrays along a specified axis. It can be particularly useful for combining datasets or merging results. The ***axis*** keyword determines the axis along which the arrays will be joined. The default is 0, which means the arrays will be concatenated vertically (row-wise).

If axis=1, they will be concatenated horizontally (column-wise).
Here is an example of concatenation:

```python
# Create two 2D arrays
arr3 = np.array([[1, 2, 3], [4, 5, 6]])
arr4 = np.array([[7, 8, 9], [10, 11, 12]])

# Concatenate the 2D arrays vertically
concatenated_2d_vertical = np.concatenate((arr3, arr4), axis=0)
print("\nConcatenated 2D array (vertical):\n", concatenated_2d_vertical)

# Concatenate the 2D arrays horizontally
concatenated_2d_horizontal = np.concatenate((arr3, arr4), axis=1)
print("\nConcatenated 2D array (horizontal):\n", concatenated_2d_horizontal)
```

```
Concatenated 2D array (vertical):
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Concatenated 2D array (horizontal):
 [[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

Array Manipulation

The *np.split()* function is used to divide an array into multiple sub-arrays along a specified axis. This can be useful for data segmentation or partitioning datasets.

- **array**: The array to be split.
- **indices_or_sections**: If an integer, it indicates the number of equal sections to split the array into. If an array of indices, it specifies the indices at which to split.
- **axis**: The axis along which to split the array. The default is 0.

```
np.split(array, indices_or_sections, axis=0)
```

```
# Create a 2D array
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])

# Split the 2D array into 2 equal parts along the first axis (rows)
split_2d = np.split(arr2d, 2)
print("\nSplit 2D array (into 2 parts):")
for i, part in enumerate(split_2d):
    print(f"Part {i}:\n{part}")
```

```
Split 2D array (into 2 parts):
Part 0:
[[1 2 3]
 [4 5 6]]
Part 1:
[[ 7  8  9]
 [10 11 12]]
```

Aggregation Functions

NumPy provides a variety of aggregation functions that allow us to compute summary statistics for arrays. Here are a few of the main aggregation functions:

- ***np.sum()***: Computes the sum of elements.
- ***np.mean()***: Finds the arithmetic mean (average).
- ***np.median()***: Computes the median mean(middle value).
- ***np.min()***: Returns the minimum value in the array.
- ***np.max()***: Returns the maximum value in the array.
- ***np.std()***: Computes the standard deviation.
- ***np.var()***: Computes the variance.

Here is an example demonstrating these functions:

```
# Create a 1D array
arr1d = np.array([10, 20, 30, 40, 50])
```

Aggregation Functions

```python
# Sum of elements
sum_result = np.sum(arr1d)
print("Sum:", sum_result)

# Mean of elements
mean_result = np.mean(arr1d)
print("Mean:", mean_result)

# Median of elements
median_result = np.median(arr1d)
print("Median:", median_result)

# Minimum value
min_result = np.min(arr1d)
print("Minimum:", min_result)
```

```python
# Maximum value
max_result = np.max(arr1d)
print("Maximum:", max_result)

# Standard deviation
std_result = np.std(arr1d)
print("Standard Deviation:", std_result)

# Variance
var_result = np.var(arr1d)
print("Variance:", var_result)
```

```
Sum: 150
Mean: 30.0
Median: 30.0
Minimum: 10
Maximum: 50
Standard Deviation: 14.142135623730951
Variance: 200.0
```

Sorting

In NumPy, sorting is an essential operation that allows us to organize data efficiently.

The ***np.sort()*** function returns a sorted copy of an array. By default, it sorts the array in ascending order. We can specify the axis along which to sort and the sorting algorithm to use.
The ***np.argsort()*** function returns the indices that would sort an array, allowing us to retrieve the original values in sorted order.

```python
# Create an array
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5])

# Sort the array
sorted_arr = np.sort(arr)
print("Sorted array:", sorted_arr)

# Sort a 2D array along a specific axis
arr2d = np.array([[3, 1, 2], [6, 5, 4]])
sorted_2d = np.sort(arr2d, axis=1)  # Sort each row
print("Sorted 2D array:\n", sorted_2d)
```

```
Sorted array: [1 1 2 3 3 4 5 5 5 6 9]
Sorted 2D array:
 [[1 2 3]
 [4 5 6]]
```

```python
# Get the indices that would sort the array
indices = np.argsort(arr)
print("Indices that would sort the array:", indices)
```

```
Indices that would sort the array: [1 3 2 0 5 8 6 9 4 7 10]
```

Searching

In NumPy, searching is an essential operation that allows us to organize data efficiently.

The ***np.searchsorted()*** function finds indices where elements should be inserted to maintain order in a sorted array. It is particularly useful for binary search operations.
The ***np.where()*** function can be used to find indices of elements that satisfy a certain condition.

```
# Create a sorted array
sorted_arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Search for the index to insert a value
index = np.searchsorted(sorted_arr, 5.5)
print("Index to insert 5.5:", index)
```

```
Index to insert 5.5: 5
```

```
# Find indices of elements greater than 5
indices_greater_than_5 = np.where(arr > 5)
print("Indices of elements greater than 5:", indices_greater_than_5)
```

```
Indices of elements greater than 5: (array([5, 6, 9]),)
```