NUMPY ASSESSMENT

1. NumPy (Numerical Python) is a fundamental library in Python used extensively for scientific computing. Its purpose is to provide efficient operations for working with arrays, matrices, and numerical data.

   It's main purposes are,

   - Efficient array processing
   - Mathematical and statistical functions
   - Data preprocessing

   Advantages of Numpy,

   - NumPy can handle multi-dimensional arrays
   - NumPy arrays are more compact and use less memory compared to Python lists for storing large datasets
   - NumPy performs vectorized operations, which means that you can apply operations to whole arrays without writing explicit loops.
   - NumPy can interface with external libraries written in C, C++, or Fortran
   - Simplifies Linear Algebra and Matrix Operation
   - Active community

---

2 . Both np.mean() and np.average() are used for calculating the average of arithmetic means in an array.

The only difference lies in the way they handle weights.

np.mean():

- Calculates the arithmetic mean (average) of the elements in an array.
- Does **not** support weights. It simply computes the sum of all elements divided by the number of elements
- Usage : Best used when you want a straightforward mean of a dataset.

Ex: np.mean([1, 2, 3]) # Output: 2.0

Np.average():

- Computes a weighted average if weights are provided, otherwise behaves like `np.mean()`.
- You can pass a weights argument to calculate the weighted average.
- Usage : Ideal when you need to compute an average where certain values contribute more (weighted average).
  Ex: np.average([1, 2, 3], weights=[0.1, 0.3, 0.6]) # Output: 2.5

---

3. Reversing a NumPy array can be done along different axes using slicing or built-in functions like np.flip(). Here are the methods for reversing along various axes with examples for both 1D and 2D arrays:

1. Reversing a 1D Array

For a 1D array, reversing means changing the order of elements.

Method 1: Slicing

You can reverse a 1D array using slicing [::-1].

```
import numpy as np
arr_1d = np.array([1, 2, 3, 4, 5])
reversed_1d = arr_1d[::-1]
print(reversed_1d)  # Output: [5 4 3 2 1]
```

Method 2: np.flip()

Alternatively, you can use np.flip() to reverse the array.

```
reversed_1d_flip = np.flip(arr_1d)
print(reversed_1d_flip)  # Output: [5 4 3 2 1]
```

2. Reversing a 2D Array

In a 2D array, we can reverse along different axes (rows, columns, or both).

Method 1: Reversing Rows (Axis 0)

To reverse the rows of a 2D array, you can slice along axis 0.

```
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
reversed_rows = arr_2d[::-1, :]
print(reversed_rows)
# Output:
# [[7 8 9]
#  [4 5 6]
#  [1 2 3]]
```

Or, use np.flip() along axis 0.

python
Copy code
```
reversed_rows_flip = np.flip(arr_2d, axis=0)
print(reversed_rows_flip)
# Output is the same as above
```

Method 2: Reversing Columns (Axis 1)

To reverse the columns of a 2D array, you can slice along axis 1.

python
Copy code
```
reversed_columns = arr_2d[:, ::-1]
print(reversed_columns)
# Output:
# [[3 2 1]
#  [6 5 4]
#  [9 8 7]]
```

Or, use np.flip() along axis 1.

```
reversed_columns_flip = np.flip(arr_2d, axis=1)
print(reversed_columns_flip)
# Output is the same as above
```

Method 3: Reversing Both Rows and Columns

You can reverse both rows and columns by combining the slices.

```
reversed_both = arr_2d[::-1, ::-1]

print(reversed_both)
```

```
# Output:
# [[9 8 7]
#  [6 5 4]
#  [3 2 1]]
```

Or, use np.flip() on both axes.

```
reversed_both_flip = np.flip(arr_2d, axis=(0, 1))
print(reversed_both_flip)
```

---

4. We can determine the data type of elements in a NumPy array using the .dtype attribute. This tells you the data type of the array's elements.

Ex:

```
import numpy as np

arr = np.array([1, 2, 3])
print(arr.dtype)  # Output: int64
```

Importance of Data Types in Memory Management and Performance

1. **Memory Efficiency**:
    - **Memory Size**: Different data types consume different amounts of memory. For example, int8 (8-bit integers) takes 1 byte per element, whereas int64 (64-bit integers) takes 8 bytes per element. Choosing the correct data type ensures that you're not wasting memory by using larger types unnecessarily.

    ```
    arr_uint8 = np.array([1, 2, 3], dtype=np.uint8)
    print(arr_uint8.nbytes)  # Output: 3 bytes
    arr_int32 = np.array([1, 2, 3], dtype=np.int32)
    print(arr_int32.nbytes)  # Output: 12 bytes
    ```

2. **Performance Optimization**:
    - **Processing Speed**: Smaller data types allow for faster computation because smaller data types require fewer CPU cycles to load and process. Using the smallest data type that meets the precision requirements enhances performance.

   - **Vectorized Operations**: NumPy is optimized for array operations, and using the appropriate data type allows the underlying C code to run faster. Operations on float64 data are generally slower than on float32 because of the larger memory and computational overhead.
3. **Precision and Accuracy**:
   - Choosing the wrong data type can lead to loss of precision. For example, using int32 for very large numbers may cause overflow, whereas float32 might introduce rounding errors for highly precise floating-point numbers.
4. **Type Compatibility**:
   - When performing operations on arrays of different data types, NumPy automatically upcasts the smaller data type to the larger one, which might cause unintended memory consumption.

---

5. NumPy ndarray Definition

In NumPy, an ndarray (n-dimensional array) is the core data structure used to represent and store multi-dimensional arrays. It can hold a collection of elements, all of the same type, indexed by a tuple of positive integers representing the array's dimensions (or axes).

Key Features of NumPy ndarray:

1. **Multi-dimensional**:
   - An ndarray can represent arrays with any number of dimensions (1D, 2D, 3D, etc.). Each dimension is referred to as an **axis**.
   - Example: A 2D array has two axes: rows and columns.
2. **Homogeneous Data**:
   - All elements in a NumPy array must be of the same data type (e.g., integers, floats, etc.). This ensures memory efficiency and speeds up operations.
3. **Fixed Size**:
   - Once created, the size of an ndarray is fixed, meaning the number of elements in each dimension cannot change. This contrasts with Python lists, which can be resized dynamically.
4. **Fast Element-wise Operations**:

- NumPy allows for **vectorized operations** (element-wise operations) without explicit loops. These operations are highly optimized for performance.
- Example: Adding two ndarrays adds the corresponding elements.

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = a + b  # Output: [5, 7, 9]
```

5. **Memory Efficiency**:
   - NumPy arrays are stored in contiguous memory blocks, unlike Python lists, allowing fast access to elements and efficient use of memory.
   - **Data Types**: Arrays store elements in compact data types (e.g., int32, float64), reducing memory usage compared to Python's general object types.
6. **Advanced Indexing and Slicing**:
   - NumPy arrays support advanced slicing and indexing mechanisms for extracting or manipulating data, including boolean indexing, multi- arr = np.array([[1, 2], [3, 4], [5, 6]])

```
print(arr[1:, :])  # Slices the array, Output: [[3, 4], [5, 6]]
```

7. **Broadcasting**:
   - NumPy allows operations on arrays of different shapes using a feature called **broadcasting**. This simplifies operations where arrays are not the same size but can still interact in a meaningful way.

```
a = np.array([1, 2, 3])
b = np.array([[1], [2], [3]])
c = a + b  # Broadcasting adds arrays, Output: [[2, 3, 4], [3, 4, 5], [4, 5, 6]]
```

8. **Support for Mathematical Functions**:
   - NumPy provides a wide range of mathematical functions, like linear algebra, Fourier transforms, and random number generation, optimized for ndarray operations.
9. **Efficient Handling of Large Datasets**:

    - NumPy arrays are optimized for handling large datasets, making them ideal for scientific computing, machine learning, and data processing.

Differences Between NumPy ndarray and Python Lists:

1. **Homogeneity**:
    - **ndarray**: All elements must have the same data type (e.g., int32, float64).
    - **Python List**: Can hold elements of different types (e.g., integers, strings, floats).
2. **Memory Efficiency**:
    - **ndarray**: Uses contiguous memory blocks, resulting in lower memory overhead and faster access.
    - **Python List**: Lists are arrays of pointers to objects, leading to higher memory overhead.
3. **Performance**:
    - **ndarray**: NumPy operations are much faster, as they leverage low-level C and Fortran libraries, and support vectorized operations.
    - **Python List**: Operations on lists require loops, which are slower compared to NumPy's optimized routines.
4. **Fixed Size vs. Dynamic**:
    - **ndarray**: Once created, the size of an ndarray is fixed.
    - **Python List**: Lists are dynamic; you can append or remove elements, changing the size of the list.
5. **Advanced Operations**:
    - **ndarray**: Supports advanced mathematical operations, broadcasting, and slicing, designed for large-scale numerical computing.
    - **Python List**: Lacks built-in support for complex mathematical functions and requires manual iteration for operations.
6. **Data Slicing**:
    - **ndarray**: Supports multi-dimensional slicing, allowing you to access elements across multiple axes.
    - **Python List**: Supports basic slicing for 1D lists but is less flexible for multi-dimensional data.

Example:

NumPy ndarray:

```
import numpy as np
arr = np.array([1, 2, 3, 4])  # NumPy array
print(arr + 2)  # Output: [3 4 5 6] (vectorized operation)
```

Python List:

```
lst = [1, 2, 3, 4]  # Python list
print([x + 2 for x in lst])  # Output: [3, 4, 5, 6] (manual loop)
```

---

6 . NumPy arrays (`ndarray`) offer significant performance benefits over Python lists, particularly for large-scale numerical operations:

1. Memory Efficiency: NumPy arrays store data in contiguous memory blocks and use fixed data types, which reduce memory overhead. Python lists are arrays of pointers to objects, leading to higher memory consumption.

2. Faster Operations: NumPy uses vectorized operations, allowing entire arrays to be processed without explicit loops. This makes NumPy much faster than Python lists, which require slower, interpreted loops.

3. Optimized Mathematical Functions: NumPy’s functions (e.g., `np.sum()`, `np.dot()`) are optimized for speed, bypassing the Python interpreter and using low-level C/Fortran code.

4. Broadcasting: NumPy can perform operations on arrays of different shapes efficiently without duplicating data, saving both time and memory. Python lists require manual loops for such operations.

5. Multi-dimensional Support: NumPy handles multi-dimensional arrays (e.g., 2D, 3D) efficiently, whereas Python lists require nested structures, which are slower and more complex to work with.

Overall, NumPy arrays provide better performance, memory optimization, and computational speed, especially for large-scale numerical tasks, compared to Python lists.

---

7. In NumPy, vstack() and hstack() are used to stack arrays vertically and horizontally, respectively. Here's a comparison of the two with examples:

1. np.vstack() (Vertical Stack)

- **Purpose**: Stacks arrays **vertically** along the first axis (row-wise).
- **Requirements**: The arrays must have the same number of columns to align properly.

Example:

```
import numpy as np

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

# Using vstack() to vertically stack
vstack_result = np.vstack((a, b))
print(vstack_result)
```

Output:

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

- Here, a and b are stacked on top of each other to form a 2D array with two rows and three columns.

Example with 2D arrays:

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

vstack_result = np.vstack((a, b))
```

```
print(vstack_result)
```

Output:

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

- The arrays are stacked vertically to create a new array with four rows and three columns.

2. np.hstack() (Horizontal Stack)

- **Purpose**: Stacks arrays **horizontally** along the second axis (column-wise).
- **Requirements**: The arrays must have the same number of rows to align properly.

Example:

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

# Using hstack() to horizontally stack
hstack_result = np.hstack((a, b))
print(hstack_result)
```

Output:

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

- Here, a and b are stacked horizontally, resulting in a 1D array with six elements.

Example with 2D arrays:

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

hstack_result = np.hstack((a, b))
print(hstack_result)
```

Output:

```
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

- The arrays are stacked horizontally to form a new array with two rows and six columns.

8. In NumPy, flipud() and fliplr() are two functions used to flip arrays along different axes. Both perform array reversal, but they act on different dimensions:

1. np.flipud() (Flip Up-Down)

- **Purpose**: Flips the array **vertically** (i.e., up to down).
- **Effect**: Reverses the order of rows in a 2D array or higher-dimensional arrays along the first axis (axis 0).

Example with 2D array:

```
import numpy as np

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

# Flipping the array upside down using flipud()
flipud_result = np.flipud(arr)
print(flipud_result)
```

Output:

```
[[7 8 9]
 [4 5 6]
 [1 2 3]]
```

- The rows are flipped vertically: the first row becomes the last, and the last row becomes the first.

Example with 1D array:

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

```
flipud_result = np.flipud(arr)
print(flipud_result)
```

Output:

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

- In the case of a 1D array, it behaves like a standard reversal.

---

2. np.fliplr() (Flip Left-Right)

- **Purpose**: Flips the array **horizontally** (i.e., left to right).
- **Effect**: Reverses the order of columns in a 2D array or higher-dimensional arrays along the second axis (axis 1).

Example with 2D array:

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

# Flipping the array left to right using fliplr()
fliplr_result = np.fliplr(arr)
print(fliplr_result)
```

Output:

```
lua
Copy code
[[3 2 1]
 [6 5 4]
 [9 8 7]]
```

- The columns are flipped horizontally: the first column becomes the last, and the last column becomes the first.

Example with 1D array:

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

# Attempting to use fliplr on a 1D array will raise an error
# fliplr_result = np.fliplr(arr)  # Uncommenting this will raise ValueError
```

- fliplr() requires at least a 2D array. It will raise a ValueError if applied to a 1D array because there are no columns to flip.

Effect on Higher-dimensional Arrays (3D Arrays):

For higher-dimensional arrays, flipud() and fliplr() will operate on the first and second axes, respectively.

Example with 3D array:

```
arr = np.array([[[1, 2, 3], [4, 5, 6]],
             [[7, 8, 9], [10, 11, 12]]])

# Using flipud()
flipud_result = np.flipud(arr)
print(flipud_result)
```

Output:

```
 [[[ 7  8  9]
  [10 11 12]]

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

- flipud() flips the 2D sub-arrays along the first axis (i.e., flips the "depth" dimension in a 3D array).

Example with fliplr() on 3D:

```
fliplr_result = np.fliplr(arr)
print(fliplr_result)
```

Output:

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

 [[10 11 12]
 [ 7  8  9]]]
```

- fliplr() flips the 2D sub-arrays along the second axis (i.e., flips left to right).

---

9 . The numpy.array_split() method is used to split an array into multiple sub-arrays. Unlike np.split(), which requires equal splits, array_split() can handle **uneven splits**, making it more flexible.

Functionality of array_split():

- It takes an array and splits it into the specified number of sub-arrays. If the array cannot be split evenly, it distributes the remainder as evenly as possible across the resulting sub-arrays.
- **Syntax**:

  python
  Copy code
  ```
  np.array_split(array, indices_or_sections, axis=0)
  ```

  - array: The array to be split.
  - indices_or_sections: The number of sub-arrays to return, or a list of indices where the array should be split.
  - axis: The axis along which to split (default is 0).

Handling Uneven Splits:

If the array cannot be split evenly, array_split() creates sub-arrays with as equal a size as possible. The first few sub-arrays will have one more element than the others.

Example:

```
import numpy as np

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

# Split into 3 sub-arrays
```

```
split_result = np.array_split(arr, 3)
print(split_result)
```

Output:

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

- Since 7 elements cannot be evenly split into 3 parts, the first sub-array has 3 elements, and the other two have 2 elements each.

Example with a 2D array:

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

# Split into 3 sub-arrays along axis 0 (rows)
split_result = np.array_split(arr, 3)
print(split_result)
```

Output:

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

- The split is uneven: one sub-array has 2 rows, and the other two have 1 row each

---

10.

1. Vectorization in NumPy

- **Definition**: Vectorization refers to the process of applying operations to entire arrays (or "vectors") without using explicit loops, making use of low-level C implementations for optimal performance.
- **How it Works**: Instead of applying operations element-by-element with for loops (as in traditional Python), NumPy applies them to entire arrays at once. This is achieved via vectorized operations, where NumPy functions operate on the entire array in a single call.

Example of Vectorization:

Without vectorization (using loops):

```
import numpy as np

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

for i in range(len(arr)):
    result[i] = arr[i] * 2  # Doubling each element

print(result)
```

With vectorization:

```
result = arr * 2  # Vectorized operation
print(result)
```

Output:

```
[2 4 6 8]
```

- **Performance Benefit**: Vectorization avoids the overhead of Python loops, using optimized C code under the hood. This results in significant speed-ups, especially for large arrays.

2. Broadcasting in NumPy

- **Definition**: Broadcasting is a powerful feature in NumPy that allows operations on arrays of different shapes and sizes by "stretching" or "broadcasting" the smaller array so that it matches the shape of the larger one, without actually copying data.
- **How it Works**: NumPy automatically adjusts the dimensions of arrays during operations. It matches dimensions either by replicating the smaller array along the missing dimensions or by treating a size of 1 as expandable.

Broadcasting Example:

```
a = np.array([1, 2, 3])  # Shape (3,)
b = np.array([[10], [20], [30]])  # Shape (3, 1)

result = a + b
```

```
print(result)
```

Output:

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

- Here, a is broadcasted across the columns to match the shape of b. Broadcasting expands the smaller array in such a way that operations are carried out without duplicating data.

Broadcasting Rules:

1. If the arrays differ in the number of dimensions, prepend 1s to the smaller array's shape until they match.
2. If the shapes of two arrays are compatible, they must be equal in corresponding dimensions or one of them must be 1.
3. Arrays are broadcasted in such a way that the smaller array is virtually replicated to match the larger array.

Example of Incompatible Shapes:

python
Copy code

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

# This will raise a ValueError as shapes are incompatible
result = a + b
```

Contribution to Efficiency:

- **Avoiding Loops**: Both vectorization and broadcasting eliminate the need for Python loops, leading to faster execution since NumPy operations are written in C, Fortran, or assembly language, which are much faster than Python’s interpreted code.
- **Memory Efficiency**: Broadcasting performs operations without making copies of data, reducing memory overhead. It allows computations on arrays of different shapes without creating large temporary arrays.
- **Simplicity**: Vectorized and broadcasted operations result in cleaner, more readable code that is easier to maintain and less error-prone.

Performance Comparison Example:

```
import numpy as np
import time

# Large arrays
a = np.random.rand(1000000)
b = np.random.rand(1000000)

# Using loops (without vectorization)
start_time = time.time()
result = np.zeros_like(a)
for i in range(len(a)):
    result[i] = a[i] + b[i]
print("Time without vectorization:", time.time() - start_time)

# Using vectorized operation
start_time = time.time()
result = a + b  # Vectorized addition
print("Time with vectorization:", time.time() - start_time)
```

Output:

```
Time without vectorization: 0.5 seconds

Time with vectorization: 0.001 seconds
```