Answers of theory Questions

1. **Explain the purpose and advantages of NumPy in scientific computing and data analysis. How does it enhance Python's capabilities for numerical operations?**

- Ans:-
    - **Purpose**: NumPy is designed to handle large arrays and matrices of numeric data efficiently. It provides powerful mathematical functions and operations essential for scientific and analytical tasks.
- **Advantages**:

Speed & Efficiency: Uses C under the hood.

Multidimensional arrays: Supports tensors and matrices.

Broadcasting & Vectorization: Replaces loops with fast array ops.

Extensive math functions: FFT, linear algebra, statistics.

Memory efficiency: Better than Python lists.

2.Compare and contrast np.mean() and np.average() functions in NumPy. When would you use one over the other? -- >

- np.mean(): 1)Arithmetic mean

  2)weight are not supported

  3)Example: np.mean(arr)

  4) Use np.mean() for regular averaging.

- np.average()

  1)Weighted average

  2)weights argument available

  3)Example np.average(arr, weights=w)

  4) Use np.average() when different elements contribute unequally.

3.Describe the methods for reversing a NumPy array along different axes. Provide examples for 1D and 2D arrays. -- > NumPy provides several methods

to reverse arrays along different axes. Here are the common techniques with examples for both 1D and 2D arrays:

1)Reversing a 1D Array For 1D arrays, you can use slicing with a step of -1:

import numpy as np

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

reversed_arr = arr[::-1] # Step -1 reverses the array

print(reversed_arr) # Output: [5 4 3 2 1]

Alternatively, you can use np.flip():

reversed_arr = np.flip(arr)

print(reversed_arr) # Output: [5 4 3 2 1]

2)Reversing a 2D Array For 2D arrays, you can reverse along different axes:

Reverse rows (axis 0):

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

1. Reverse the order of rows:

reversed_rows = arr_2d[::-1, :] or np.flip(arr_2d, axis=0) print(reversed_rows)

Output: [[7 8 9] [4 5 6] [1 2 3]] Reverse columns (axis 1):

Reverse the order of columns

reversed_cols = arr_2d[:, ::-1] # or np.flip(arr_2d, axis=1)

print(reversed_cols)

Output:

[[3 2 1]

[6 5 4]

[9 8 7]]

Reverse both rows and columns:

Reverse both rows and columns

reversed_both = arr_2d[::-1, ::-1] # or np.flip(arr_2d)

print(reversed_both)

Output:

[[9 8 7]

[6 5 4]

[3 2 1]]

- Using NumPy Functions NumPy provides specific functions for these operations:

- np.flip(array) - Reverses all axes
- np.flip(array, axis=0) - Reverses along rows (first axis)
- np.flip(array, axis=1) - Reverses along columns (second axis)
- np.flipud(array) - Equivalent to flip(axis=0) (up-down)

  np.fliplr(array) - Equivalent to flip(axis=1) (left-right)

4.How can you determine the data type of elements in a NumPy array? Discuss the importance of data types in memory management and performance.

- Determining Data Type in NumPy
- arr = np.array([1.0, 2.0])

print(arr.dtype) # float64

Importance of dtype:

Memory efficiency: Choose optimal size (int8, float32).

Speed: Smaller dtypes = faster operations.

Precision control: Crucial in scientific computing.

5 Define ndarrays in NumPy and explain their key features. How do they differ from standard Python lists? -->

- Definition: ndarray is the core data structure in NumPy for N-dimensional arrays.

- Key Features:
  - Fixed-size, homogeneous data
  - Fast vectorized ops
  - Supports multi-dimensional slicing
  - Methods for reshaping, aggregating, transforming

vs Python Lists:

- Faster and memory-efficient
- Supports broadcasting
- Enables mathematical ops (not possible directly with lists)

6.Analyze the performance benefits of NumPy arrays over Python lists for large-scale numerical operations.

-- >Example:

import numpy as np

arr = np.arange(1e6)

*%timeit arr * 2 # Vectorized*

- vs Python list

lst = list(range(int(1e6)))

*%timeit [x*2 for x in lst] # Much slower*

<u>Performance Benefits:</u>

Vectorization removes Python loop overhead.

Operations use optimized C code internally.

Uses less memory with fixed data types.

7. Compare vstack() and hstack() functions in NumPy. Provide examples demonstrating their usage and output.

-- > Difference:

vstack() stacks along rows (adds new rows).

hstack() stacks along columns (extends horizontally).

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

b = np.array([3, 4])

Vertical stack

np.vstack((a, b))

 # → [[1, 2],

 #   [3, 4]]

Horizontal stack

np.hstack((a, b))

 # → [1, 2, 3, 4]

8.Explain the differences between fliplr() and flipud() methods in NumPy, including their effects on various array dimensions.

fliplr(): Reverses columns

flipud(): Reverses rows

Useful in image processing, matrix transformations.

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

np.fliplr(arr) # Flip left to right

→ [[2, 1], [4, 3]]

np.flipud(arr) # Flip upside down

 # → [[3, 4], [1, 2]]

9. Discuss the functionality of the array_split() method in NumPy. How does it handle uneven splits?

- np.array_split() is used to divide a NumPy array into sub-arrays, even when the number of elements cannot be split equally.

If not, array_split() distributes the remainder across the first few sub-arrays, ensuring no error is thrown (unlike np.split() which fails on uneven division).

Example:

```
import numpy as np

arr = np.arange(10)
result = np.array_split(arr, 3)

for i, r in enumerate(result):
print(f"Part {i+1}: {r}")
```

10.Explain the concepts of vectorization and broadcasting in NumPy. How do they contribute to efficient array operations?

Vectorization

- Definition: Vectorization is the process of replacing explicit loops with array expressions to perform operations on entire arrays simultaneously.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
squared = arr ** 2  # Vectorized
# Output: [1 4 9 16]
```

Without vectorization (slower):

```
squared = [x**2 for x in [1, 2, 3, 4]]
```

- Broadcasting

- Definition: Broadcasting automatically expands smaller arrays so they can match the shape of larger arrays in arithmetic operations without actual data replication.

Rules:

Dimensions must match or be 1.

NumPy virtually expands the smaller array to perform operations.

Example:

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

b = np.array([10, 20, 30])   # shape (3,)

result = a + b

Output:

[[11 21 31]

[12 22 32]

[13 23 33]]

a expands to shape (3,3) to match b automatically.