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:- NumPy is a fundamental library in Python for scientific computing and data analysis. Its primary purpose is to efficiently handle large, multi-dimensional arrays and perform fast numerical operations. NumPy enhances Python by providing:

1. Speed and Performance: NumPy arrays are faster and more memory-efficient than Python lists due to their optimized C-based implementation.
2. Vectorized Operations: It allows operations on entire arrays without loops, boosting performance.
3. Support for Multi-Dimensional Arrays: Essential for complex data structures like matrices and tensors.
4. Advanced Mathematical Functions: Includes tools for linear algebra, statistics, and random number generation.
5. Interoperability: Integrates smoothly with other libraries like pandas, SciPy, and machine learning frameworks.

These features make NumPy ideal for numerical computations and large-scale data analysis.

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

#Ans:-
# np.mean():-

# Calculates the arithmetic mean of the elements along a specified
axis or of the entire array.
# It does not support weights, meaning all elements are treated
equally when calculating the mean.
# Typically used when you just need the simple mean of the data
without considering any weights.

#Example:-
import numpy as np
data = np.array([1, 2, 3, 4, 5])
mean_value = np.mean(data)
print(mean_value)

#np.average():-
# It also computes the average, but it allows you to assign weights to
the values.
# WIt supports a weights parameter, where each value is multiplied by
its corresponding weight before averaging.
# Use np.average() when you need to calculate a weighted average,
which is useful when certain values should contribute more to the
final result than others.

#Example:-
data = np.array([1, 2, 3, 4, 5])
```

```python
weights = np.array([1, 2, 3, 4, 5])  # Assigning different weights

weighted_avg = np.average(data, weights=weights)
print(weighted_avg)

# 3.3. Describe the methods for reversing a NumPy array along
different axes. Provide examples for 1D and 2D arrays.

#Ans:-

# 1.For 1D Arrays:-
# Use slicing to reverse the array:
#Example:-
import numpy as np
arr_1d = np.array([1, 2, 3, 4, 5])
reversed_1d = arr_1d[::-1]
print(reversed_1d)


# 2.For 2D Arrays:-
# Reverse Rows (axis=0):
#Example:-
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
reversed_rows = arr_2d[::-1, :]
print(reversed_rows)


# Reverse Columns (axis=1):
#Example:-
reversed_cols = arr_2d[:, ::-1]
print(reversed_cols)

#Reverse Both Axes:-
#Example:-
reversed_both = arr_2d[::-1, ::-1]
print(reversed_both)

# 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.

#Ans:-
# Determining the Data Type of a NumPy Array:-
# To determine the data type of elements in a NumPy array, we can use
the (dtype) attribute.
#Example:-
import numpy as np
arr = np.array([1, 2, 3])
print(arr.dtype)

# Importance of Data Types in Memory Management and Performance:-
```

```
# 1.Memory Management:
# Efficient Storage: NumPy arrays store data more compactly compared
to Python lists because they use fixed-size data types (like int32,
float64), which reduces memory consumption.
# Control over Precision: Choosing appropriate data types (e.g.,
float32 vs. float64) lets you manage the trade-off between precision
and memory usage.
#Smaller data types (e.g., int8, float32) use less memory than larger
ones (e.g., int64, float64).

#2.Performance:-
# Faster Computations: NumPy arrays are more efficient because
operations are performed using highly optimized C and Fortran
routines.
# Using fixed-size data types eliminates the overhead of Python's
dynamic typing.
# Vectorization: With fixed data types, NumPy can apply vectorized
operations (performing operations on entire arrays at once), which is
much faster than applying loops over Python lists.
```

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

Ans:- In NumPy, ndarrays (n-dimensional arrays) are the core data structure used for storing and manipulating large datasets. They are homogeneously-typed (all elements must be of the same data type) and can have any number of dimensions.

Key Features of NumPy ndarrays:-

- Multidimensional: An ndarray can have multiple dimensions (e.g., 1D, 2D, 3D, etc.), which allows it to represent more complex data like matrices, tensors, and higher-order structures.
- Homogeneous Data Types: All elements in an ndarray must have the same data type (e.g., all integers or all floats). This ensures consistency and makes memory allocation more efficient.
- Efficient Memory Usage: ndarrays store data in contiguous memory blocks, leading to better memory locality and allowing for faster access and operations compared to Python lists.
- Vectorized Operations: ndarrays support vectorized operations, meaning you can apply mathematical operations on entire arrays at once, without needing explicit loops. This boosts performance.
- Broadcasting: NumPy can perform operations on arrays of different shapes using broadcasting, where smaller arrays are "expanded" to match the shape of larger arrays.

- Built-in Functions and Methods: NumPy provides a wide range of built-in functions for mathematical operations, linear algebra, statistics, etc., which are optimized for use with ndarrays.

Differences Between ndarrays and Python Lists:-

*Data Type:- ndarray: All elements must be of the same data type (e.g., int32, float64). List: Can contain elements of different data types (e.g., integers, floats, strings, etc.).

*Performance:- ndarray: Optimized for numerical computations, offering faster performance due to efficient memory management and support for vectorized operations. List: Slower, especially for large datasets, due to the overhead of Python's dynamic typing and the need for manual iteration over elements.

Dimensionality:- ndarray: Supports multiple dimensions (1D, 2D, 3D, etc.). List: Typically represents only 1D data (though lists of lists can represent 2D data, they are not optimized for this purpose).

Memory Efficiency:- ndarray: Uses fixed-size, contiguous memory blocks for efficient memory usage. List: Stores references to objects in memory, which results in higher memory overhead and fragmentation.

Operations:- ndarray: Supports element-wise operations, broadcasting, and other mathematical functions without needing explicit loops. List: Requires explicit loops for element-wise operations, which can be less efficient.

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

#Ans:-
#NumPy arrays are more memory efficient compared to Python lists. They
store homogeneous data types in contiguous memory blocks,
#reducing memory overhead and improving memory utilization, especially
for large datasets.
#Lower Memory Consumption: NumPy arrays are more compact than Python
lists.

# 1. Efficient Memory Usage:-

# Fixed Data Type: NumPy arrays (ndarrays) store elements of the same
data type in contiguous memory blocks, which reduces memory overhead.
# In contrast, Python lists store references to objects, leading to
increased memory usage and fragmentation, especially for large
datasets.
#Compact Storage: NumPy's homogeneous, fixed-size data types (e.g.,
int32, float64) use less memory compared to the dynamic typing of
Python lists,
#which must store type and value information separately.

#2. Vectorized Operations:-
```

```python
#Element-wise Computation: NumPy supports vectorized operations,
allowing you to apply mathematical operations directly to entire
arrays without the need for explicit loops.
#This avoids the overhead of Python's interpreted loops and leads to
faster execution.
#Parallelization and Optimizations: Behind the scenes, NumPy uses
highly optimized C and Fortran routines that take advantage of low-
level CPU optimizations like Single Instruction Multiple Data (SIMD)
and multi-threading, leading to faster computations.

#Example Numpy Array:-
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
result = arr * 2  # Multiply all elements by 2 in a single operation

#Example python list:-
lst = [1, 2, 3, 4, 5]
result = [x * 2 for x in lst]

#3. Broadcasting:-

# Automatic Dimension Handling: NumPy’s broadcasting allows operations
between arrays of different shapes without manually reshaping or
expanding them.
# This feature streamlines computations and reduces the complexity of
code, while also avoiding unnecessary data duplication.

#Example:-
arr1 = np.array([1, 2, 3])
arr2 = np.array([[10], [20], [30]])
result = arr1 + arr2

#4. Optimized Built-in Functions:-

# Mathematical and Statistical Operations: NumPy provides a wide range
of optimized mathematical and statistical functions (e.g., mean, sum,
dot, sin) that are much faster than equivalent Python implementations
using loops or list comprehensions.
# These functions leverage low-level optimizations to perform
operations in a fraction of the time.

#Example:-
arr = np.random.rand(1000000)
mean_val = np.mean(arr)

# 5. Performance Comparison Example:-
# Let’s compare the time it takes to perform a basic operation
(element-wise addition) using both a NumPy array and a Python list.

#Example Numpy:-
```

```python
import numpy as np
import time
arr = np.arange(1000000)  # 1 million elements
start = time.time()
result = arr + arr  # Element-wise addition
end = time.time()
print(f"NumPy Time: {end - start:.6f} seconds")

#Example python list:-
lst = list(range(1000000))  # 1 million elements
start = time.time()
result = [x + x for x in lst]  # Element-wise addition using list
comprehension
end = time.time()
print(f"Python List Time: {end - start:.6f} seconds")

#6. Scalability:
#Large-Scale Data Processing: NumPy is designed to handle large
datasets efficiently. As the data size grows, the performance gap
between NumPy arrays and Python lists becomes more pronounced.
#This makes NumPy the preferred choice for large-scale numerical
computations in fields like machine learning, data analysis, and
scientific computing.



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

#Ans:-

#1. numpy.vstack()
# Purpose: Stacks arrays vertically (along rows).
# Input: Arrays must have the same number of columns.

#Example:-
import numpy as np

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

arr2 = np.array([[7, 8, 9],
                 [10, 11, 12]])

result = np.vstack((arr1, arr2))
print(result)

#2.numpy.hstack()
# Purpose: Stacks arrays horizontally (along columns).
# Input: Arrays must have the same number of rows.
```

```python
#Example:-
arr1 = np.array([[1, 2, 3],
                 [4, 5, 6]])

arr2 = np.array([[7, 8, 9],
                 [10, 11, 12]])

result = np.hstack((arr1, arr2))
print(result)

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

# Ans:-
#1. np.fliplr() (Flip Left-Right)
# Purpose: Reverses the order of elements along the horizontal axis
(left-right) for 2D arrays.
# Effect: Mirrors the array along its vertical axis (the left-right
axis).

#Example:-
import numpy as np

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

result = np.fliplr(arr)
print(result)
#Effect on Other Dimensions:
# 1D Arrays: Not applicable; fliplr() requires at least a 2D array.
# 3D Arrays: Only the 2D slices along the last two dimensions are
flipped horizontally.


# 2.np.flipud() (Flip Up-Down)
# Purpose: Reverses the order of elements along the vertical axis (up-
down) for 2D arrays.
# Effect: Mirrors the array along its horizontal axis (the up-down
axis).

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

result = np.flipud(arr)
print(result)
#Effect on Other Dimensions:
# 1D Arrays: Not applicable; flipud() requires at least a 2D array.
# 3D Arrays: Only the 2D slices along the last two dimensions are
flipped vertically.
```

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

#Ans:-
# Functionality of np.array_split():-
# 1.Basic Usage:

#Purpose: To divide an array into a specified number of sub-arrays or
according to specified indices along a given axis.

#Syntax: np.array_split(ary, indices_or_sections, axis=0)

# Parameters:

#ary: The input array to be split.
#indices_or_sections: Number of equal-sized sub-arrays to create, or
an array of indices where the splits should occur.
#axis: The axis along which to split the array (default is 0).

#2.Handling Uneven Splits:
#When the array cannot be split evenly, np.array_split() distributes
the elements as evenly as possible. Some sub-arrays might have one
more element than others.
#This is in contrast to np.split(), which requires the splits to be
evenly divisible; otherwise, it raises a ValueError.

#Example:-
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
result = np.array_split(arr, 3)
print(result)

#Example 2: Handling Uneven Splits:-
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
result = np.array_split(arr, 4)
print(result)

#Example 3: Splitting Using Indices
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
result = np.array_split(arr, [3, 6])
print(result)

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

#Ans:-

#1. Vectorization
#Concept:-
# Vectorization refers to the process of performing operations on
```

```
entire arrays rather than using explicit loops to process individual
elements.
# It leverages NumPy's ability to perform element-wise operations in a
highly optimized manner, typically through compiled C or Fortran code.

#Benefits:-
# Performance: Vectorized operations are faster than loops because
they are implemented in lower-level languages that execute more
efficiently.
# Simplicity: Vectorized code is often more concise and readable
compared to equivalent code that uses loops.

#Example:-
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
result_vectorized = arr * 2
print(result_vectorized)

#2. Broadcasting

#Concept:-
#Broadcasting is a technique that allows NumPy to perform arithmetic
operations on arrays of different shapes in a way that makes them
compatible.
#It automatically expands the smaller array to match the shape of the
larger array so that element-wise operations can be performed.

#Rules for Broadcasting:-
#Align Shapes: Starting from the trailing dimensions, the sizes of the
arrays must be either the same or one of them must be 1.
#Expand Dimensions: If arrays do not match in size, NumPy will
"broadcast" the smaller array across the larger array to make the
dimensions compatible.

#Benefits:-
#Flexibility: Allows operations between arrays of different shapes
without requiring explicit reshaping or duplication.
#Efficiency: Reduces the need for creating large temporary arrays,
optimizing both memory usage and performance.
```

Practical Questions:

```
# 1. Create a 3x3 NumPy array with random integers between 1 and 100.
Then, interchange its rows and columns.
```

```python
#Code:-
import numpy as np

array = np.random.randint(1, 101, size=(3, 3))
print("Original Array:")
print(array)


transposed_array = np.transpose(array)
print("\nTransposed Array:")
print(transposed_array)

#2. Generate a 1D NumPy array with 10 elements. Reshape it into a 2x5
array, then into a 5x2 array.

#code:-
import numpy as np

# Generate a 1D array with 10 elements
array_1d = np.arange(10)
print("1D Array:")
print(array_1d)

# Reshape it into a 2x5 array
array_2x5 = array_1d.reshape(2, 5)
print("\n2x5 Array:")
print(array_2x5)

# Reshape the 2x5 array into a 5x2 array
array_5x2 = array_2x5.reshape(5, 2)
print("\n5x2 Array:")
print(array_5x2)

#3. Create a 4x4 NumPy array with random float values. Add a border of
zeros around it, resulting in a 6x6 array.

#Code:-
import numpy as np

array_4x4 = np.random.random((4, 4))
print("Original 4x4 Array:")
print(array_4x4)


array_with_border = np.pad(array_4x4, pad_width=1, mode='constant',
constant_values=0)
print("\n6x6 Array with Border:")
print(array_with_border)

#4. Using NumPy, create an array of integers from 10 to 60 with a step
of 5.
```

```python
#Code:-
import numpy as np

array = np.arange(10, 65, 5)
print(array)

#5. Create a NumPy array of strings ['python', 'numpy', 'pandas'].
Apply different case transformations (uppercase, lowercase, title
case, etc.) to each element.


#Code:-

#1. Create the NumPy Array:
import numpy as np
array = np.array(['python', 'numpy', 'pandas'])
print("Original Array:")
print(array)

#2.Apply Case Transformations:-

uppercase_array = np.char.upper(array)
print("\nUppercase Array:")
print(uppercase_array)


lowercase_array = np.char.lower(array)
print("\nLowercase Array:")
print(lowercase_array)


titlecase_array = np.char.title(array)
print("\nTitle Case Array:")
print(titlecase_array)

# 6. Generate a NumPy array of words. Insert a space between each
character of every word in the array.

#Code:-

import numpy as np

# Create a NumPy array of words
words_array = np.array(['python', 'numpy', 'pandas'])

# Insert a space between each character of every word
spaced_words_array = np.char.join(' ', words_array)

print("Original Array:")
print(words_array)
```

```python
print("\nArray with Spaces Between Characters:")
print(spaced_words_array)

#7. Create two 2D NumPy arrays and perform element-wise addition,
subtraction, multiplication, and division.

#Code:-

import numpy as np

# Create two 2D NumPy arrays
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[10, 20, 30], [40, 50, 60]])

# Perform element-wise addition
addition_result = array1 + array2

# Perform element-wise subtraction
subtraction_result = array1 - array2

# Perform element-wise multiplication
multiplication_result = array1 * array2

# Perform element-wise division
division_result = array1 / array2

print("Array 1:")
print(array1)
print("\nArray 2:")
print(array2)

print("\nElement-wise Addition:")
print(addition_result)

print("\nElement-wise Subtraction:")
print(subtraction_result)

print("\nElement-wise Multiplication:")
print(multiplication_result)

print("\nElement-wise Division:")
print(division_result)



# 8. Use NumPy to create a 5x5 identity matrix, then extract its
diagonal elements

# Code:-
import numpy as np
```

```python
# Create a 5x5 identity matrix
identity_matrix = np.eye(5)
print("5x5 Identity Matrix:")
print(identity_matrix)

# Extract the diagonal elements
diagonal_elements = np.diagonal(identity_matrix)
print("\nDiagonal Elements:")
print(diagonal_elements)


#9. Generate a NumPy array of 100 random integers between 0 and 1000.
Find and display all prime numbers in this array

#Code:-
import numpy as np

# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

# Generate a NumPy array of 100 random integers between 0 and 1000
random_array = np.random.randint(0, 1001, size=100)
print("Random Array:")
print(random_array)

# Find and display all prime numbers in the array
primes = [num for num in random_array if is_prime(num)]
print("\nPrime Numbers in the Array:")
print(primes)

# 10. Create a NumPy array representing daily temperatures for a
month. Calculate and display the weekly averages.

#Code:-
import numpy as np

daily_temperatures = np.random.randint(0, 36, size=30)
print("Daily Temperatures for the Month:")
```

```python
print(daily_temperatures)

# Reshape the array into 4 weeks (each week has 7 days)
weekly_temperatures = daily_temperatures.reshape(4, 7)

# Calculate the weekly averages
weekly_averages = np.mean(weekly_temperatures, axis=1)
print("\nWeekly Averages:")
print(weekly_averages)
```