Theoretical Answers

Answer 1 :-

Purpose of NumPy in scientific computing and data analysis :-

1) It efficiently handles arrays.
2) It helps to perform complex mathematical operations.
3) It seamlessly integrates with other libraries.

Advantages of NumPy in scientific computing and data analysis :-

1) NumPy is ideal for handling large amounts of homogeneous (same – data type).
2) NumPy offers significant improvements in speed and memory efficiency.
3) It also provides high-level syntax for a wide range of numerical operations, making it a powerful tool for scientific computing and data processing on the CPU.

How does it enhance Python's capabilities for numerical operations :-

1) NumPy is built using C. As C is one of the fastest programming languages after C++
2) NumPy computation is fast.
3) NumPy work with multidimensional arrays efficiently.
4) NumPy facilitates advance mathematical and other types of operations on large numbers of data.

Answer 2 :-

Comparison of np.mean() and np.average() and when to use one over the other :-

np.mean() Function :-

1) This method is used to compute/calculate the arithmetic mean along with the specified axis.

2) All elements have equal weight.

3) Weight cannot be passed through the parameter of the given function.

4) **Syntax :**- np.mean(arr, axis = None)

np.average() Function :-

1) This method is used to compute the weighted average along the specified axis.

2) All elements may or may not have equal weight.

3) Weight can be passed through the parameter of the given function.

4) **Syntax :**- numpy.average(arr, axis = None, weights = None).

Answer 3 :-

Slicing method is one of the methods for reversing a NumPy array along different axes.

Examples for both 1D and 2D arrays:

Generated the np.array for 1D then reversed it using slicing method :-

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

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

```
# Reverse the 1D array using slicing :-

reversed_x = x[ : : -1]
reversed_x
```

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

Generated the np.array for 2D then reversed it using slicing method :-

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

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

```
# Reverse the 2D array using slicing with axis = 0, 1 :-

reversed_b = a[ : :-1, :]
reversed_b
```

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

```
reversed_b = a[: , : :-1]
reversed_b
```

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

Answer 4 :-

We can determine or check the data type of elements in NumPy array by using dtype attribute in the code while generating the array. Then it returns the data type all the elements in the array.

NumPy arrays can hold various data types like integers (int), floats (float), strings (str), etc. You can specify the data type during creation or let NumPy infer it..dtype attribute reveals the data type of the elements in the array

The importance of data types in memory management and performance: -

Data types are important because they are essential attributes of data that inform a computer system how to interpret its value. Understanding the different data types helps users choose the type that fits their needs and goals.

Answer 5 :-

NumPy arrays (ndarrays) are multidimensional collections of elements, all of the same data type. Unlike Python lists, NumPy arrays are designed for high-performance operations on large datasets. Arrays: The Core Data Structure.

The key features of NumPy:-

There are several ways to create NumPy arrays:

From Python Lists:

```
import numpy as np

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

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

Using Built-in Fuctions:

- np.zeros(shape): Creates an array filled with zeros.
- np.ones(shape): Creates an array filled with ones.
- np.empty(shape): Creates an array with uninitialized elements (faster than zeros).

```
zeros_arr = np.zeros(5)
print(zeros_arr)

ones_arr = np.ones((2, 3))
print(ones_arr)
```

```
[0. 0. 0. 0. 0.]
[[1. 1. 1.]
 [1. 1. 1.]]
```

Difference between ndarray and standard Python lists:

NumPy Arrays :-

- Only consists of elements belonging to the same data type.
- Need to explicitly import the array module for execution.
- Can directly handle arithmetic operations.
- Preferred for a longer sequence of data items.
- Less flexibility since addition, and deletion has to be done element-wise.
- A loop has to be formed to print or access the components of the array.
- Comparatively more compact in memory size.
- Nested arrays have to be of same size.
- **Example:** import array (arr = array.array('i', [1, 2, 3]))

Python Lists:

- Can consist of elements belonging to different data types.
- No need to explicitly import a module for the execution.
- Cannot directly handle arithmetic operations.
- Preferred for a shorter sequence of data items.
- Greater flexibility allows easy modification (addition, deletion) of data.
- The entire list can be printed without any explicit looping.
- Consume larger memory for easy addition of elements.
- Consume larger memory for easy addition of elements.
- **Example:** my_list = [1, 2, 3, 4]

Answer 6 :-

NumPy arrays are homogeneous and contiguous in memory, allowing for optimized processing with compiled C code. This results in faster execution times for mathematical operations and array manipulations compared to Python lists, particularly when dealing with large datasets or complex numerical computations.

Python lists are versatile data structures that can hold heterogeneous elements and dynamically resize, making them easy to use in many scenarios. However, when dealing with large datasets or complex numerical computations, lists may not be the most efficient choice due to several reasons:

Lists can hold elements of different types, so Python must perform type checking and handle different data representations, which can slow down operations.

Memory space: Each element in a Python list is a full Python object, containing not just the value but also type information, reference count, and other meta-data. This can result in significant memory overhead compared to more optimized data structures.

Answer 7 :-

The main comparison of vstack and hstack function:

```
# Generated 2 arrays:-
x = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
y = np.array([[9, 10, 11, 12], [13, 14, 15, 16]])
```

```
x
```

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

```
y
```

```
array([[ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

Vstack() positions them in a vertical line(row-wise).

- Syntax: np.vstack()
- Requirements: Arrays must have the same number of columns.

```
# Apply vsatck() :-

a = np.vstack((x, y))
a
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

hstack() positions views in a horizontal line(column-wise).

- Syntax: np.hstack()
- Requirements: Arrays must have the same number of rows.

```
# Apply hstack() :-

b = np.hstack((x, y))
b
```

```
array([[ 1,  2,  3,  4,  9, 10, 11, 12],
       [ 5,  6,  7,  8, 13, 14, 15, 16]])
```

Answer 8 :-

The fliplr() and flipud() methods in NumPy are used to flip or reverse arrays along specific axes.

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

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

fliplr() method in NumPy :-

- fliplr() specialized for horizontal flipping (left to right) in array.
- It reverses the order of elements along the second axis (columns).
- **Syntax :** np.fliplr( )
- Effect on 2D :

```
# Apply fliplr() :-

arr1 = np.fliplr(arr)
arr1
```

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

flipud() method in NumPy :-

- flipud() specialized for vertical flipping (upside down) of array.
- It reverses the order of elements along the first axis (rows).
- **Syntax :** np.flipud( )
- Effect on 2D :

```
# Apply flipud() :-

arr2 = np.flipud(arr)
arr2
```

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

Answer 9 :-

The array_split() method in NumPy is used to split an array into multiple sub-arrays along a specified axis. It offers more flexibility compared to the split() method by allowing for uneven splits when the array cannot be evenly divided.

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

Example 1:

```
string = "Ajay, Sunita, Bijay, Sanjay"
string.split(",")
```

```
['Ajay', ' Sunita', ' Bijay', ' Sanjay']
```

```
np.array(string.split(","))
```

```
array(['Ajay', ' Sunita', ' Bijay', ' Sanjay'], dtype='<U7')
```

Example 2:

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

```
result = np.array_split(arr, 3)
```

```
result
```

```
[array([1, 2, 3, 4]), array([5, 6, 7]), array([ 8,  9, 10])]
```

The array_split() method in NumPy provides a convenient way to split arrays into multiple sub-arrays along a specified axis. It accommodates uneven splits gracefully, ensuring that the resulting sub-arrays are as evenly distributed as possible given the size of the original array and the number of splits specified.

Answer 10 :-

Vectorization:

Vectorization in NumPy refers to the process of applying operations element-wise to entire arrays or sub-arrays without the need for explicit Python loops. It leverages the capabilities of NumPy arrays to perform computations efficiently in a batch or vectorized manner.

Example :-

```
# Generated 2 arrays :-

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

```
a
```

```
array([1, 2, 3])
```

```
b
```

```
array([4, 5, 6])
```

```
# vectorization :-

c = a + b
c
```

```
array([5, 7, 9])
```

Broadcasting - A Powerful Feature.

Broadcasting is a mechanism in NumPy that allows performing operations on arrays of different shapes under certain conditions. When operands have different shapes, NumPy expands the smaller array to match the larger one in a specific way to perform element-wise operations.

Example :-

```
# Generated 2 arrays :-

a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([20, 40, 60])
```

```
a
```

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

```
b
```

```
array([20, 40, 60])
```

```
# Broadcasting :-

c = a + b
c
```

```
array([[21, 42, 63],
       [24, 45, 66]])
```