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

NumPy is an open source mathematical and scientific computing library for Python programming tasks. The name NumPy is shorthand for Numerical Python. The NumPy library offers a collection of high-level mathematical functions including support for multi-dimensional arrays, masked arrays and matrices. NumPy also includes various logical and mathematical capabilities for those arrays such as shape manipulation, sorting, selection, linear algebra, statistical operations, random number generation and discrete Fourier transforms.

Scientific computing NumPy handles advanced mathematical operations such as matrix multiplication, eigenvalue calculation and differential equations. This makes NumPy particularly valuable across a vast range of simulation, modeling, visualization, computational processing and other scientific computing tasks.

```
 **Data manipulation and analysis**
   NumPy can be used for data cleaning, transformation and
aggregation. The data can then be readily processed through varied
NumPy mathematical operations such as statistical analysis, Fourier
analysis and linear algebra -- all vital for advanced data analytics
and data science tasks.

   NumPy is an indispensable tool for scientific computing, data
analysis, and numerical computations in Python. Its ability to
efficiently handle arrays, perform complex mathematical operations,
and seamlessly integrate with other libraries solidifies its position
as the cornerstone of numerical computing in Python.
```

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

np.mean()

```
Use to calculate arithmetic mean.
All elements have equal weight.
Weight cannot be passed trough the parameter of the given function.
Syntax :
np.mean(arr, axis = None)
where ‘arr’ is the given array.
```

np.average()

```
 Use to calculate the arithmetic mean as well as weighted average.
 All elements may or may not have equal weight.
 Weight can be passed through the parameter of the given function.
 Syntax :
 numpy.average(arr, axis = None, weights = None)
 Where ‘arr’ is the given array
```

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8,9,10])
print("Array : ",arr,"\n")
print("Mean of the Array : ",np.mean(arr))
```

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8,9,10])
print("Array : ",arr,"\n")
print("Mean of the Array : ",np.average(arr),"\n")
weight = np.array([4,5,6,12,15,10,2,8,19,20])
print("Weight average of Array : ",np.average(arr, weights=weight))
```

```
Array :  [ 1  2  3  4  5  6  7  8  9 10]

Mean of the Array :  5.5
Array :  [ 1  2  3  4  5  6  7  8  9 10]

Mean of the Array :  5.5

Weight average of Array :  6.574257425742574
```

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

As we know Numpy is a general-purpose array-processing package that provides a high-performance multidimensional array object, and tools for working with these arrays. Let's discuss how can we reverse a Numpy array.

Using flip() function to Reverse a Numpy array The numpy.flip() function reverses the order of array elements along the specified axis, preserving the shape of the array.

Using the list slicing method to reverse a Numpy array This method makes a copy of the list instead of sorting it in order. To accommodate all of the current components, making a clone requires additional room. More RAM is used up in this way. Here, we're utilizing Python's slicing method to invert our list.

Using flipud function to Reverse a Numpy array The numpy.flipud() function flips the array(entries in each column) in up-down direction, shape preserved.

```
# importing Numpy
import numpy as np

# 1d Array of Tuple
arr = [(1, 2, 3), ('Hi', 'Hello', 'Hey')]
x = map(np.array, arr)

# Changing map object to a list, then
# to an NDarray
x = np.array(list(x))
```

```
print(x)

# Checking the Dimension of the Resulting
# NDArray
print(x.ndim)



import numpy as np

# creating 2D float array.
float_array = np.array([[1., 2., 3.], [4., 5., 6.]])
print('Before converting: ', float_array, '\n Data type of array: ',
float_array.dtype)

# setting dtype to np.int32 to convert the data type of float_array
from float to int
float_array = np.array(float_array, dtype=np.int32)
print('After Converting: ', float_array, '\n Data type of array: ',
float_array.dtype)
```

```
[['1' '2' '3']
 ['Hi' 'Hello' 'Hey']]
2
Before converting:  [[1. 2. 3.]
 [4. 5. 6.]]
 Data type of array:  float64
After Converting:  [[1 2 3]
 [4 5 6]]
 Data type of array:  int32
```

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.

We can check the datatype of Numpy array by using dtype. Then it returns the data type all the elements in the array. In the given example below we import NumPy library and craete an array using “array()” method with integer value. Then we store the data type of the array in a variable named “data_type” using the ‘dtype’ attribute, and after then we can finally, we print the data type.

Create Arrays With a Defined Data Type We can create an array with a defined data type by specifying “dtype” attribute in numpy.array() method while initializing an array. In the below code we have created various types of defined arrays such as ‘float64’, ‘int32’, ‘complex128’, and ‘bool’.

Convert Data Type of NumPy Arrays We can convert data type of an arrays from one type to another using astype() function. In the below code we have initialize an array with float type values. After that we have convert that float64 type array to int32 type using astype() function. Finally, print the array and their types of original array and new array.

```python
import numpy as np

# Create an array with float data type elements
arr_original = np.array([1.2, 2.5, 3.7])

# Converting to int32
arr_new = arr_original.astype(np.int32)

# Print the original its type
print("Original array:", arr_original)
print("Data type of original array:", arr_original.dtype)

# Print new array and its type
print("\nNew array:", arr_new)
print("Data type of new array:", arr_new.dtype)
```

```
Original array: [1.2 2.5 3.7]
Data type of original array: float64

New array: [1 2 3]
Data type of new array: int32
```

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

Array in Numpy is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy, number of dimensions of the array is called rank of the array.A tuple of integers giving the size of the array along each dimension is known as shape of the array. An array class in Numpy is called as ndarray. Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists.

A Python list is a collection that is ordered and changeable. In Python, lists are written with square brackets.

Some important points about Python Lists:

The list can be homogeneous or heterogeneous. Element-wise operation is not possible on the list. Python list is by default 1-dimensional. But we can create an N-Dimensional list. But then too it will be 1 D list storing another 1D list Elements of a list need not be contiguous in memory.

The main features of NumPy's multi-dimensional arrays (numpy.ndarray) are as follows:

Require installation and importing of NumPy Can store only elements of the same type However, it is possible to store pointers to different types by using the object type NumPy: astype() to change dtype of an array Can represent multi-dimensional arrays Provide various methods and functions for numerical computation Useful in various scenarios such as matrix operations and image processing Matrix operations with NumPy in Python Image processing with Python, NumPy

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

NumPy is the fundamental package for scientific computing in Python. Numpy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences. Numpy is not another programming language but a Python extension module. It provides fast and efficient operations on arrays of homogeneous data.

Some important points about Numpy arrays:

We can create an N-dimensional array in Python using Numpy.array(). The array is by default Homogeneous, which means data inside an array must be of the same Datatype. (Note You can also create a structured array in Python). Element-wise operation is possible. Numpy array has various functions, methods, and variables, to ease our task of matrix computation. Elements of an array are stored contiguously in memory. For example, all rows of a two-dimensioned array must have the same number of columns. A three-dimensional array must have the same number of rows and columns on each card.

Representation of Numpy array Single Dimensional Numpy Array Multi-dimensional Numpy Array

Homogeneous Data: NumPy arrays store elements of the same data type, making them more compact and memory-efficient than lists. Fixed Data Type: NumPy arrays have a fixed data type, reducing memory overhead by eliminating the need to store type information for each element. Contiguous Memory: NumPy arrays store elements in adjacent memory locations, reducing fragmentation and allowing for efficient access. Array Metadata: NumPy arrays have extra metadata like shape, strides, and data type. However, this overhead is usually smaller than the per-element overhead in lists. Performance: NumPy arrays are optimized for numerical computations, with efficient element-wise operations and mathematical functions. These operations are implemented in C, resulting in faster performance than equivalent operations on lists.

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

vstack() function is used to stack arrays vertically (row-wise) to make a single array. It takes a sequence of arrays and joins them vertically. This is equivalent to concatenation along the first axis after 1-D arrays of shape (N,) have been reshaped to (1,N). This function is useful when you have two or more arrays with the same number of columns, and you want to concatenate them vertically (row-wise). It is also useful to append a single array as a new row to an existing 2D array.

The numpy.hstack() function is used to stack arrays in sequence horizontally (column wise). This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis. Rebuilds arrays divided by hsplit. This function is useful in the scenarios when we have to concatenate two arrays of different shapes along the second axis (column-wise). For example, to combine two arrays of shape (n, m) and (n, l) to form an array of shape (n, m+l).

```
# hstack() function

import numpy as rasmita
```

```
# input array
in_arr1 = rasmita.array([ 1, 2, 3] )
print ("1st Input array : \n", in_arr1)

in_arr2 = rasmita.array([ 4, 5, 6] )
print ("2nd Input array : \n", in_arr2)

# Stacking the two arrays horizontally
out_arr = rasmita.hstack((in_arr1, in_arr2))
print ("Output horizontally stacked array:\n ", out_arr)



# vstack() function

import numpy as rasmita

# input array
in_arr1 = rasmita.array([ 1, 2, 3] )
print ("1st Input array : \n", in_arr1)

in_arr2 = rasmita.array([ 4, 5, 6] )
print ("2nd Input array : \n", in_arr2)

# Stacking the two arrays vertically
out_arr = rasmita.vstack((in_arr1, in_arr2))
print ("Output vertically stacked array:\n ", out_arr)
```

```
1st Input array :
 [1 2 3]
2nd Input array :
 [4 5 6]
Output horizontally stacked array:
  [1 2 3 4 5 6]
1st Input array :
 [1 2 3]
2nd Input array :
 [4 5 6]
Output vertically stacked array:
  [[1 2 3]
 [4 5 6]]
```

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

numpy.fliplr(array) : Flip array(entries in each column) in left-right direction, shape preserved

numpy.fliplr(array) : Flip array(entries in each column) in left-right direction, shape preserved

```python
# numpy.flipud() method

import numpy as patra

array = patra.arange(8).reshape((2,2,2))
print("Original array : \n", array)

# flipud : means flip up-down
print("\nFlipped array : \n", geek.flipud(array))



# numpy.fliplr() method

import numpy as patra

array = patra.arange(8).reshape((2,2,2))
print("Original array : \n", array)

# fliplr : means flip left-right
print("\nFlipped array left-right : \n", patra.fliplr(array))
```

```
Original array :
 [[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]

Flipped array :
 [[[4 5]
  [6 7]]

 [[0 1]
  [2 3]]]
Original array :
 [[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]

Flipped array left-right :
 [[[2 3]
  [0 1]]

 [[6 7]
  [4 5]]]
```

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

Array splitting in NumPy is like a slice of cake. Think of each element in a NumPy array as a slice of cake. Splitting divides this “cake” into smaller “slices” (sub-arrays), often along specific dimensions or based on certain criteria. We can split horizontally, vertically, or even diagonally depending on our needs.

The split(), hsplit(), vsplit(), and dsplit() functions are important tools for dividing arrays along various axes and dimensions. These functions are particularly useful when working with one-dimensional arrays, matrices, or high-dimensional datasets. NumPy’s array-splitting capabilities are crucial for enhancing the efficiency and flexibility of data processing workflows.

Key concepts and terminology Here are some important terms to understand when splitting arrays:

Axis: The dimension along which the array is split (e.g., rows, columns, depth). Sub-arrays: The smaller arrays resulting from the split. Splitting methods: Different functions in NumPy for splitting arrays (e.g., np.split(), np.vsplit(), np.hsplit(), etc.). Equal vs. Unequal splits: Whether the sub-arrays have the same size or not.

```
import numpy as np
Arr = np.array([1, 2, 3, 4, 5, 6])
array = np.array_split(arr, 3)
print(array)
```

```
[array([['1', '2', '3']], dtype='<U21'), array([['Hi', 'Hello',
'Hey']], dtype='<U21'), array([], shape=(0, 3), dtype='<U21')]
```

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

Broadcasting provides a means of vectorizing array operations, therefore eliminating the need for Python loops. This is because NumPy is implemented in C Programming, which is a very efficient language.

It does this without making needless copies of data which leads to efficient algorithm implementations. But broadcasting over multiple arrays in NumPy extension can raise cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows down the computation.

The resulting array returned after broadcasting will have the same number of dimensions as the array with the greatest number of dimensions.

NumPy Broadcasting Arrays in Python Let’s assume that we have a large data set, each data is a list of parameters. In Numpy, we have a 2-D array, where each row is data and the number of rows is the size of the data set. Suppose we want to apply some sort of scaling to all these data every parameter gets its scaling factor or say every parameter is multiplied by some factor.

Just to have a clear understanding, let’s count calories in foods using a macro-nutrient breakdown. Roughly put, the caloric parts of food are made of fats (9 calories per gram), protein (4 CPG), and carbs (4 CPG).

So if we list some foods (our data), and for each food list its macro-nutrient breakdown (parameters), we can then multiply each nutrient by its caloric value (apply scaling) to compute the caloric breakdown of every food item.With this transformation, we can now compute all kinds of useful information. For example, what is the total number of calories present in some food, or given a breakdown of my dinner know how many calories did I get from protein and so on?

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

```
import numpy as np

array = np.random.randint(100, size=(3, 3))
print(array)
```

```
[[ 0  8 66]
 [86 31 81]
 [15 85 85]]
```

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

```
import numpy as np

# creating the list
list = [1,2,3,4,5,6,7,8,9,10]

# creating 1-d array
n = np.array(list)
print(n)
```

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

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

```
# importing Numpy package
import numpy as np

# Creating a 4X4 Numpy matrix
array = np.ones((4, 4))

print("Original array")
print(array)

print("\n0 on the border and 1 inside the array")


array = np.pad(array, pad_width=1, mode='constant',
               constant_values=0)
```

```
print(array)
```

```
Original array
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

0 on the border and 1 inside the array
[[0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

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

```
import numpy as np

array = np.ones((5, 5))

print("Original array")
print(array)

# Creating an array of integers from 30 to 70 using np.arange()
array = np.arange(10, 60)

# Printing a message indicating an array of integers from 30 to 70
print("Array of the integers from 10 to 60")

# Printing the array of integers from 30 to 70
print(array)
```

```
Original array
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
Array of the integers from 10 to 60
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33
 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
57
 58 59]
```

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

```
import numpy as np

x = np.array(['python', 'numpy', 'pandas'], dtype=str)


print (x)
```

```
['python' 'numpy' 'pandas']
```

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

```
# importing numpy as np
import numpy as np


# creating array of string
x = np.array(["geeks", "for", "geeks"],
                 dtype=np.str)
print("Printing the Original Array:")
print(x)

# inserting space using np.char.join()
r = np.char.join(" ", x)
print("Printing the array after inserting space\
between the elements")
print(r)
```

```
  File "<ipython-input-46-91588bed7853>", line 13
    print("Printing the array after inserting space\
          ^
SyntaxError: unterminated string literal (detected at line 13)
```

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

```
# Importing the NumPy library
import numpy as np

# Displaying a message for addition operation
print("Add:")
# Performing addition
print(np.add(1.0, 4.0))

# Displaying a message for subtraction operation
print("Subtract:")
# Performing subtraction
print(np.subtract(1.0, 4.0))
```

```
# Displaying a message for multiplication operation
print("Multiply:")
# Performing multiplication
print(np.multiply(1.0, 4.0))

# Displaying a message for division operation
print("Divide:")
# Performing division
print(np.divide(1.0, 4.0))
```

```
Add:
5.0
Subtract:
-3.0
Multiply:
4.0
Divide:
0.25
```

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

```
# Importing the NumPy library with an alias 'np'
import numpy as np

# Creating a diagonal matrix with diagonal elements 1, 2, 3, 4, 5
using np.diag()
x = np.diag([1, 2, 3, 4, 5])

# Printing the diagonal matrix 'x'
print(x)
```

```
[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]
```

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

```
import numpy as np

random_number = np.random.randint(0, 1000)

print(random_number)



def prime(x, y):
    prime_list = []
```

```
    for i in range(x, y):
        if i == 0 or i == 1:
            continue
        else:
            for j in range(2, int(i/2)+1):
                if i % j == 0:
                    break
            else:
                prime_list.append(i)
    return prime_list



starting_range = 0
ending_range = 1000
lst = prime(starting_range, ending_range)
if len(lst) == 0:
    print("There are no prime numbers in this range")
else:
    print("The prime numbers in this range are: ", lst)
```

```
930
The prime numbers in this range are:  [2, 3, 5, 7, 11, 13, 17, 19, 23,
29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101,
103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337,
347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421,
431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,
509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601,
607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683,
691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787,
797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881,
883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983,
991, 997]
```

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

```
import numpy as np
data = np.arange(1, 5)
data
np.average(data)
np.average(np.arange(1, 11), weights=np.arange(10, 0, -1))
```

```
4.0
```