Theoretical Questions:

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

Purpose of NumPy in Scientific Computing and Data Analysis:

NumPy (Numerical Python) is a foundational library in Python for numerical and scientific computing. Its primary purpose is to provide support for large, multi-dimensional arrays and matrices, along with a wide collection of mathematical functions to operate on these arrays efficiently. NumPy is fundamental to many other libraries in the Python ecosystem, such as Pandas, SciPy, and TensorFlow, all of which rely on its capabilities for fast array processing.

Advantages of NumPy:

Efficient Memory Usage: NumPy arrays (or ndarray) are more memory-efficient than Python's built-in data structures like lists, as they store elements in contiguous blocks of memory. This minimizes overhead and allows for the manipulation of large datasets without excessive memory usage.

Fast Performance: NumPy operations are implemented in C, providing faster execution of numerical operations compared to Python's built-in loops. Its vectorized operations avoid the overhead of loops in Python, offering a significant performance boost in many applications.

Multi-Dimensional Arrays: NumPy supports multi-dimensional arrays (tensors), enabling complex data manipulation that would be cumbersome to handle with standard Python lists. Operations across multiple dimensions (like matrix multiplication, element-wise addition) are seamlessly integrated.

Broad Functionality for Numerical Operations: NumPy offers an extensive library of mathematical functions such as linear algebra, Fourier transforms, random number generation, and statistical operations, which simplifies complex numerical computations.

Broadcasting: NumPy allows arithmetic operations between arrays of different shapes, a feature known as broadcasting. This eliminates the need for manual data manipulation or reshaping, simplifying the code and improving performance.

Integration with Other Libraries: NumPy serves as the foundation for other key libraries like Pandas, SciPy, and Matplotlib. It provides a common structure for data, making it easier for these libraries to interact with one another.

Interfacing with C/C++ and Fortran: NumPy provides easy integration with code written in C, C++, and Fortran, which is important for scientific computing where performance is critical. This allows for the execution of computationally expensive functions much faster.

How NumPy Enhances Python's Numerical Capabilities:

Vectorization: NumPy's ability to perform operations on whole arrays without using explicit loops (vectorization) enhances Python’s capability for numerical computations. This leads to cleaner code and significant speed improvements.

Multi-dimensional Computations: Python, by default, lacks efficient tools for working with multi-dimensional data structures. NumPy's arrays provide an efficient way to represent, manipulate, and compute with multi-dimensional data.

Reduced Overhead: Python's default lists are dynamically typed, meaning each element carries type and size information. NumPy arrays are homogeneous, reducing overhead and allowing for better memory and CPU cache utilization.

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

In NumPy, both np.mean() and np.average() are used to compute the central tendency of a dataset, but they have key differences in functionality and use cases.

#np.mean()

Purpose: Computes the arithmetic mean (simple average) of the elements along a specified axis.

Weights: No support for weighted averages. All elements are considered equally in the calculation.

```
 import numpy as np
 a=np.array([1,2,3,4,5])
 np.mean(a, axis=None,dtype=None,out=None,keepdims=False)
print(np.mean(a))
```

```
3.0
```

a: Input array.

axis: The axis along which to compute the mean. If None, the mean of the flattened array is calculated.

dtype: The data type for the result.

out: Optional output array.

keepdims: If True, the result will retain the dimensions of the input array.

np.average()

Purpose: Computes the weighted average of elements, where each element can contribute differently to the result based on provided weights.

Weights: Supports weighted averages via an optional weights parameter. If weights are not provided, it defaults to calculating the simple average, similar to np.mean().

```
Syntax
np.average(a, axis=None, weights=None, returned=False)
```

a: Input array.

axis: Axis along which to compute the average.

weights: Array of weights associated with the elements of a. It must have the same shape as a or be broadcastable to its shape.

returned: If True, also returns the sum of the weights along the specified axis.

```
# Example
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
weights = np.array([1, 1, 2, 2, 1])
avg = np.average(arr, weights=weights)
```

When to Use np.mean() vs. np.average():

Use np.mean(): When you want the simple arithmetic mean without considering weights or when the dataset does not require varying the contribution of individual elements. It is straightforward and typically faster.

Use np.average():

When you need to compute a weighted average, where each element’s contribution to the final result varies according to its assigned weight. If no weights are provided, np.average() behaves the same as np.mean(), but it's useful in cases where the weights might be specified later.

Example Use Cases:

np.mean(): Calculating the average temperature over a set of days where each day is equally important. np.average(): Calculating the average grade of a student where different assignments have different weights (e.g., exams count more than homework).

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

Reversing a NumPy array along different axes can be achieved using various methods, such as slicing, np.flip(), and np.flipud()/np.fliplr() for 2D arrays. Below are the methods and examples for both 1D and 2D arrays.

1. Slicing Method

1D Array (Reversing all elements):

In a 1D array, you can reverse the elements by slicing with [::-1], where -1 is the step that moves backward through the array

```
import numpy as np

arr_1d = np.array([1, 2, 3, 4, 5])
reversed_arr_1d = arr_1d[::-1]
print(reversed_arr_1d)
```

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

2D Array (Reversing elements along both axes):

For a 2D array, you can reverse the elements along both axes by using slicing. To reverse the rows and columns, use [::-1, ::-1].

```
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
reversed_arr_2d = arr_2d[::-1, ::-1]
print(reversed_arr_2d)
```

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

Reverse along one axis in 2D array:

Reverse rows (flip along axis 0): arr[::-1, :]

Reverse columns (flip along axis 1): arr[:, ::-1]

```
# Reverse rows
reversed_rows = arr_2d[::-1, :]
print(reversed_rows)

# Reverse columns
reversed_columns = arr_2d[:, ::-1]
print(reversed_columns)
```

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

2. Using np.flip()

np.flip() is a NumPy function that reverses the order of elements along a specified axis.

```
# Example for 1D Array:
reversed_arr_1d_flip = np.flip(arr_1d)
print(reversed_arr_1d_flip)
```

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

```
# Example for 2D Array:
# Reverse along axis 0 (rows)
reversed_rows_flip = np.flip(arr_2d, axis=0)
print(reversed_rows_flip)

# Reverse along axis 1 (columns)
reversed_columns_flip = np.flip(arr_2d, axis=1)
print(reversed_columns_flip)
```

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

1. Using np.flipud() and np.fliplr()

These two functions are specifically designed for 2D arrays:

np.flipud(): Flips the array in the up/down direction (along axis 0).

np.fliplr(): Flips the array in the left/right direction (along axis 1).

```python
# Flip array upside down
reversed_rows_flipud = np.flipud(arr_2d)

# Flip array left to right
reversed_cols_fliplr = np.fliplr(arr_2d)

print(reversed_rows_flipud)
print(reversed_cols_fliplr)
```

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

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

In NumPy, the data type (dtype) of the elements in an array is crucial for efficient memory management and performance. You can determine the data type of a NumPy array's elements using the dtype attribute.

How to Determine the Data Type of a NumPy Array

To find the data type of the elements in a NumPy array, use the .dtype attribute.

```python
#Example:
import numpy as np
arr = np.array([1, 2, 3])
print(arr.dtype)
```

```
int64
```

You can also create arrays with specific data types by specifying the dtype argument when creating an array

```
# Example
arr_float = np.array([1, 2, 3], dtype='float32')
print(arr_float.dtype)

float32
```

Importance of Data Types in Memory Management and Performance

1. Memory Management:

The data type determines the amount of memory that is allocated for each element in the array. Different data types consume different amounts of memory, and choosing the appropriate data type helps in optimizing memory usage.

Example:

int32 requires 4 bytes (32 bits) per element.

float64 requires 8 bytes (64 bits) per element.

bool requires 1 byte per element.

If you're working with large datasets, using unnecessarily large data types can lead to excessive memory usage. Conversely, using a smaller data type when a larger one is needed can cause data truncation and overflow errors.

```
arr_int32 = np.ones(1000000, dtype='int32')
arr_float64 = np.ones(1000000, dtype='float64')

print(arr_int32.nbytes)
print(arr_float64.nbytes)

4000000
8000000
```

Choosing int32 instead of float64 when you only need integer values saves 50% of the memory.

1. Performance:

Data types also impact the speed of computations. Smaller data types generally allow faster operations since they require fewer CPU cycles to process.

Operations on smaller data types (e.g., int8, float32) are typically faster than operations on larger ones (e.g., int64, float64).

Choosing an appropriate data type can significantly improve the performance of matrix operations, machine learning models, and scientific computing tasks.

```python
import time
arr_int32 = np.ones(10000000, dtype='int32')
arr_float64 = np.ones(10000000, dtype='float64')
start = time.time()
arr_int32_sum = np.sum(arr_int32)
end = time.time()
print("Time for int32 sum:", end - start)
start = time.time()
arr_float64_sum = np.sum(arr_float64)
end = time.time()
print("Time for float64 sum:", end - start)
```

```
Time for int32 sum: 0.010655879974365234
Time for float64 sum: 0.008707523345947266
```

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

Definition of ndarray in NumPy

An ndarray (N-dimensional array) is the core data structure of the NumPy library. It is a homogeneous, multi-dimensional container of fixed-size items, which means that all elements in an ndarray have the same data type and size.

Key Features of ndarray:

Multidimensionality:

An ndarray can have one or more dimensions (axes). Each dimension is called an "axis." For example, a 2D array has two axes: rows and columns.

Homogeneous Elements:

All elements in an ndarray must be of the same data type (e.g., integers, floats, booleans). This is in contrast to Python lists, which can hold elements of different types.

Efficient Memory Layout:

ndarray uses a contiguous block of memory, making it highly efficient for computations compared to Python lists, which are pointers to objects.

Fixed Size:

Once created, the size of an ndarray cannot be changed (though you can create a new array by reshaping or slicing the existing one).

Vectorized Operations:

NumPy arrays support element-wise operations and broadcasting, allowing for efficient and concise mathematical operations without explicit loops.

Optimized for Mathematical Operations:

NumPy arrays are optimized for mathematical and scientific computations, leveraging low-level C and Fortran libraries for performance.

Support for Broadcasting:

NumPy allows broadcasting, which enables operations between arrays of different shapes by "stretching" smaller arrays to match the shape of the larger ones without copying data.

Rich Functionality:

NumPy provides a vast range of built-in functions for performing operations on arrays, such as linear algebra, statistics, and array manipulations (e.g., reshaping, stacking, etc.).

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

print(arr)
print("Shape:", arr.shape)
print("Data type:", arr.dtype)
```

```
[[1 2 3]
 [4 5 6]]
Shape: (2, 3)
Data type: int64
```

```
# Example: Element-wise Operations
# Using NumPy ndarray:
import numpy as np

# Two 1D NumPy arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
result = arr1 + arr2
print(result)
```

```
[5 7 9]
```

```python
# Two Python lists
# Using Python Lists (Loop required):
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Element-wise addition using a loop
result = [a + b for a, b in zip(list1, list2)]
print(result)
```

```
[5, 7, 9]
```

Why Use ndarray Instead of Python Lists?

Performance:

NumPy arrays are optimized for mathematical operations and memory management, making them significantly faster than Python lists, especially for large datasets.

Memory Efficiency:

NumPy stores arrays in contiguous blocks of memory, minimizing memory overhead and maximizing cache efficiency.

Mathematical Operations: Operations on NumPy arrays are performed in a vectorized fashion, without the need for loops, making them concise and fast.

Multidimensional Arrays: NumPy provides native support for N-dimensional arrays, whereas Python lists only support one-dimensional arrays (though you can nest lists to mimic multidimensionality, it's less efficient and harder to manage).

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

Performance Benefits of NumPy Arrays Over Python Lists for Large-Scale Numerical Operations

NumPy arrays (ndarray) offer significant performance advantages over Python lists, particularly when handling large-scale numerical operations. These benefits stem from the way NumPy arrays are implemented at a lower level and the optimizations they provide for memory usage and computation.

Key Reasons for Performance Benefits:

1.Efficient Memory Storage:

Fixed Data Type: All elements in a NumPy array have the same data type, stored in a contiguous block of memory. This reduces the memory overhead, as each element takes up the exact same amount of memory. Compact Memory Layout: Python lists store pointers to objects, which adds extra overhead, whereas NumPy arrays store raw values directly, allowing for better cache locality and reducing memory access times.

2.Vectorized Operations:

NumPy arrays support element-wise operations that are executed at a low level in highly optimized C and Fortran routines. This eliminates the need for explicit loops, making operations on large arrays much faster than with Python lists.

Example: Adding two NumPy arrays can be done in a single statement without looping, whereas Python lists require looping through each element manually.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = arr1 + arr2
```

This is much faster than performing element-wise addition using loops in Python.

3.Optimized Underlying Libraries:

NumPy uses optimized low-level libraries (such as BLAS and LAPACK) written in C, C++, and Fortran, which handle array operations efficiently. These libraries are designed for numerical computation and provide highly efficient implementations of common operations like matrix multiplication, linear algebra, and statistical functions.

4.Avoiding Type Checking Overhead:

Python lists are heterogeneous, meaning they can store elements of different types. During operations, Python needs to check the type of each element, adding overhead. In contrast, NumPy arrays are homogeneous, and the type is known beforehand, eliminating the need for repeated type checks during operations.

5.Broadcasting:

NumPy’s broadcasting feature allows for operations between arrays of different shapes without the need to explicitly reshape or replicate arrays. This reduces both memory consumption and computation time, as NumPy avoids creating unnecessary copies of arrays.

Example: Adding a scalar to a large NumPy array is done in a single operation using broadcasting.

```
arr = np.array([1, 2, 3])
result = arr + 5
```

Benchmarking Example:

NumPy Arrays vs Python Lists

To better understand the performance difference between NumPy arrays and Python lists, consider the following example that sums two large arrays/lists.

```
import time
size = 10**7
list1 = list(range(size))
list2 = list(range(size))
start_time = time.time()
result = [a + b for a, b in zip(list1, list2)]
end_time = time.time()

print(f"Time taken using Python lists: {end_time - start_time}
seconds")
```

```
Time taken using Python lists: 2.1498780250549316 seconds
```

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

In NumPy, vstack() and hstack() are used for stacking arrays along different axes. They are helpful for combining arrays either vertically or horizontally.

1. vstack(): Vertical Stacking

Stacks arrays along the vertical axis (row-wise).

The arrays must have the same number of columns.

```
# Example
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

result_vstack = np.vstack((arr1, arr2))
print(result_vstack)
```

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

Here, arr1 and arr2 are stacked on top of each other as rows in a new 2D array.

1. hstack(): Horizontal Stacking

Stacks arrays along the horizontal axis (column-wise).

The arrays must have the same number of rows.

```
# Example
arr5 = np.array([1, 2, 3])
arr6 = np.array([4, 5, 6])
result_hstack = np.hstack((arr5, arr6))
print(result_hstack)
```

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

Here, arr5 and arr6 are concatenated side by side, creating a single 1D array.

vstack(): Stacks arrays vertically (row-wise), requiring the same number of columns.

hstack(): Stacks arrays horizontally (column-wise), requiring the same number of rows.

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

In NumPy, fliplr() and flipud() are used to reverse or flip the elements of an array along different axes. Let's explore their differences and behavior.

1. fliplr(): Flip Left to Right

Effect: Reverses the elements in each row (left to right).

Applicable: Only for 2D arrays or higher, with at least two dimensions (arrays with more than one column).

Axis of Flip: Horizontal axis (along the columns).

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

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

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

Here, fliplr() flips the array horizontally (along the rows), reversing the elements within each row.

```
# 3D Array Example:
arr_3d = np.array([[[1, 2], [3, 4]],
                   [[5, 6], [7, 8]]])

result_fliplr_3d = np.fliplr(arr_3d)
print(result_fliplr_3d)
```

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

 [[7 8]
  [5 6]]]
```

2. flipud(): Flip Up to Down

Effect: Reverses the elements in each column (top to bottom).

Applicable: For arrays of any dimension.

Axis of Flip: Vertical axis (along the rows).

```
#Example of 2d
result_flipud = np.flipud(arr)
print(result_flipud)
```

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

```
# Example of 3d
result_flipud_3d = np.flipud(arr_3d)
print(result_flipud_3d)
```

```
[[[5 6]
  [7 8]]

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

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

The array_split() method in NumPy is used to split an array into multiple sub-arrays. This method is flexible in handling arrays and allows for splitting along a specified axis.

Functionality of array_split()

Purpose: To divide an array into multiple sub-arrays along a specified axis.

Parameters:

ary: The array to be split.

indices_or_sections: Can be an integer or a 1D array of indices. If an integer is provided, it specifies the number of equal-sized sub-arrays to create. If an array of indices is provided, it specifies where to split the array.

axis: The axis along which to split. Default is 0 (rows).

Handling Uneven Splits

When splitting an array into sub-arrays of unequal sizes:

If the number of elements in the array is not perfectly divisible by the number of splits, array_split() will handle this by creating sub-arrays that may have different sizes.

For example, if you try to split an array into more parts than there are elements, some sub-arrays will be empty.

```
# Example
#Splitting an Array into Equal Parts
import numpy as np
arr = np.arange(10)
result = np.array_split(arr, 3)
print(result)
```

```
[array([0, 1, 2, 3]), array([4, 5, 6]), array([7, 8, 9])]
```

Here, the array is split into three parts. The parts are of unequal sizes because the array size (10) is not perfectly divisible by 3.

```
# Example
#Splitting Along an Axis in a 2D Array
arr2d = np.arange(12).reshape(3, 4)
result2d = np.array_split(arr2d, 3, axis=1)
print(result2d)
```

```
[array([[0, 1],
       [4, 5],
       [8, 9]]), array([[ 2],
       [ 6],
       [10]]), array([[ 3],
       [ 7],
       [11]])]
```

array_split() is useful for dividing an array into multiple sub-arrays.

It can handle cases where the array cannot be evenly split, creating sub-arrays of different sizes as needed.

The function is versatile and can split arrays along different axes and at specified indices, offering flexibility in data manipulation.

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

Vectorization and broadcasting are two powerful concepts in NumPy that significantly enhance the efficiency and performance of array operations.

Here’s a detailed look at each concept and how they contribute to efficient computations:

Vectorization

Concept:

Vectorization refers to the process of performing operations on entire arrays or vectors rather than on individual elements in a loop.

It leverages NumPy's ability to apply operations to entire arrays at once, using optimized low-level implementations that are generally much faster than Python loops.

Benefits:

Efficiency: Vectorized operations are implemented in C or Fortran, leading to highly optimized and faster computations compared to Python’s native loops.

Conciseness: Code using vectorized operations is often more concise and easier to read.

```
#Example: Suppose you want to add two arrays element-wise:
import numpy as np

# Two arrays
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

# Vectorized addition
c = a + b
print(c)
```

```
[ 6  8 10 12]
```

Broadcasting

Concept:

Broadcasting allows NumPy to perform operations on arrays of different shapes and sizes. It automatically expands the smaller array(s) to match the shape of the larger array during the operation.

This is done according to a set of rules that ensure that the shapes of arrays are compatible for element-wise operations.

Rules:

Align Dimensions: If the arrays have different numbers of dimensions, the smaller-dimensional array is padded with ones on the left side until the number of dimensions matches.

Shape Compatibility: After aligning dimensions, the shapes of the arrays must be compatible. They are compatible if:

They are the same, or

One of them is 1, or

They can be broadcast to a common shape.

```
#Example: Suppose you want to add a scalar to each element of an
array:
import numpy as np

# Array and scalar
arr = np.array([1, 2, 3, 4])
scalar = 10

# Broadcasting scalar addition
result = arr + scalar
print(result)
```

```
[11 12 13 14]
```

Vectorization: Enhances performance by applying operations to entire arrays at once, avoiding the overhead of Python loops.

Broadcasting: Facilitates operations on arrays of different shapes by automatically expanding the smaller array(s) to match the larger one’s shape, making it easier to perform element-wise operations.

Practical Questions:

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(1, 101, size=(3, 3))
print("Original Array:")
print(array)
transposed_array = array.T
print("\nTransposed Array:")
print(transposed_array)
```

```
Original Array:
[[ 80  18  42]
 [ 46  55  43]
 [ 13  83 100]]

Transposed Array:
[[ 80  46  13]
 [ 18  55  83]
 [ 42  43 100]]
```

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

```
import numpy as np
array_1d = np.arange(10)

print("Original 1D Array:")
print(array_1d)


array_2x5 = array_1d.reshape(2, 5)
print("\nReshaped into 2x5 Array:")

print(array_2x5)


array_5x2 = array_2x5.reshape(5, 2)
```

```
print("\nReshaped into 5x2 Array:")
print(array_5x2)
```

```
Original 1D Array:
[0 1 2 3 4 5 6 7 8 9]

Reshaped into 2x5 Array:
[[0 1 2 3 4]
 [5 6 7 8 9]]

Reshaped into 5x2 Array:
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

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

```
import numpy as np
array_1d = np.arange(10)

print("Original 1D Array:")

print(array_1d)


array_2x5 = array_1d.reshape(2, 5)

print("\nReshaped into 2x5 Array:")

print(array_2x5)

array_5x2 = array_2x5.reshape(5, 2)

print("\nReshaped into 5x2 Array:")

print(array_5x2)
```

```
Original 1D Array:
[0 1 2 3 4 5 6 7 8 9]

Reshaped into 2x5 Array:
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

```
Reshaped into 5x2 Array:
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

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

To create a NumPy array of integers from 10 to 60 with a step of 5, you can use the np.arange() function. This function generates values within a specified range with a defined step size.

```
import numpy as np
array = np.arange(10, 61, 5)
print(array)
```

```
[10 15 20 25 30 35 40 45 50 55 60]
```

np.arange(start, stop, step) generates values starting from start up to (but not including) stop, with a step increment.

In this case, np.arange(10, 61, 5) starts at 10, ends before 61, and increments by 5.

#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
strings = np.array(['python', 'numpy', 'pandas'])

uppercase_strings = np.char.upper(strings)
lowercase_strings = np.char.lower(strings)
titlecase_strings = np.char.title(strings)
capitalized_strings = np.char.capitalize(strings)

print("Original Strings:")
print(strings)

print("\nUppercase:")
print(uppercase_strings)

print("\nLowercase:")
print(lowercase_strings)

print("\nTitle Case:")
print(titlecase_strings)
print("\nCapitalized:")
print(capitalized_strings)

Original Strings:
['python' 'numpy' 'pandas']

Uppercase:
['PYTHON' 'NUMPY' 'PANDAS']

Lowercase:
['python' 'numpy' 'pandas']

Title Case:
['Python' 'Numpy' 'Pandas']

Capitalized:
['Python' 'Numpy' 'Pandas']
```

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

```
import numpy as np
words = np.array(['hello', 'world', 'numpy'])
words_with_spaces = np.char.join(' ', words)


print("Original Words:")
print(words)

print("\nWords with Spaces Between Characters:")
print(words_with_spaces)

Original Words:
['hello' 'world' 'numpy']

Words with Spaces Between Characters:
['h e l l o' 'w o r l d' 'n u m p y']
```

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

```python
import numpy as np
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[7, 8, 9], [10, 11, 12]])


# Addition
addition_result = array1 + array2

# Subtraction
subtraction_result = array1 - array2

# Multiplication
multiplication_result = array1 * array2

# 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)
```

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

Array 2:
[[ 7  8  9]
 [10 11 12]]

Element-Wise Addition:
[[ 8 10 12]
```

```
 [14 16 18]]

Element-Wise Subtraction:
[[-6 -6 -6]
 [-6 -6 -6]]

Element-Wise Multiplication:
[[ 7 16 27]
 [40 55 72]]

Element-Wise Division:
[[0.14285714 0.25       0.33333333]
 [0.4        0.45454545 0.5       ]]
```

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

```
import numpy as np
identity_matrix = np.eye(5)

print("5x5 Identity Matrix:")

print(identity_matrix)

diagonal_elements = np.diagonal(identity_matrix)

print("\nDiagonal Elements:")

print(diagonal_elements)

5x5 Identity Matrix:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]

Diagonal Elements:
[1. 1. 1. 1. 1.]
```

#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
array = np.random.randint(0, 1000, size=100)
print("Array of Random Integers:")
print(array)
```

```
def is_prime(n):
    """Return True if n is a prime number, else False."""
    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
prime_numbers = [num for num in array if is_prime(num)]
print("\nPrime Numbers in the Array:")
print(prime_numbers)

Array of Random Integers:
[704 621 158 976 925 732 330 829  85 756 553 130 619 302 132 514 719
997
 195 419 861 113 883 722 634 135 801 956 767 639 607 749 533 827 901
524
   5 771 764 417 450  94 686 965 481  34 797  35 550 891 788 149 391
397
 295 141 702 941 912 319 656   2 122 430 547 408 658 212 550 547 245
764
 968 163 485 342 934 111 168 165 804 613 399 921  61 464 927 585 741
371
 445 251 861 180 961 628 422 978 965  50]

Prime Numbers in the Array:
[829, 619, 719, 997, 419, 113, 883, 607, 827, 5, 797, 149, 397, 941,
2, 547, 547, 163, 613, 61, 251]
```

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

```
import numpy as np

np.random.seed(0)
daily_temperatures = np.random.randint(20, 35, size=30)
print("Daily Temperatures for the Month:")
print(daily_temperatures)

weekly_temperatures = daily_temperatures.reshape(5, 6)
print("\nWeekly Temperatures:")
print(weekly_temperatures)
```

```
weekly_averages = np.mean(weekly_temperatures, axis=1)
print("\nWeekly Averages:")
print(weekly_averages)

Daily Temperatures for the Month:
[32 25 20 23 31 23 27 29 23 25 22 24 27 26 28 28 32 30 21 26 27 27 34
28
 21 25 29 33 28 29]

Weekly Temperatures:
[[32 25 20 23 31 23]
 [27 29 23 25 22 24]
 [27 26 28 28 32 30]
 [21 26 27 27 34 28]
 [21 25 29 33 28 29]]

Weekly Averages:
[25.66666667 25.         28.5        27.16666667 27.5       ]
```