Name – Gathram Kesava Murthy

Registered mail id- kesavagathram.murthy@outlook.com

Course Name – Master Generative AI: Data Science Course

Assignment Name – Numpy

Submission Date – 30th October 2024

Drive Link –

https://colab.research.google.com/drive/1_RPKfAGoGPtMk-9HNjxZNjTexQ4vU7Jl?usp=sharing

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, short for Numerical Python, is a powerful library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. Here's an overview of its purpose, advantages, and how it enhances Python's capabilities for numerical operations:

Purpose of NumPy:

1. **Efficient Array Operations**: NumPy allows for the creation of N-dimensional arrays (ndarrays) that facilitate efficient storage and manipulation of numerical data.
2. **Mathematical Functions**: It provides a comprehensive set of mathematical functions for performing operations on arrays, including linear algebra, statistics, Fourier transforms, and more.
3. **Interoperability**: NumPy serves as the foundation for many other scientific computing libraries, such as SciPy, Pandas, and Matplotlib, allowing for seamless integration and enhanced functionality.

Advantages of NumPy:

1. **Performance**: NumPy arrays are implemented in C and optimized for performance, making operations on large datasets significantly faster than standard Python lists.
2. **Memory Efficiency**: NumPy arrays consume less memory compared to Python lists, as they store data in a contiguous block of memory and have a fixed data type.
3. **Convenience**: It offers a range of high-level operations and functions, such as broadcasting, which simplifies complex operations on arrays of different shapes.
4. **Rich Functionality**: NumPy includes a wide array of mathematical functions, enabling complex computations with minimal code.
5. **Community Support**: Being a widely used library in the scientific and data analysis communities, it has extensive documentation and a strong user community for support and resources.

Enhancing Python's Numerical Operations

1. **Vectorization:** NumPy allows for vectorized operations, meaning that operations can be applied to entire arrays without the need for explicit loops. This leads to cleaner code and improved performance.
2. **Broadcasting:** NumPy can perform operations on arrays of different shapes by automatically expanding the smaller array to match the larger array's shape.
3. **Support for Mathematical Functions**: NumPy provides functions for trigonometric, statistical, and other mathematical operations that work seamlessly with its array structures.
4. **Integration with Other Libraries**: NumPy serves as the backbone for many other libraries in the scientific computing stack, enhancing their capabilities and providing a consistent interface for numerical operations.

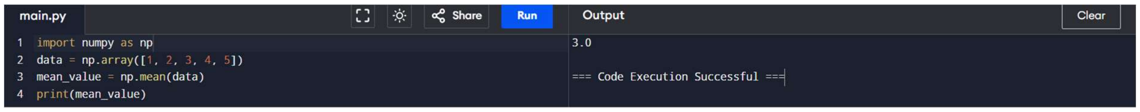
2. **Compare and contrast np.mean() and np.average() functions in NumPy. When would you use one over the other?**

The np.mean() and np.average() functions in NumPy are both used to compute the average of array elements, but they differ in their functionality and use cases. Here's a comparison of the two:

np.mean()

- **Purpose**: Computes the arithmetic mean (average) of the elements in an array.
- **Syntax**: np.mean(a, axis=None, dtype=None, out=None, keepdims=False)
- **Parameters**:
    - a: Input array.
    - axis: Axis or axes along which the means are computed. Default is None, which computes the mean of the flattened array.
    - dtype: Data type to use in computing the mean. Default is None.
    - out: Alternative output array in which to place the result.
    - keepdims: If True, the reduced axes will be left in the result as dimensions with size one.
- **Use Case**: Use np.mean() when you want a straightforward calculation of the average of the elements in the array, without any weighting.

Example:



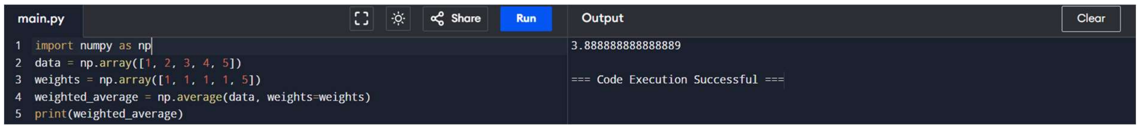
```python
import numpy as np
data = np.array([1, 2, 3, 4, 5])
mean_value = np.mean(data)
print(mean_value)
```

```
3.0

=== Code Execution Successful ===
```

np.average()

- **Purpose**: Computes the weighted average of the elements in an array.
- **Syntax**: np.average(a, axis=None, weights=None, returned=False)
- **Parameters**:
    - a: Input array.
    - axis: Axis or axes along which the averages are computed. Default is None.
    - weights: Optional array of weights associated with the values in a. If provided, it must be the same shape as a, or be broadcastable to the shape of a.
    - returned: If True, it also returns the weighted average and the sum of weights.
- **Use Case**: Use np.average() when you need to compute an average that takes weights into account, allowing different elements to contribute differently to the final average.

Example



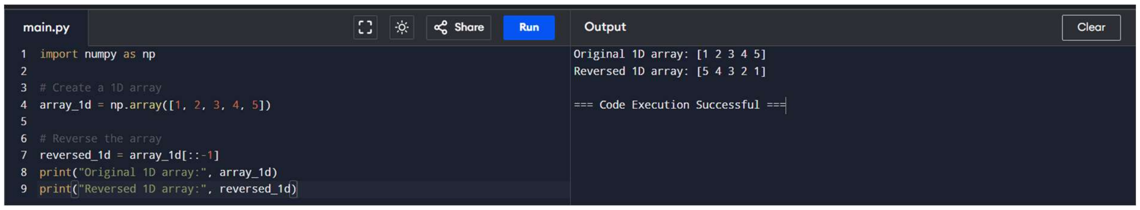
```python
import numpy as np
data = np.array([1, 2, 3, 4, 5])
weights = np.array([1, 1, 1, 1, 5])
weighted_average = np.average(data, weights=weights)
print(weighted_average)
```

```
3.888888888888889

=== Code Execution Successful ===
```

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

Reversing a NumPy array can be done easily using slicing techniques. The methods for reversing can vary depending on whether you are working with one-dimensional (1D) or two-dimensional (2D) arrays, and you can reverse along specific axes for 2D arrays. Here's how to do it for both cases:

Reversing a 1D NumPy Array



```python
import numpy as np

# Create a 1D array
array_1d = np.array([1, 2, 3, 4, 5])

# Reverse the array
reversed_1d = array_1d[::-1]
print("Original 1D array:", array_1d)
print("Reversed 1D array:", reversed_1d)
```

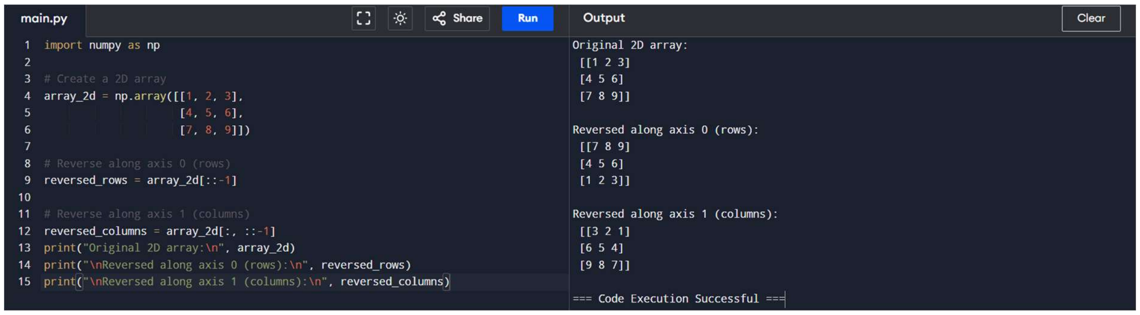
```
Original 1D array: [1 2 3 4 5]
Reversed 1D array: [5 4 3 2 1]

=== Code Execution Successful ===
```

Reversing a 2D NumPy Array

For a 2D array, you can reverse along different axes using slicing.

- **Reversing along axis 0 (rows):** This means reversing the order of the rows.
- **Reversing along axis 1 (columns):** This means reversing the order of the columns.

Example:



```python
import numpy as np

# Create a 2D array
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])

# Reverse along axis 0 (rows)
reversed_rows = array_2d[::-1]

# Reverse along axis 1 (columns)
reversed_columns = array_2d[:, ::-1]
print("Original 2D array:\n", array_2d)
print("\nReversed along axis 0 (rows):\n", reversed_rows)
print("\nReversed along axis 1 (columns):\n", reversed_columns)
```

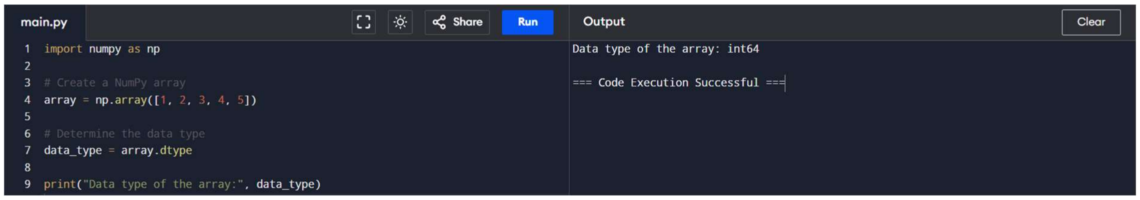
```
Original 2D array:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Reversed along axis 0 (rows):
 [[7 8 9]
 [4 5 6]
 [1 2 3]]

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

=== Code Execution Successful ===
```

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.**

In NumPy, you can determine the data type of elements in an array using the `.dtype` attribute. The data type (`dtype`) of a NumPy array is crucial because it defines the type of data that can be stored in the array and how much memory is allocated for each element. Here's how to check the data type and an overview of its importance:

Determining the Data Type

Example:



```python
import numpy as np

# Create a NumPy array
array = np.array([1, 2, 3, 4, 5])

# Determine the data type
data_type = array.dtype

print("Data type of the array:", data_type)
```

```
Data type of the array: int64

=== Code Execution Successful ===
```

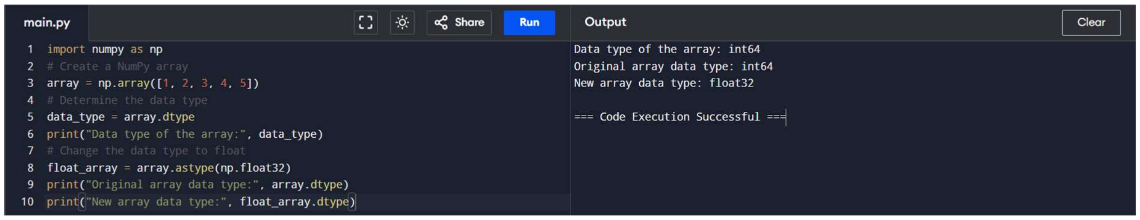
Importance of Data Types in Memory Management and Performance

1. **Memory Management:**
    - **Storage Size:** Each data type in NumPy has a specific size in bytes. For instance:
        - int32 occupies 4 bytes.
        - int64 occupies 8 bytes.
        - float32 occupies 4 bytes, while float64 occupies 8 bytes.
    - **Efficiency:** Using the appropriate data type can lead to significant memory savings, especially when working with large datasets. For example, using float32 instead of float64 can halve the memory usage for floating-point numbers.
2. **Performance:**
    - **Speed of Operations:** Operations on arrays of smaller data types are generally faster. For example, operations on int8 or float32 arrays can be more efficient than on int64 or float64 due to reduced memory bandwidth and cache usage.
    - **Vectorized Operations:** NumPy is optimized for vectorized operations, which can be more effective when the data type is consistent and appropriate for the task. The use of appropriate data types can lead to enhanced performance in numerical computations.
3. **Type Safety:**
    - **Preventing Errors:** Knowing the data type helps in preventing errors that can arise from incompatible data types. For instance, operations between incompatible types (like adding an integer array to a string array) can lead to runtime errors.
    - **Type-Specific Functions:** Many NumPy functions are optimized for specific data types. Using the correct type ensures that you can leverage these optimizations effectively.

Changing Data Types

You can also change the data type of a NumPy array using the .astype() method:

Example:



```python
import numpy as np
# Create a NumPy array
array = np.array([1, 2, 3, 4, 5])
# Determine the data type
data_type = array.dtype
print("Data type of the array:", data_type)
# Change the data type to float
float_array = array.astype(np.float32)
print("Original array data type:", array.dtype)
print("New array data type:", float_array.dtype)
```

```
Data type of the array: int64
Original array data type: int64
New array data type: float32

=== Code Execution Successful ===
```

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

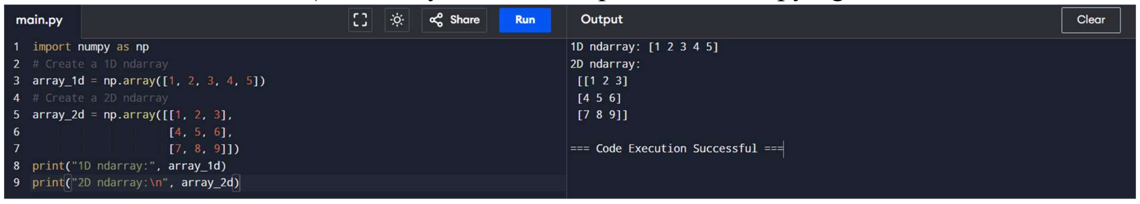
In NumPy, **ndarrays** (N-dimensional arrays) are the core data structure used for representing and manipulating numerical data. They provide a powerful way to work with large datasets efficiently. Here's a detailed overview of ndarrays, their key features, and how they differ from standard Python lists.

Definition of ndarrays

An **ndarray** is a multi-dimensional, homogeneous array of fixed-size items, where all items are of the same data type. This uniformity allows NumPy to perform efficient operations on large datasets and enables advanced mathematical computations.

Key Features of ndarrays:

1. **Homogeneous Data**: All elements in an ndarray must be of the same data type, which ensures consistency in calculations and operations.
2. **N-Dimensional**: Ndarrays can have any number of dimensions, allowing for flexible representation of data:
   - 1D arrays (vectors)
   - 2D arrays (matrices)
   - 3D arrays (tensors) and beyond.
3. **Efficient Memory Usage**: Ndarrays use contiguous memory blocks, resulting in lower memory overhead compared to Python lists. This layout enhances cache efficiency and accelerates data processing.
4. **Performance**: NumPy operations on ndarrays are optimized and executed in compiled code, leading to significantly faster computations compared to equivalent operations on Python lists.
5. **Broadcasting**: Ndarrays support broadcasting, which allows operations on arrays of different shapes without the need for explicit replication of data.
6. **Vectorization**: Ndarrays enable vectorized operations, allowing element-wise operations to be performed without the need for explicit loops, resulting in cleaner and more efficient code.
7. **Rich Functionality**: NumPy provides a wide range of mathematical functions and methods that can be directly applied to ndarrays for various operations like statistical analysis, linear algebra, and more.
8. **Shape and Reshaping**: Ndarrays have a defined shape (number of dimensions and size of each dimension), and they can be reshaped without copying the data.



```python
import numpy as np
# Create a 1D ndarray
array_1d = np.array([1, 2, 3, 4, 5])
# Create a 2D ndarray
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])
print("1D ndarray:", array_1d)
print("2D ndarray:\n", array_2d)
```

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

=== Code Execution Successful ===
```

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

When it comes to large-scale numerical operations, NumPy arrays offer significant performance benefits over standard Python lists. Below are the key advantages of using NumPy arrays and the underlying reasons for their improved performance:

1. Homogeneous Data Types

- **Efficiency**: NumPy arrays are homogeneous, meaning all elements are of the same data type. This allows for more efficient memory allocation and faster processing since the size and type of data are known in advance. In contrast, Python lists can contain mixed types, leading to more overhead in memory management.
- **Type-Specific Optimization**: NumPy can optimize operations based on the specific data type (e.g., integers, floats), resulting in faster execution of mathematical operations.

2. Contiguous Memory Allocation

- **Memory Layout**: NumPy arrays use contiguous blocks of memory, enabling better cache coherence. This means that when accessing elements of an array, the CPU can load data more efficiently from memory into its cache, speeding up computations.
- **Reduced Overhead**: In Python lists, each element is a separate object that carries additional overhead, which can slow down performance when dealing with large datasets.

3. Vectorized Operations

- **No Explicit Loops**: NumPy allows for vectorized operations, enabling element-wise operations on entire arrays without the need for explicit loops. This leads to cleaner code and significantly reduces the time taken to perform operations compared to looping through each element in a Python list.
- **Bulk Processing**: Many NumPy operations are implemented in C, which is compiled and optimized, allowing for bulk processing of data. This can be orders of magnitude faster than equivalent operations on Python lists.

4. Broadcasting

- **Flexible Operations**: NumPy's broadcasting feature allows arrays of different shapes to be used together in operations, automatically expanding the smaller array to match the shape of the larger one. This feature simplifies code and enhances performance since it minimizes the need for additional data structures or copying data.

5. Rich Mathematical Functions

- **Optimized Libraries**: NumPy is built on optimized libraries like BLAS and LAPACK, which are designed for high-performance linear algebra and numerical computations. These libraries take advantage of low-level optimizations that are not accessible when using standard Python lists.
- **Built-in Functions**: NumPy provides a wide range of optimized mathematical functions that are designed to work with ndarrays, further improving performance.

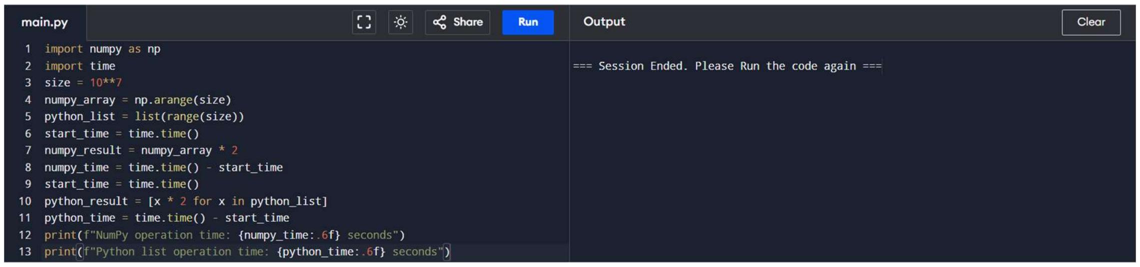
6. Parallel Processing

- **Multithreading**: Some NumPy operations can be parallelized internally, allowing multiple CPU cores to be utilized. This can lead to significant performance improvements, especially for large datasets.

7. Memory Efficiency

- **Lower Memory Footprint**: Since NumPy arrays have a fixed size and type, they generally require less memory compared to Python lists. This is particularly important when working with large datasets, as it reduces the overall memory usage and increases the amount of data that can be held in RAM.

Example:



```python
import numpy as np
import time
size = 10**7
numpy_array = np.arange(size)
python_list = list(range(size))
start_time = time.time()
numpy_result = numpy_array * 2
numpy_time = time.time() - start_time
start_time = time.time()
python_result = [x * 2 for x in python_list]
python_time = time.time() - start_time
print(f"NumPy operation time: {numpy_time:.6f} seconds")
print(f"Python list operation time: {python_time:.6f} seconds")
```

```
=== Session Ended. Please Run the code again ===
```

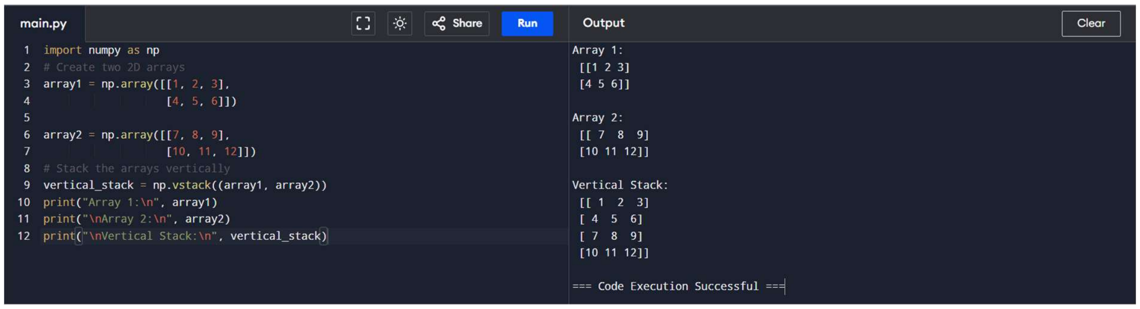
7. **Compare vstack() and hstack() functions in NumPy. Provide examples demonstrating their usage and output.**

In NumPy, the vstack() and hstack() functions are used to stack arrays vertically and horizontally, respectively. Here's a detailed comparison of the two functions along with examples demonstrating their usage and output.

vstack()

- **Purpose**: Stacks arrays in sequence vertically (row-wise). This means that it adds arrays on top of each other.
- **Input**: It can take a single array or a sequence of arrays. All arrays must have the same shape along all but the first axis.

Example of vstack()



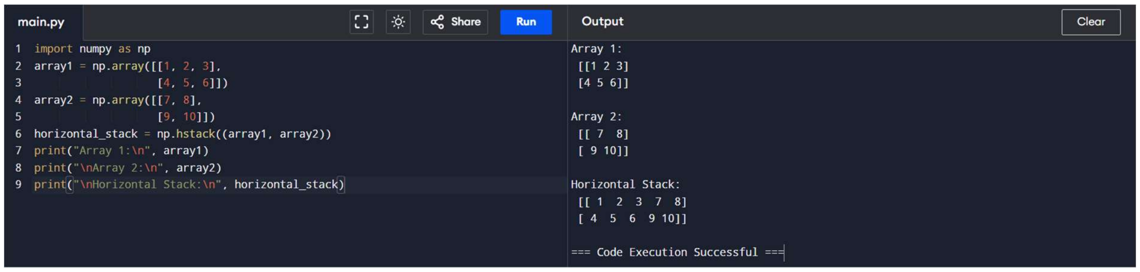
```python
import numpy as np
# Create two 2D arrays
array1 = np.array([[1, 2, 3],
                   [4, 5, 6]])

array2 = np.array([[7, 8, 9],
                   [10, 11, 12]])
# Stack the arrays vertically
vertical_stack = np.vstack((array1, array2))
print("Array 1:\n", array1)
print("\nArray 2:\n", array2)
print("\nVertical Stack:\n", vertical_stack)
```

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

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

Vertical Stack:
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

=== Code Execution Successful ===
```

hstack()

- **Purpose**: Stacks arrays in sequence horizontally (column-wise). This means that it adds arrays side by side.
- **Input**: Similar to vstack(), it can take a single array or a sequence of arrays. All arrays must have the same shape along all but the second axis.

Example of hstack()



```python
import numpy as np
array1 = np.array([[1, 2, 3],
                   [4, 5, 6]])
array2 = np.array([[7, 8],
                   [9, 10]])
horizontal_stack = np.hstack((array1, array2))
print("Array 1:\n", array1)
print("\nArray 2:\n", array2)
print("\nHorizontal Stack:\n", horizontal_stack)
```

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

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

Horizontal Stack:
 [[ 1  2  3  7  8]
 [ 4  5  6  9 10]]

=== Code Execution Successful ===
```

Key Differences

- **Direction**:
  - vstack() adds rows, stacking arrays vertically.
  - hstack() adds columns, stacking arrays horizontally.
- **Shape Requirements**:
  - For vstack(), the input arrays must have the same number of columns.
  - For hstack(), the input arrays must have the same number of rows.

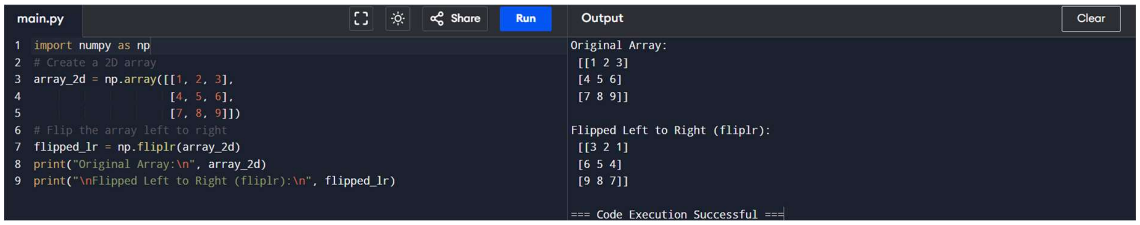
8. **Explain the differences between fliplr() and flipud() methods in NumPy, including their effects on various array dimensions.**

In NumPy, the methods fliplr() and flipud() are used to flip arrays in specific directions. Here's a detailed explanation of the differences between these two methods, including their effects on various array dimensions.

fliplr()

- **Purpose**: Flips an array left to right (horizontally). This means that the columns of the array are reversed.
- **Input**: It takes a 2D array (or higher dimensions, where it applies only to the last two dimensions).

Example of fliplr()



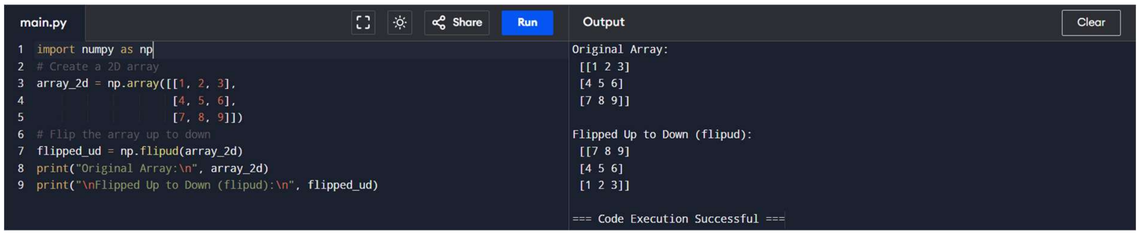
```python
import numpy as np
# Create a 2D array
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])
# Flip the array left to right
flipped_lr = np.fliplr(array_2d)
print("Original Array:\n", array_2d)
print("\nFlipped Left to Right (fliplr):\n", flipped_lr)
```

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

Flipped Left to Right (fliplr):
 [[3 2 1]
 [6 5 4]
 [9 8 7]]

=== Code Execution Successful ===
```

flipud()

- **Purpose**: Flips an array up to down (vertically). This means that the rows of the array are reversed.
- **Input**: It takes a 2D array (or higher dimensions, where it applies only to the last two dimensions).

Example of flipud()



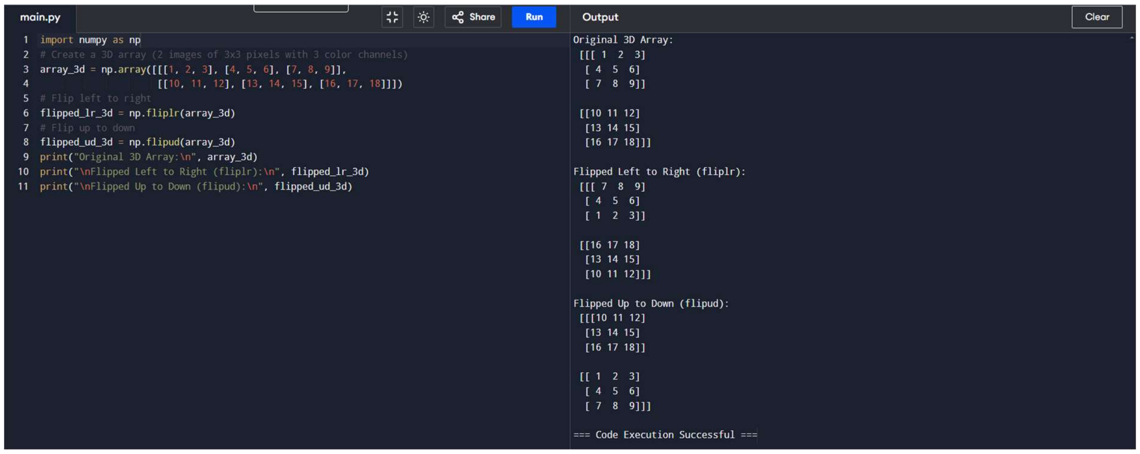
```python
import numpy as np
# Create a 2D array
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])
# Flip the array up to down
flipped_ud = np.flipud(array_2d)
print("Original Array:\n", array_2d)
print("\nFlipped Up to Down (flipud):\n", flipped_ud)
```

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

Flipped Up to Down (flipud):
 [[7 8 9]
 [4 5 6]
 [1 2 3]]

=== Code Execution Successful ===
```

Effects on Higher Dimensions

- **For 2D Arrays:** The behavior is as described above**.**
- **For 3D Arrays:** If you have a 3D array (e.g., representing RGB images), fliplr() will flip each 2D slice horizontally, while flipud() will flip each 2D slice vertically.

Example :



```python
import numpy as np
# Create a 3D array (2 images of 3x3 pixels with 3 color channels)
array_3d = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
                     [[10, 11, 12], [13, 14, 15], [16, 17, 18]]])
# Flip left to right
flipped_lr_3d = np.fliplr(array_3d)
# Flip up to down
flipped_ud_3d = np.flipud(array_3d)
print("Original 3D Array:\n", array_3d)
print("\nFlipped Left to Right (fliplr):\n", flipped_lr_3d)
print("\nFlipped Up to Down (flipud):\n", flipped_ud_3d)
```

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

 [[10 11 12]
  [13 14 15]
  [16 17 18]]]

Flipped Left to Right (fliplr):
 [[[ 7  8  9]
  [ 4  5  6]
  [ 1  2  3]]

 [[16 17 18]
  [13 14 15]
  [10 11 12]]]

Flipped Up to Down (flipud):
 [[[10 11 12]
  [13 14 15]
  [16 17 18]]

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

=== Code Execution Successful ===
```

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

The array_split() method in NumPy is a powerful tool used to split an array into multiple sub-arrays along a specified axis. This method is particularly useful when you need to divide large datasets into smaller chunks for processing or analysis.

Functionality of array_split()

- **Basic Syntax**:

  numpy.array_split(ary, indices_or_sections, axis=0)

- ary: The input array to be split.
- indices_or_sections: This can be either an integer or an array-like structure:
  - If it is an integer, it defines the number of equal-sized sub-arrays to create. If the array cannot be split evenly, the last sub-array will be smaller.
  - If it is an array-like structure, it specifies the indices at which to split the array.
- axis: The axis along which to split the array. The default is 0 (the first axis).

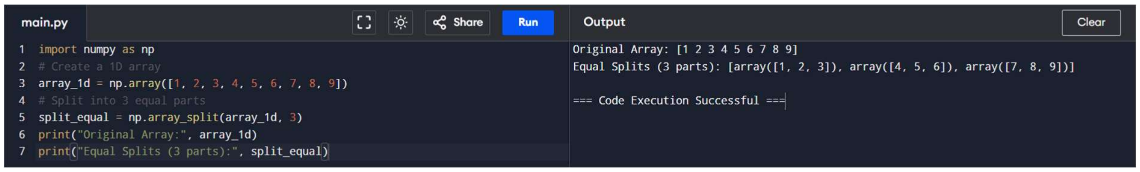
Handling Uneven Splits

When the input array cannot be split evenly, array_split() handles the situation gracefully:

- It will distribute the elements as evenly as possible among the resulting sub-arrays.
- If there are remaining elements after the even distribution, these will be added one by one to the last sub-array.

Examples

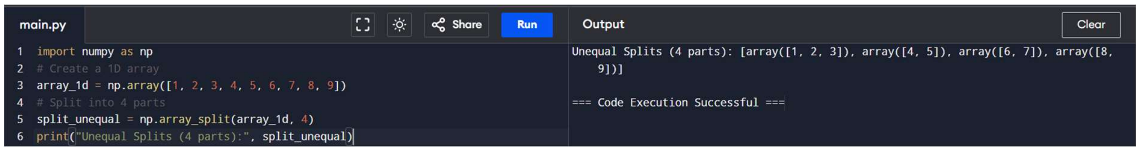
Example 1: Splitting an Array into Equal Parts



```python
import numpy as np
# Create a 1D array
array_1d = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
# Split into 3 equal parts
split_equal = np.array_split(array_1d, 3)
print("Original Array:", array_1d)
print("Equal Splits (3 parts):", split_equal)
```

```
Original Array: [1 2 3 4 5 6 7 8 9]
Equal Splits (3 parts): [array([1, 2, 3]), array([4, 5, 6]), array([7, 8, 9])]

=== Code Execution Successful ===
```

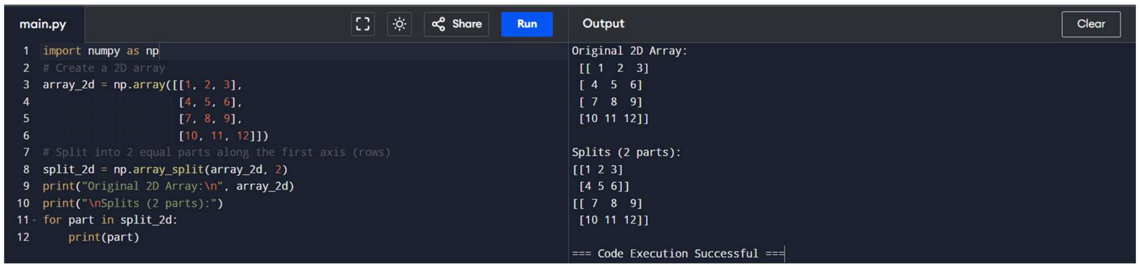
Example 2: Splitting an Array with Uneven Parts



```python
import numpy as np
# Create a 1D array
array_1d = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
# Split into 4 parts
split_unequal = np.array_split(array_1d, 4)
print("Unequal Splits (4 parts):", split_unequal)
```

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

=== Code Execution Successful ===
```

Example 3: Splitting a 2D Array



```python
import numpy as np
# Create a 2D array
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9],
                     [10, 11, 12]])
# Split into 2 equal parts along the first axis (rows)
split_2d = np.array_split(array_2d, 2)
print("Original 2D Array:\n", array_2d)
print("\nSplits (2 parts):")
for part in split_2d:
    print(part)
```

```
Original 2D Array:
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Splits (2 parts):
[[1 2 3]
 [4 5 6]]
[[ 7  8  9]
 [10 11 12]]

=== Code Execution Successful ===
```

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

In NumPy, **vectorization** and **broadcasting** are two key concepts that significantly enhance the efficiency of array operations. They allow for optimized computation and minimize the need for explicit loops, which can be slow in Python. Here's a detailed explanation of each concept and how they contribute to efficient array operations.

Vectorization

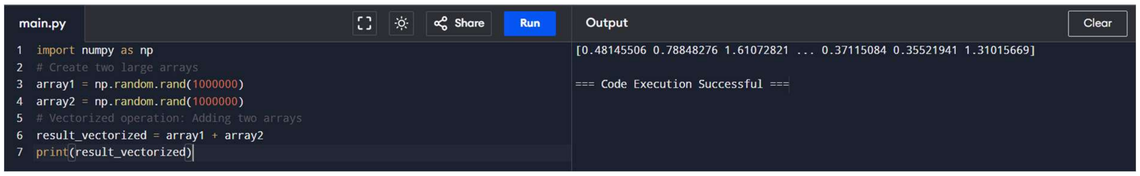
Concept

Vectorization refers to the process of converting operations that would typically be performed element-wise using loops into operations that can be applied to entire arrays at once. This takes advantage of NumPy's underlying implementation in C and its ability to perform operations on entire arrays in a single operation, rather than iterating through individual elements.

Benefits

1. **Performance**: Vectorized operations are typically much faster than using Python loops because they are implemented in optimized C code, which can handle data in bulk.
2. **Code Simplicity**: Vectorization leads to cleaner, more concise code, making it easier to read and maintain.

Example of Vectorization:



```python
import numpy as np
# Create two large arrays
array1 = np.random.rand(1000000)
array2 = np.random.rand(1000000)
# Vectorized operation: Adding two arrays
result_vectorized = array1 + array2
print(result_vectorized)
```

```
[0.48145506 0.78848276 1.61072821 ... 0.37115084 0.35521941 1.31015669]

=== Code Execution Successful ===
```

Broadcasting

Concept

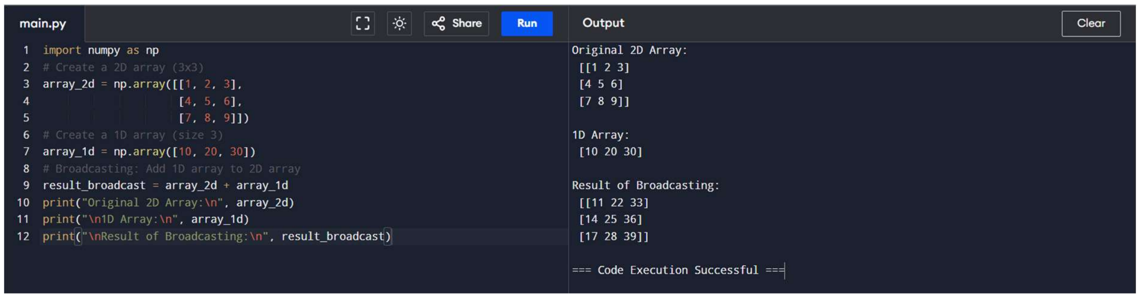
Broadcasting is a technique used in NumPy to perform operations on arrays of different shapes. When the shapes of the arrays are not the same, NumPy can automatically expand the smaller array to match the shape of the larger one in a way that makes sense for the operation being performed.

How It Works

1. **Dimensions Matching**: When performing operations, NumPy compares the shapes of the arrays element-wise, starting from the trailing dimensions (i.e., from the rightmost side).
2. **Expansion**: If the dimensions are not the same, NumPy will attempt to "broadcast" the smaller array across the larger array, effectively replicating it as needed.
3. **Conditions for Broadcasting**:
   - If the dimensions of the two arrays are equal, they are compatible.

- If one of the dimensions is 1, it can be stretched to match the other dimension.
- If the shapes are incompatible (neither condition is met), NumPy raises a ValueError.

Example of Broadcasting:



```python
import numpy as np
# Create a 2D array (3x3)
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])
# Create a 1D array (size 3)
array_1d = np.array([10, 20, 30])
# Broadcasting: Add 1D array to 2D array
result_broadcast = array_2d + array_1d
print("Original 2D Array:\n", array_2d)
print("\n1D Array:\n", array_1d)
print("\nResult of Broadcasting:\n", result_broadcast)
```

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

1D Array:
 [10 20 30]

Result of Broadcasting:
 [[11 22 33]
 [14 25 36]
 [17 28 39]]

=== Code Execution Successful ===
```

Contribution to Efficient Array Operations

1. **Reduction of Looping**: Both vectorization and broadcasting eliminate the need for explicit loops in array operations, which can be slow and inefficient in Python.
2. **Optimized Performance**: By utilizing underlying C and Fortran libraries, NumPy operations are executed in compiled code, making them significantly faster than equivalent operations performed in pure Python.
3. **Memory Efficiency**: Broadcasting allows for operations on arrays of different shapes without creating additional copies of data, conserving memory usage.
4. **Cleaner Code**: Vectorized and broadcasted operations lead to more concise and readable code, making it easier to develop and maintain numerical applications.