```python
# Rajnish singh- numpy assignment



# question 1. Create a NumPy array 'arr' of integers from 0 to 5 and print its data type.


import numpy as np

# Create the NumPy array
arr = np.arange(6)  # This creates an array with values from 0 to 5

# Print the array
print("Array:", arr)

# Print the data type of the array
print("Data type:", arr.dtype)





# question 2- Given a NumPy array 'arr', check if its data type is float64.

import numpy as np

# Create the NumPy array
arr = np.array([1.5, 2.6, 3.7])

# Check if the data type is float64
if arr.dtype == np.float64:
    print("The data type of the array is float64.")
else:
    print("The data type of the array is not float64. It is", arr.dtype)





#question-3.  Create a NumPy array 'arr' with a data type of complex128 containing three complex numbers.

import numpy as np

# Create the NumPy array with complex128 data type
arr = np.array([1+2j, 3+4j, 5+6j], dtype=np.complex128)

# Print the array
print("Array:", arr)
```

```
# Print the data type of the array
print("Data type:", arr.dtype)
```

```
#question-4.  Convert an existing NumPy array 'arr' of integers to float32 data
type.

import numpy as np

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

# Convert the array to float32 data type
arr_float32 = arr.astype(np.float32)

# Print the converted array and its data type
print("Converted array:", arr_float32)
print("Data type:", arr_float32.dtype)
```

```
#question-5. Given a NumPy array 'arr' with float64 data type, convert it to
float32 to reduce decimal precision

import numpy as np

# Example array with float64 data type
arr = np.array([1.123456789, 2.987654321, 3.456789123], dtype=np.float64)

# Convert the array to float32 data type
arr_float32 = arr.astype(np.float32)

# Print the original and converted arrays along with their data types
print("Original array:", arr)
print("Original data type:", arr.dtype)
print("Converted array:", arr_float32)
print("Converted data type:", arr_float32.dtype)
```

```
#question-6. Write a function array_attributes that takes a NumPy array as input
```

```
and returns its shape, size, and data
type.

import numpy as np

def array_attributes(arr):
    """
    Returns the shape, size, and data type of the given NumPy array.

    Parameters:
    arr (numpy.ndarray): The NumPy array whose attributes are to be retrieved.

    Returns:
    tuple: A tuple containing the shape, size, and data type of the array.
    """
    shape = arr.shape
    size = arr.size
    dtype = arr.dtype

    return shape, size, dtype

# Example usage:
if __name__ == "__main__":
    # Create a sample NumPy array
    example_array = np.array([[1, 2, 3], [4, 5, 6]])

    # Get array attributes
    shape, size, dtype = array_attributes(example_array)

    # Print the attributes
    print("Shape:", shape)
    print("Size:", size)
    print("Data type:", dtype)
```

#question7. Create a function array_dimension that takes a NumPy array as input and returns its dimensionality.

```
import numpy as np

def array_dimension(arr):
    """
    Returns the dimensionality of the given NumPy array.

    Parameters:
    arr (numpy.ndarray): The NumPy array whose dimensionality is to be retrieved.
    Returns:
    int: The number of dimensions of the array.
    """
    return arr.ndim

# Example usage:
if __name__ == "__main__":
    # Create a sample NumPy array with different dimensions
    example_array_1d = np.array([1, 2, 3, 4])
    example_array_2d = np.array([[1, 2], [3, 4]])
    example_array_3d = np.array([[[1], [2]], [[3], [4]]])

    # Get array dimensions
    dim_1d = array_dimension(example_array_1d)
    dim_2d = array_dimension(example_array_2d)
    dim_3d = array_dimension(example_array_3d)

    # Print the dimensions
    print("Dimensionality of 1D array:", dim_1d)
    print("Dimensionality of 2D array:", dim_2d)
    print("Dimensionality of 3D array:", dim_3d)
```

```
#question-8. Design a function item_size_info that takes a NumPy array as input and
returns the item size and the total
size in bytes.

import numpy as np

def item_size_info(arr):
    """
    Returns the item size and total size in bytes of the given NumPy array.

    Parameters:
    arr (numpy.ndarray): The NumPy array whose item size and total size are to be
retrieved.

    Returns:
    tuple: A tuple containing the item size and the total size in bytes of the
array.
    """
    item_size = arr.itemsize
    total_size = arr.nbytes
    return item_size, total_size

# Example usage:
if __name__ == "__main__":
    # Create a sample NumPy array with different data types
    example_array_int = np.array([1, 2, 3, 4], dtype=np.int32)
    example_array_float = np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float64)
    example_array_complex = np.array([1+2j, 3+4j], dtype=np.complex128)

    # Get item size and total size
    item_size_int, total_size_int = item_size_info(example_array_int)
    item_size_float, total_size_float = item_size_info(example_array_float)
    item_size_complex, total_size_complex = item_size_info(example_array_complex)

    # Print the item size and total size
    print("Int array - Item size:", item_size_int, "bytes, Total size:",
total_size_int, "bytes")
    print("Float array - Item size:", item_size_float, "bytes, Total size:",
total_size_float, "bytes")
    print("Complex array - Item size:", item_size_complex, "bytes, Total size:",
total_size_complex, "bytes")
```

#question-9. Create a function array_strides that takes a NumPy array as input and returns the strides of the array.

```
import numpy as np

def array_strides(arr):
    """
    Returns the strides of the given NumPy array.

    Parameters:
    arr (numpy.ndarray): The NumPy array whose strides are to be retrieved.

    Returns:
    tuple: A tuple containing the strides of the array in bytes.
    """
    return arr.strides

# Example usage:
if __name__ == "__main__":
    # Create sample NumPy arrays with different shapes
    example_array_1d = np.array([1, 2, 3, 4])
    example_array_2d = np.array([[1, 2, 3], [4, 5, 6]])
    example_array_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

    # Get strides for each example array
    strides_1d = array_strides(example_array_1d)
    strides_2d = array_strides(example_array_2d)
    strides_3d = array_strides(example_array_3d)

    # Print the strides
    print("Strides of 1D array:", strides_1d)
    print("Strides of 2D array:", strides_2d)
    print("Strides of 3D array:", strides_3d)
```

question 10. Design a function shape_stride_relationship that takes a NumPy array as input and returns the shape
and strides of the array.

```
import numpy as np

def shape_stride_relationship(arr):
    """
    Returns the shape and strides of the given NumPy array.

    Parameters:
    arr (numpy.ndarray): The NumPy array whose shape and strides are to be
retrieved.

    Returns:
    tuple: A tuple containing the shape and strides of the array.
    """
    shape = arr.shape
    strides = arr.strides

    return shape, strides

# Example usage:
if __name__ == "__main__":
    # Create sample NumPy arrays with different shapes
    example_array_1d = np.array([1, 2, 3, 4])
    example_array_2d = np.array([[1, 2, 3], [4, 5, 6]])
    example_array_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

    # Get shape and strides for each example array
    shape_1d, strides_1d = shape_stride_relationship(example_array_1d)
    shape_2d, strides_2d = shape_stride_relationship(example_array_2d)
    shape_3d, strides_3d = shape_stride_relationship(example_array_3d)

    # Print the shape and strides
    print("1D array - Shape:", shape_1d, ", Strides:", strides_1d)
    print("2D array - Shape:", shape_2d, ", Strides:", strides_2d)
    print("3D array - Shape:", shape_3d, ", Strides:", strides_3d)
```

#question-11. Create a function `create_zeros_array` that takes an integer `n` as input and returns a NumPy array of zeros with `n` elements.

```
import numpy as np

def create_zeros_array(n):
    """
    Creates a NumPy array of zeros with n elements.

    Parameters:
    n (int): The number of elements in the array.

    Returns:
    numpy.ndarray: A NumPy array of zeros with n elements.
    """
    if n < 0:
        raise ValueError("The number of elements must be a non-negative integer.")

    return np.zeros(n)

# Example usage:
if __name__ == "__main__":
    # Create arrays with different sizes
    array_5 = create_zeros_array(5)
    array_10 = create_zeros_array(10)

    # Print the created arrays
    print("Array with 5 zeros:", array_5)
    print("Array with 10 zeros:", array_10)
```

```
#question-12. Write a function `create_ones_matrix` that takes integers `rows` and
`cols` as inputs and generates a 2D
NumPy array filled with ones of size `rows x cols`.

import numpy as np

def create_ones_matrix(rows, cols):
    """
    Creates a 2D NumPy array filled with ones with the specified number of rows and
columns.

    Parameters:
    rows (int): The number of rows in the matrix.
    cols (int): The number of columns in the matrix.

    Returns:
    numpy.ndarray: A 2D NumPy array of ones with dimensions (rows, cols).
    """
    if rows <= 0 or cols <= 0:
        raise ValueError("Both rows and columns must be positive integers.")

    return np.ones((rows, cols))

# Example usage:
if __name__ == "__main__":
    # Create matrices with different sizes
    matrix_2x3 = create_ones_matrix(2, 3)
    matrix_4x5 = create_ones_matrix(4, 5)

    # Print the created matrices
    print("2x3 matrix of ones:\n", matrix_2x3)
    print("4x5 matrix of ones:\n", matrix_4x5)
```

```
#question 13. Write a function `generate_range_array` that takes three integers
start, stop, and step as arguments and
creates a NumPy array with a range starting from `start`, ending at stop
(exclusive), and with the specified
```

`step`.

```
import numpy as np

def generate_range_array(start, stop, step):
    """
    Creates a NumPy array with a range starting from 'start', ending at 'stop'
(exclusive), and with the specified 'step'.

    Parameters:
    start (int): The starting value of the range.
    stop (int): The end value of the range (exclusive).
    step (int): The step size between values in the range.

    Returns:
    numpy.ndarray: A NumPy array containing the range of values.
    """
    if step <= 0:
        raise ValueError("Step must be a positive integer.")

    return np.arange(start, stop, step)

# Example usage:
if __name__ == "__main__":
    # Generate arrays with different ranges
    range_array_1 = generate_range_array(0, 10, 2)
    range_array_2 = generate_range_array(5, 20, 3)
    range_array_3 = generate_range_array(-10, 0, 5)

    # Print the generated arrays
    print("Range array 1:", range_array_1)
    print("Range array 2:", range_array_2)
    print("Range array 3:", range_array_3)
```

#question 14. Design a function `generate_linear_space` that takes two floats `start`, `stop`, and an integer `num` as arguments and generates a NumPy array with num equally spaced values between `start` and `stop`
(inclusive).

```
import numpy as np
```

```
def generate_linear_space(start, stop, num):
    """
    Generates a NumPy array with num equally spaced values between start and stop
(inclusive).

    Parameters:
    start (float): The starting value of the range.
    stop (float): The end value of the range (inclusive).
    num (int): The number of values to generate.

    Returns:
    numpy.ndarray: A NumPy array containing the equally spaced values.
    """
    if num <= 0:
        raise ValueError("The number of values must be a positive integer.")

    return np.linspace(start, stop, num)

# Example usage:
if __name__ == "__main__":
    # Generate arrays with different linear spaces
    linear_space_5 = generate_linear_space(0.0, 1.0, 5)
    linear_space_10 = generate_linear_space(-1.0, 1.0, 10)
    linear_space_3 = generate_linear_space(10.0, 20.0, 3)

    # Print the generated arrays
    print("Linear space with 5 values:", linear_space_5)
    print("Linear space with 10 values:", linear_space_10)
    print("Linear space with 3 values:", linear_space_3)




#question 15. Create a function `create_identity_matrix` that takes an integer `n`
as input and generates a square
identity matrix of size `n x n` using `numpy.eye`.

import numpy as np

def create_identity_matrix(n):
    """
    Creates a square identity matrix of size n x n.

    Parameters:
    n (int): The size of the identity matrix.

    Returns:
```

```
    numpy.ndarray: A square identity matrix of size n x n.
    """
    if n <= 0:
        raise ValueError("The size of the matrix must be a positive integer.")

    return np.eye(n)

# Example usage:
if __name__ == "__main__":
    # Create identity matrices with different sizes
    identity_2x2 = create_identity_matrix(2)
    identity_3x3 = create_identity_matrix(3)
    identity_4x4 = create_identity_matrix(4)

    # Print the created identity matrices
    print("2x2 Identity Matrix:\n", identity_2x2)
    print("3x3 Identity Matrix:\n", identity_3x3)
    print("4x4 Identity Matrix:\n", identity_4x4)
```

#question 16. Write a function that takes a Python list and converts it into a NumPy array.

```
import numpy as np

def list_to_numpy_array(py_list):
    """
    Converts a Python list into a NumPy array.

    Parameters:
    py_list (list): The Python list to be converted.

    Returns:
    numpy.ndarray: A NumPy array created from the Python list.
    """
    # Convert the Python list to a NumPy array
    return np.array(py_list)

# Example usage:
if __name__ == "__main__":
    # Create sample Python lists
    list_1d = [1, 2, 3, 4, 5]
    list_2d = [[1, 2, 3], [4, 5, 6]]

    # Convert lists to NumPy arrays
    array_1d = list_to_numpy_array(list_1d)
    array_2d = list_to_numpy_array(list_2d)

    # Print the converted NumPy arrays
    print("1D NumPy array:", array_1d)
    print("2D NumPy array:\n", array_2d)
```

#question 17. Create a NumPy array and demonstrate the use of `numpy.view` to create a new array object with the
same data.

```
import numpy as np

# Create a NumPy array with some data
original_array = np.array([1, 2, 3, 4, 5, 6])

# Create a view of the original array with the same data
view_array = original_array.view()

# Modify the original array to show that the view is affected as well
original_array[0] = 10

# Print the original array and the view to show they reflect the same data
print("Original Array:", original_array)
print("View Array:", view_array)
```

#question 18. Write a function that takes two NumPy arrays and concatenates them along a specified axis.

```
import numpy as np

def concatenate_arrays(arr1, arr2, axis=0):
    """
    Concatenates two NumPy arrays along a specified axis.

    Parameters:
    arr1 (numpy.ndarray): The first NumPy array.
    arr2 (numpy.ndarray): The second NumPy array.
    axis (int): The axis along which to concatenate. Default is 0.
    Returns:
    numpy.ndarray: The concatenated NumPy array.
    """
    # Concatenate the arrays along the specified axis
    return np.concatenate((arr1, arr2), axis=axis)

# Example usage:
if __name__ == "__main__":
    # Create sample NumPy arrays
    array1 = np.array([[1, 2, 3], [4, 5, 6]])
    array2 = np.array([[7, 8, 9], [10, 11, 12]])

    # Concatenate along axis 0 (rows)
    result_axis0 = concatenate_arrays(array1, array2, axis=0)

    # Concatenate along axis 1 (columns)
    result_axis1 = concatenate_arrays(array1, array2, axis=1)

    # Print the results
    print("Concatenated along axis 0:\n", result_axis0)
    print("Concatenated along axis 1:\n", result_axis1)
```

question 19. Create two NumPy arrays with different shapes and concatenate them horizontally using `numpy.

concatenate`.

```
import numpy as np

# Create two NumPy arrays with compatible shapes for horizontal concatenation
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[7, 8], [9, 10]])

# Check the shapes to ensure they are compatible for horizontal concatenation
print("Shape of array1:", array1.shape)
print("Shape of array2:", array2.shape)

# Concatenate the arrays horizontally (along axis 1)
concatenated_array = np.concatenate((array1, array2), axis=1)

# Print the concatenated array
print("Concatenated Array:\n", concatenated_array)
```

question 20. Write a function that vertically stacks multiple NumPy arrays given as a list.

```
import numpy as np

def stack_arrays_vertically(arrays):
    """
    Vertically stacks multiple NumPy arrays provided in a list.

    Parameters:
    arrays (list of numpy.ndarray): A list of NumPy arrays to be stacked
vertically.

    Returns:
    numpy.ndarray: The vertically stacked NumPy array.
    """
    # Ensure that the input is a list of NumPy arrays
    if not all(isinstance(arr, np.ndarray) for arr in arrays):
        raise ValueError("All elements of the list must be NumPy arrays.")

    # Vertically stack the arrays
    return np.vstack(arrays)

# Example usage:
if __name__ == "__main__":
    # Create sample NumPy arrays
    array1 = np.array([[1, 2, 3], [4, 5, 6]])
    array2 = np.array([[7, 8, 9], [10, 11, 12]])
    array3 = np.array([[13, 14, 15]])

    # Stack arrays vertically
    stacked_array = stack_arrays_vertically([array1, array2, array3])

    # Print the result
    print("Stacked Array:\n", stacked_array)
```

question 21. Write a Python function using NumPy to create an array of integers within a specified range (inclusive)
with a given step size.

```
import numpy as np

def create_range_array(start, stop, step):
    """
    Creates a NumPy array of integers within a specified range (inclusive) with a
given step size.

    Parameters:
    start (int): The starting value of the range.
    stop (int): The end value of the range (inclusive).
    step (int): The step size between values.

    Returns:
    numpy.ndarray: A NumPy array of integers within the specified range.
    """
    if step <= 0:
        raise ValueError("Step size must be a positive integer.")

    # Create the array using np.arange and then adjust the stop value to be
inclusive
    return np.arange(start, stop + 1, step)

# Example usage:
if __name__ == "__main__":
    # Create arrays with different ranges and step sizes
    range_array1 = create_range_array(0, 10, 2)   # Example with step size 2
    range_array2 = create_range_array(-5, 5, 3)   # Example with step size 3
    range_array3 = create_range_array(10, 30, 5)  # Example with step size 5

    # Print the created arrays
    print("Range Array 1:", range_array1)
    print("Range Array 2:", range_array2)
    print("Range Array 3:", range_array3)
```

#question 22. Write a Python function using NumPy to generate an array of 10 equally spaced values between 0 and 1
(inclusive).

```
import numpy as np

def generate_linspace_array(start, stop, num):
    """
    Generates a NumPy array of equally spaced values between start and stop
```

```
(inclusive).

    Parameters:
    start (float): The starting value of the range.
    stop (float): The end value of the range (inclusive).
    num (int): The number of equally spaced values to generate.

    Returns:
    numpy.ndarray: A NumPy array of equally spaced values.
    """
    if num <= 0:
        raise ValueError("Number of values must be a positive integer.")

    return np.linspace(start, stop, num)

# Example usage:
if __name__ == "__main__":
    # Generate an array of 10 equally spaced values between 0 and 1
    linspace_array = generate_linspace_array(0, 1, 10)

    # Print the generated array
    print("Linspace Array:", linspace_array)
```

```
#question 23. Write a Python function using NumPy to create an array of 5
logarithmically spaced values between 1 and
1000 (inclusive).

import numpy as np

def create_logspace_array(start, stop, num):
    """
    Creates a NumPy array of logarithmically spaced values between start and stop
(inclusive).

    Parameters:
    start (float): The starting exponent (base 10^start).
    stop (float): The ending exponent (base 10^stop).
    num (int): The number of logarithmically spaced values to generate.

    Returns:
    numpy.ndarray: A NumPy array of logarithmically spaced values.
    """
    if num <= 0:
        raise ValueError("Number of values must be a positive integer.")
    return np.logspace(start, stop, num)

# Example usage:
if __name__ == "__main__":
    # Create an array of 5 logarithmically spaced values between 1 and 1000
    logspace_array = create_logspace_array(0, 3, 5)

    # Print the generated array
    print("Logspace Array:", logspace_array)

#question 24. Create a Pandas DataFrame using a NumPy array that contains 5 rows
and 3 columns, where the values
are random integers between 1 and 100.

import numpy as np
import pandas as pd

def create_dataframe(rows, cols):
    """
    Creates a Pandas DataFrame from a NumPy array of random integers.

    Parameters:
    rows (int): The number of rows in the DataFrame.
    cols (int): The number of columns in the DataFrame.

    Returns:
    pandas.DataFrame: A DataFrame with random integer values.
    """
    # Generate a NumPy array with random integers between 1 and 100
    data = np.random.randint(1, 101, size=(rows, cols))

    # Create a DataFrame from the NumPy array
    df = pd.DataFrame(data, columns=[f'Column{i+1}' for i in range(cols)])

    return df

# Example usage:
if __name__ == "__main__":
    # Create a DataFrame with 5 rows and 3 columns
    df = create_dataframe(5, 3)

    # Print the DataFrame
    print("DataFrame:\n", df)
```

```
#question 25. Write a function that takes a Pandas DataFrame and replaces all
negative values in a specific column
with zeros. Use NumPy operations within the Pandas DataFrame.

import numpy as np
import pandas as pd

def replace_negatives_with_zeros(df, column_name):
    """
    Replaces all negative values in a specific column of a Pandas DataFrame with
zeros.

    Parameters:
    df (pandas.DataFrame): The DataFrame in which to replace negative values.
    column_name (str): The name of the column in which to replace negative values.

    Returns:
    pandas.DataFrame: The DataFrame with negative values replaced by zeros in the
specified column.
    """
    if column_name not in df.columns:
        raise ValueError(f"Column '{column_name}' does not exist in the
DataFrame.")

    # Replace negative values in the specified column with zeros
    df[column_name] = np.where(df[column_name] < 0, 0, df[column_name])

    return df

# Example usage:
if __name__ == "__main__":
    # Create a sample DataFrame
    data = {
        'A': [10, -5, 20, -15, 30],
        'B': [-1, -2, 3, 4, -5],
        'C': [100, 200, -300, 400, -500]
    }
    df = pd.DataFrame(data)

    # Replace negative values in column 'A'
    df_modified = replace_negatives_with_zeros(df, 'A')

    # Print the modified DataFrame
    print("Modified DataFrame:\n", df_modified)
```

```
#question 26. Access the 3rd element from the given NumPy array.

import numpy as np

# Given NumPy array
arr = np.array([10, 20, 30, 40, 50])

# Access the 3rd element (index 2)
third_element = arr[2]

# Print the 3rd element
print("The 3rd element is:", third_element)
```

```
#question 27. Retrieve the element at index (1, 2) from the 2D NumPy array.

import numpy as np

# Given 2D NumPy array
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Retrieve the element at index (1, 2)
element = arr_2d[1, 2]

# Print the element
print("The element at index (1, 2) is:", element)

#question 28

import numpy as np

# Given NumPy array
arr = np.array([3, 8, 2, 10, 5, 7])

# Create a boolean mask for elements greater than 5
mask = arr > 5

# Use the mask to extract elements greater than 5
filtered_elements = arr[mask]
```

```
# Print the result
print("Elements greater than 5:", filtered_elements)
```

```
#question 29. Perform basic slicing to extract elements from index 2 to 5
(inclusive) from the given NumPy array.

import numpy as np

# Given NumPy array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Perform basic slicing to extract elements from index 2 to 5 (inclusive)
sliced_elements = arr[2:6]  # End index is 6 because slicing is exclusive at the
end

# Print the result
print("Elements from index 2 to 5 (inclusive):", sliced_elements)
```

```
#question 30. Slice the 2D NumPy array to extract the sub-array `[[2, 3], [5, 6]]`
from the given array.

import numpy as np

# Given 2D NumPy array
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Slice the 2D array to extract the sub-array [[2, 3], [5, 6]]
# We need rows 1 to 2 (inclusive) and columns 1 to 2 (inclusive)
sub_array = arr_2d[1:3, 1:3]

# Print the sub-array
print("Sub-array:\n", sub_array)
```

```
#question 31.Write a NumPy function to extract elements in specific order from a
given 2D array based on indices
provided in another array.

import numpy as np

def extract_elements_by_indices(arr_2d, indices):
    """
    Extract elements from a 2D array based on provided indices.

    Parameters:
    arr_2d (numpy.ndarray): The 2D array from which to extract elements.
    indices (numpy.ndarray): A 2D array of indices specifying which elements to
extract.
                             Should have the same shape as arr_2d.

    Returns:
    numpy.ndarray: The extracted elements based on the provided indices.
    """
    # Ensure indices is a 2D array with the same shape as arr_2d
    if indices.shape != arr_2d.shape:
        raise ValueError("The shape of indices must match the shape of arr_2d.")

    # Extract elements using advanced indexing
    extracted_elements = arr_2d[indices]

    return extracted_elements

# Example usage:
if __name__ == "__main__":
    # Create a 2D NumPy array
    arr_2d = np.array([[10, 20, 30],
                       [40, 50, 60],
                       [70, 80, 90]])

    # Create an indices array to specify which elements to extract
    indices = np.array([[0, 1, 2],
                        [1, 0, 1],
                        [2, 1, 0]])

    # Extract elements based on indices
    result = extract_elements_by_indices(arr_2d, indices)

    # Print the result
    print("Extracted Elements:\n", result)
```

```
#question 32. Create a NumPy function that filters elements greater than a
threshold from a given 1D array using
boolean indexing.

import numpy as np

def filter_elements_greater_than_threshold(arr, threshold):
    """
    Filter elements in a 1D NumPy array that are greater than a specified
threshold.

    Parameters:
    arr (numpy.ndarray): The 1D array to filter.
    threshold (float): The threshold value. Elements greater than this value will
be kept.

    Returns:
    numpy.ndarray: An array containing elements greater than the specified
threshold.
    """
    # Ensure the input is a 1D array
    if arr.ndim != 1:
        raise ValueError("The input array must be 1-dimensional.")

    # Create a boolean mask for elements greater than the threshold
    mask = arr > threshold

    # Apply the mask to filter elements
    filtered_elements = arr[mask]

    return filtered_elements

# Example usage:
if __name__ == "__main__":
    # Create a sample 1D NumPy array
    arr = np.array([1, 5, 8, 12, 3, 7])

    # Define a threshold
    threshold = 5

    # Filter elements greater than the threshold
    result = filter_elements_greater_than_threshold(arr, threshold)

    # Print the result
    print("Elements greater than", threshold, ":", result)
```

```
#question 33. Develop a NumPy function that extracts specific elements from a 3D
array using indices provided in three
separate arrays for each dimension.

import numpy as np

def extract_elements_from_3d_array(arr_3d, indices_x, indices_y, indices_z):
    """
    Extract specific elements from a 3D NumPy array using indices provided for each
dimension.

    Parameters:
    arr_3d (numpy.ndarray): The 3D array from which to extract elements.
    indices_x (numpy.ndarray): 1D array of indices for the first dimension
(x-axis).
    indices_y (numpy.ndarray): 1D array of indices for the second dimension
(y-axis).
    indices_z (numpy.ndarray): 1D array of indices for the third dimension
(z-axis).

    Returns:
    numpy.ndarray: The extracted elements based on the provided indices.
    """
    # Ensure the input is a 3D array
    if arr_3d.ndim != 3:
        raise ValueError("The input array must be 3-dimensional.")

    # Ensure indices are 1D arrays with the same length
    if not (indices_x.ndim == indices_y.ndim == indices_z.ndim == 1):
        raise ValueError("All index arrays must be 1-dimensional.")
    if not (len(indices_x) == len(indices_y) == len(indices_z)):
        raise ValueError("All index arrays must have the same length.")

    # Extract elements using advanced indexing
    extracted_elements = arr_3d[indices_x, indices_y, indices_z]

    return extracted_elements

# Example usage:
if __name__ == "__main__":
    # Create a sample 3D NumPy array
    arr_3d = np.array([[[1, 2, 3],
                        [4, 5, 6],
                        [7, 8, 9]],

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

                       [[19, 20, 21],
                        [22, 23, 24],
                        [25, 26, 27]]])

    # Define indices for each dimension
    indices_x = np.array([0, 1, 2])  # Rows
    indices_y = np.array([1, 2, 0])  # Columns
    indices_z = np.array([2, 0, 1])  # Depth

    # Extract elements based on the indices
    result = extract_elements_from_3d_array(arr_3d, indices_x, indices_y,
indices_z)

    # Print the result
    print("Extracted Elements:", result)
```

```
#question 34. Write a NumPy function that returns elements from an array where both
two conditions are satisfied
using boolean indexing.

import numpy as np

def filter_elements_by_conditions(arr, condition1, condition2):
    """
    Filter elements in a NumPy array where both conditions are satisfied.

    Parameters:
    arr (numpy.ndarray): The array to filter.
    condition1 (callable): A function or lambda that returns a boolean array for
the first condition.
    condition2 (callable): A function or lambda that returns a boolean array for
the second condition.

    Returns:
    numpy.ndarray: An array containing elements that satisfy both conditions.
    """
    # Ensure the input is a NumPy array
    if not isinstance(arr, np.ndarray):
        raise ValueError("The input must be a NumPy array.")

    # Apply the conditions to create boolean masks
    mask1 = condition1(arr)
    mask2 = condition2(arr)

    # Combine the masks to satisfy both conditions
    combined_mask = mask1 & mask2

    # Filter the array using the combined mask
    filtered_elements = arr[combined_mask]

    return filtered_elements

# Example usage:
if __name__ == "__main__":
    # Create a sample NumPy array
    arr = np.array([1, 5, 8, 12, 3, 7, 10])

    # Define conditions
    condition1 = lambda x: x > 5  # Elements greater than 5
    condition2 = lambda x: x % 2 == 0  # Elements that are even

    # Filter elements based on the conditions
    result = filter_elements_by_conditions(arr, condition1, condition2)

    # Print the result
    print("Filtered Elements:", result)
```

#question 35. Create a NumPy function that extracts elements from a 2D array using row and column indices provided
in separate arrays.

```
import numpy as np

def extract_elements(arr_2d, row_indices, col_indices):
    """
    Extract elements from a 2D NumPy array using row and column indices provided in
separate arrays.

    Parameters:
    arr_2d (numpy.ndarray): The 2D array from which to extract elements.
    row_indices (numpy.ndarray): 1D array of row indices.
    col_indices (numpy.ndarray): 1D array of column indices.

    Returns:
    numpy.ndarray: An array containing elements at the specified row and column
indices.
    """
    # Ensure the input is a 2D array
    if arr_2d.ndim != 2:
        raise ValueError("The input array must be 2-dimensional.")

    # Ensure indices are 1D arrays with the same length
    if not (row_indices.ndim == col_indices.ndim == 1):
        raise ValueError("Both index arrays must be 1-dimensional.")
    if len(row_indices) != len(col_indices):
        raise ValueError("Row and column index arrays must have the same length.")

    # Extract elements using advanced indexing
    extracted_elements = arr_2d[row_indices, col_indices]

    return extracted_elements

# Example usage:
if __name__ == "__main__":
    # Create a sample 2D NumPy array
    arr_2d = np.array([[10, 20, 30],
                       [40, 50, 60],
                       [70, 80, 90]])

    # Define row and column indices
    row_indices = np.array([0, 1, 2])
    col_indices = np.array([2, 1, 0])

    # Extract elements based on the indices
    result = extract_elements(arr_2d, row_indices, col_indices)

    # Print the result
    print("Extracted Elements:", result)
```

```
#question 36. Given an array arr of shape (3, 3), add a scalar value of 5 to each
element using NumPy broadcasting.

import numpy as np

def add_scalar_to_array(arr, scalar):
    """
    Add a scalar value to each element of a 2D NumPy array using broadcasting.

    Parameters:
    arr (numpy.ndarray): The 2D array to which the scalar will be added.
    scalar (float): The scalar value to add to each element of the array.

    Returns:
    numpy.ndarray: The array after adding the scalar to each element.
    """
    # Add the scalar to the array. Broadcasting will handle the rest.
    result = arr + scalar
    return result

# Example usage:
if __name__ == "__main__":
    # Create a 2D NumPy array of shape (3, 3)
    arr = np.array([[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]])

    # Define the scalar value to add
    scalar = 5

    # Add the scalar to each element of the array
    result = add_scalar_to_array(arr, scalar)

    # Print the result
    print("Array after adding scalar:\n", result)
```

#question 37. Consider two arrays arr1 of shape (1, 3) and arr2 of shape (3, 4). Multiply each row of arr2 by the
corresponding element in arr1 using NumPy broadcasting.

```
import numpy as np

def multiply_rows_by_elements(arr1, arr2):
    """
    Multiply each row of arr2 by the corresponding element in arr1 using
broadcasting.

    Parameters:
    arr1 (numpy.ndarray): 1D array of shape (1, 3) containing the multipliers.
    arr2 (numpy.ndarray): 2D array of shape (3, 4) where each row will be
multiplied.
```

```
    Returns:
    numpy.ndarray: The resulting array after multiplication.
    """
    # Ensure arr1 is reshaped to (3, 1) for broadcasting
    arr1_reshaped = arr1.reshape(-1, 1)

    # Perform the multiplication using broadcasting
    result = arr2 * arr1_reshaped

    return result

# Example usage:
if __name__ == "__main__":
    # Create the arrays
    arr1 = np.array([[2, 3, 4]])  # Shape (1, 3)
    arr2 = np.array([[1, 2, 3, 4],
                     [5, 6, 7, 8],
                     [9, 10, 11, 12]])  # Shape (3, 4)

    # Multiply each row of arr2 by the corresponding element in arr1
    result = multiply_rows_by_elements(arr1, arr2)

    # Print the result
    print("Resulting Array:\n", result)
```

#quesion 38. Given a 1D array arr1 of shape (1, 4) and a 2D array arr2 of shape (4, 3), add arr1 to each row of arr2 using
NumPy broadcasting.

```
import numpy as np

def add_array_to_rows(arr1, arr2):
    """
    Add a 1D array to each row of a 2D array using broadcasting.

    Parameters:
    arr1 (numpy.ndarray): 1D array of shape (1, 4) to be added to each row of arr2.
    arr2 (numpy.ndarray): 2D array of shape (4, 3) to which arr1 will be added.

    Returns:
    numpy.ndarray: The resulting array after adding arr1 to each row of arr2.
    """
    # Ensure arr1 is reshaped to (4, 1) for broadcasting
    arr1_reshaped = arr1.reshape(-1, 1)

    # Perform the addition using broadcasting
    result = arr2 + arr1_reshaped

    return result

# Example usage:
if __name__ == "__main__":
    # Create the arrays
    arr1 = np.array([[1, 2, 3, 4]])  # Shape (1, 4)
    arr2 = np.array([[5, 6, 7],
                     [8, 9, 10],
                     [11, 12, 13],
                     [14, 15, 16]])  # Shape (4, 3)

    # Add arr1 to each row of arr2
    result = add_array_to_rows(arr1, arr2)

    # Print the result
    print("Resulting Array:\n", result)
```

```
#question 39. Consider two arrays arr1 of shape (3, 1) and arr2 of shape (1, 3).
Add these arrays using NumPy
broadcasting.
import numpy as np

def add_arrays_with_broadcasting(arr1, arr2):
    """
    Add two arrays using broadcasting.

    Parameters:
    arr1 (numpy.ndarray): Array of shape (3, 1).
    arr2 (numpy.ndarray): Array of shape (1, 3).

    Returns:
    numpy.ndarray: The resulting array after addition.
    """
    # Perform the addition using broadcasting
    result = arr1 + arr2

    return result
```

```
# Example usage:
if __name__ == "__main__":
    # Create the arrays
    arr1 = np.array([[1],
                     [2],
                     [3]])  # Shape (3, 1)
    arr2 = np.array([[4, 5, 6]])  # Shape (1, 3)

    # Add arr1 to arr2 using broadcasting
    result = add_arrays_with_broadcasting(arr1, arr2)

    # Print the result
    print("Resulting Array:\n", result)
```

```
#question 40. Given arrays arr1 of shape (2, 3) and arr2 of shape (2, 2), perform
multiplication using NumPy
broadcasting. Handle the shape incompatibility.

import numpy as np

def multiply_with_padding(arr1, arr2):
    """
    Perform multiplication of two arrays by padding the smaller array to match the
larger one using broadcasting.

    Parameters:
    arr1 (numpy.ndarray): Array of shape (2, 3).
    arr2 (numpy.ndarray): Array of shape (2, 2).

    Returns:
    numpy.ndarray: The resulting array after multiplication.
    """
    # Determine the new shape
    new_shape = (max(arr1.shape[0], arr2.shape[0]), max(arr1.shape[1],
arr2.shape[1]))

    # Pad arr1 to match the new shape
    arr1_padded = np.pad(arr1, ((0, new_shape[0] - arr1.shape[0]), (0, new_shape[1]
- arr1.shape[1])), mode='constant', constant_values=1)

    # Pad arr2 to match the new shape
    arr2_padded = np.pad(arr2, ((0, new_shape[0] - arr2.shape[0]), (0, new_shape[1]
- arr2.shape[1])), mode='constant', constant_values=1)
```

```
    # Perform element-wise multiplication using broadcasting
    result = arr1_padded * arr2_padded

    return result

# Example usage:
if __name__ == "__main__":
    # Create the arrays
    arr1 = np.array([[1, 2, 3],
                     [4, 5, 6]])  # Shape (2, 3)
    arr2 = np.array([[7, 8],
                     [9, 10]])  # Shape (2, 2)

    # Multiply with padding
    result = multiply_with_padding(arr1, arr2)

    # Print the result
    print("Resulting Array:\n", result)
```

```
#quesion 41. Calculate column wise mean for the given array:

import numpy as np

def column_wise_mean(arr):
    """
    Calculate the column-wise mean of a 2D NumPy array.

    Parameters:
    arr (numpy.ndarray): A 2D array.

    Returns:
    numpy.ndarray: An array containing the mean of each column.
    """
    # Calculate column-wise mean
    mean_values = np.mean(arr, axis=0)
    return mean_values

# Example usage:
if __name__ == "__main__":
    # Create a 2D NumPy array
    arr = np.array([[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]])
    # Calculate the column-wise mean
    result = column_wise_mean(arr)

    # Print the result
    print("Column-wise mean:", result)
```

```
#questio 42. Find maximum value in each row of the given array:
arr = np.array([[1, 2, 3], [4, 5, 6]])

import numpy as np

def row_max_values(arr):
    """
    Find the maximum value in each row of a 2D NumPy array.

    Parameters:
    arr (numpy.ndarray): A 2D array.

    Returns:
    numpy.ndarray: An array containing the maximum value of each row.
    """
    # Find the maximum value in each row
    max_values = np.amax(arr, axis=1)
    return max_values

# Example usage:
if __name__ == "__main__":
    # Create a 2D NumPy array
    arr = np.array([[1, 2, 3],
                    [4, 5, 6]])

    # Find the maximum values in each row
    result = row_max_values(arr)

    # Print the result
    print("Maximum values in each row:", result)
```

```
#question 43. For the given array, find indices of maximum value in each column.

import numpy as np

def column_max_indices(arr):
    """
    Find the indices of the maximum values in each column of a 2D NumPy array.

    Parameters:
    arr (numpy.ndarray): A 2D array.

    Returns:
    numpy.ndarray: An array containing the indices of the maximum value in each
column.
    """
    # Find the indices of the maximum values in each column
    max_indices = np.argmax(arr, axis=0)
    return max_indices

# Example usage:
if __name__ == "__main__":
    # Create a 2D NumPy array
    arr = np.array([[1, 2, 3],
                    [4, 5, 6]])

    # Find the indices of the maximum values in each column
    result = column_max_indices(arr)

    # Print the result
    print("Indices of maximum values in each column:", result)
```

```
#question 44. For the given array, apply custom function to calculate moving sum
along rows.


import numpy as np

def moving_sum(arr, window_size):
    """
    Calculate the moving sum along rows of a 2D NumPy array.

    Parameters:
    arr (numpy.ndarray): A 2D array.
    window_size (int): The size of the moving window.

    Returns:
    numpy.ndarray: A 2D array containing the moving sums.
    """
    # Number of rows and columns
    num_rows, num_cols = arr.shape

    # Initialize the result array
    result = np.zeros((num_rows, num_cols - window_size + 1), dtype=int)

    # Calculate the moving sum for each row
    for i in range(num_rows):
        for j in range(num_cols - window_size + 1):
            result[i, j] = np.sum(arr[i, j:j + window_size])

    return result

# Example usage:
if __name__ == "__main__":
    # Create a 2D NumPy array
    arr = np.array([[1, 2, 3],
                    [4, 5, 6]])

    # Define the window size
    window_size = 2

    # Calculate the moving sum along rows
    result = moving_sum(arr, window_size)

    # Print the result
    print("Moving sum along rows with window size", window_size, ":\n", result)
```

```
#question 45. In the given array, check if all elements in each column are even.

import numpy as np

def are_all_elements_even(arr):
    """
    Check if all elements in each column of a 2D NumPy array are even.

    Parameters:
    arr (numpy.ndarray): A 2D array.
    Returns:
    numpy.ndarray: A boolean array where each element indicates whether all
elements
                    in the corresponding column are even.
    """
    # Check if each element is even
    even_mask = (arr % 2 == 0)

    # Check if all elements in each column are even
    all_even = np.all(even_mask, axis=0)

    return all_even

# Example usage:
if __name__ == "__main__":
    # Create a 2D NumPy array
    arr = np.array([[2, 4, 6],
                    [3, 5, 7]])

    # Check if all elements in each column are even
    result = are_all_elements_even(arr)

    # Print the result
    print("Are all elements in each column even?", result)
```

```
#question 46. Given a NumPy array arr, reshape it into a matrix of dimensions `m`
rows and `n` columns. Return the
reshaped matrix.


import numpy as np

def reshape_array(arr, m, n):
    """
    Reshape a 1D NumPy array into a matrix of dimensions m x n.

    Parameters:
    arr (numpy.ndarray): The original 1D array to be reshaped.
    m (int): The number of rows in the reshaped matrix.
    n (int): The number of columns in the reshaped matrix.

    Returns:
```

```
    numpy.ndarray: The reshaped matrix.
    """
    # Reshape the array
    reshaped_matrix = np.reshape(arr, (m, n))

    return reshaped_matrix

# Example usage:
if __name__ == "__main__":
    # Create a 1D NumPy array
    original_array = np.array([1, 2, 3, 4, 5, 6])

    # Define the new dimensions
    m = 2  # number of rows
    n = 3  # number of columns

    # Reshape the array
    result = reshape_array(original_array, m, n)

    # Print the result
    print("Reshaped matrix:\n", result)
```

```
#question 47. Create a function that takes a matrix as input and returns the
flattened array.

import numpy as np

def flatten_matrix(matrix):
    """
    Flatten a 2D NumPy array into a 1D array.

    Parameters:
    matrix (numpy.ndarray): The 2D array to be flattened.

    Returns:
    numpy.ndarray: The flattened 1D array.
    """
    # Flatten the matrix
    flattened_array = matrix.flatten()
    return flattened_array

# Example usage:
```

```
if __name__ == "__main__":
    # Create a 2D NumPy array
    input_matrix = np.array([[1, 2, 3],
                             [4, 5, 6]])

    # Flatten the matrix
    result = flatten_matrix(input_matrix)

    # Print the result
    print("Flattened array:", result)
```

```
#question 48. Write a function that concatenates two given arrays along a specified
axis.
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])

import numpy as np

def concatenate_arrays(array1, array2, axis):
    """
    Concatenate two NumPy arrays along a specified axis.

    Parameters:
    array1 (numpy.ndarray): The first array to concatenate.
    array2 (numpy.ndarray): The second array to concatenate.
    axis (int): The axis along which to concatenate the arrays.
                Use 0 for vertical (row-wise) and 1 for horizontal (column-wise).

    Returns:
    numpy.ndarray: The concatenated array.
    """
    # Concatenate the arrays along the specified axis
    concatenated_array = np.concatenate((array1, array2), axis=axis)

    return concatenated_array

# Example usage:
if __name__ == "__main__":
    # Create two 2D NumPy arrays
    array1 = np.array([[1, 2],
                       [3, 4]])
    array2 = np.array([[5, 6],
                       [7, 8]])
    # Specify the axis for concatenation
    axis = 0  # Concatenate along rows (vertically)

    # Concatenate the arrays
    result = concatenate_arrays(array1, array2, axis)

    # Print the result
    print("Concatenated array along axis", axis, ":\n", result)
```

```
#question 49. Create a function that splits an array into multiple sub-arrays along
a specified axis

import numpy as np

def split_array(array, num_splits, axis):
    """
    Split a NumPy array into multiple sub-arrays along a specified axis.

    Parameters:
    array (numpy.ndarray): The array to be split.
    num_splits (int): The number of splits to create along the specified axis.
    axis (int): The axis along which to split the array.
                Use 0 for vertical splitting (rows) and 1 for horizontal splitting
(columns).

    Returns:
    list of numpy.ndarray: A list of sub-arrays obtained after splitting.
    """
    # Split the array along the specified axis
    split_arrays = np.array_split(array, num_splits, axis=axis)

    return split_arrays

# Example usage:
if __name__ == "__main__":
    # Create a 2D NumPy array
    original_array = np.array([[1, 2, 3],
                               [4, 5, 6],
                               [7, 8, 9]])

    # Specify the number of splits and axis
    num_splits = 3  # Number of splits
    axis = 1        # Split along columns (axis=1)
    # Split the array
    result = split_array(original_array, num_splits, axis)

    # Print the result
    for i, sub_array in enumerate(result):
        print(f"Sub-array {i}:\n{sub_array}\n")
```

#question 50. Write a function that inserts and then deletes elements from a given array at specified indices.

```
import numpy as np

def modify_array(array, indices_to_insert, values_to_insert, indices_to_delete):
    """
    Insert and then delete elements from a NumPy array at specified indices.

    Parameters:
    array (numpy.ndarray): The original array.
    indices_to_insert (list of int): The indices where elements should be inserted.
    values_to_insert (list of int): The values to insert at the specified indices.
    indices_to_delete (list of int): The indices of elements to delete after
insertion.

    Returns:
    numpy.ndarray: The modified array after insertion and deletion.
    """
    # Insert elements into the array
    modified_array = np.insert(array, indices_to_insert, values_to_insert)

    # Since deleting indices might change after insertion, sort the
indices_to_delete in reverse order
    sorted_delete_indices = sorted(indices_to_delete, reverse=True)

    # Delete elements from the array
    for index in sorted_delete_indices:
        modified_array = np.delete(modified_array, index)

    return modified_array

# Example usage:
if __name__ == "__main__":
    # Create the original NumPy array
    original_array = np.array([1, 2, 3, 4, 5])
    # Specify indices and values for insertion
    indices_to_insert = [2, 4]  # Indices where values will be inserted
    values_to_insert = [10, 11] # Values to insert at the specified indices

    # Specify indices for deletion
    indices_to_delete = [1, 3]  # Indices of elements to delete

    # Modify the array
    result = modify_array(original_array, indices_to_insert, values_to_insert,
indices_to_delete)

    # Print the result
    print("Modified array:", result)
```

```
#question 51. Create a NumPy array `arr1` with random integers and another array
`arr2` with integers from 1 to 10.
Perform element-wise addition between `arr1` and `arr2`.

import numpy as np

# Create arr1 with random integers
# Set the seed for reproducibility
np.random.seed(0)
arr1 = np.random.randint(0, 10, size=(3, 4))  # Example shape (3, 4)

# Create arr2 with integers from 1 to 10
arr2 = np.arange(1, 11).reshape((2, 5))  # Example shape (2, 5)

# Perform element-wise addition
# For demonstration purposes, ensure shapes are compatible for addition
# To make the shapes compatible, resize arr2 to match arr1's shape
arr2_resized = arr2[:3, :4]  # Resize arr2 to shape (3, 4)

# Perform element-wise addition
result = arr1 + arr2_resized

# Print the arrays and the result
print("arr1:\n", arr1)
print("arr2_resized:\n", arr2_resized)
print("Result of element-wise addition:\n", result)
```

```
#question 52. Generate a NumPy array `arr1` with sequential integers from 10 to 1
and another array `arr2` with integers
from 1 to 10. Subtract `arr2` from `arr1` element-wise.

import numpy as np

# Generate arr1 with sequential integers from 10 to 1
arr1 = np.arange(10, 0, -1).reshape((2, 5))  # Example shape (2, 5)

# Generate arr2 with integers from 1 to 10
arr2 = np.arange(1, 11).reshape((2, 5))  # Example shape (2, 5)

# Perform element-wise subtraction
result = arr1 - arr2

# Print the arrays and the result
print("arr1:\n", arr1)
print("arr2:\n", arr2)
print("Result of element-wise subtraction:\n", result)
```

```
#53. Create a NumPy array `arr1` with random integers and another array `arr2` with
integers from 1 to 5.
Perform element-wise multiplication between `arr1` and `arr2`.

import numpy as np

# Set the seed for reproducibility
np.random.seed(0)

# Generate arr1 with random integers
arr1 = np.random.randint(1, 10, size=(3, 4))  # Example shape (3, 4)

# Generate arr2 with integers from 1 to 5
arr2 = np.arange(1, 6).reshape((1, 5))  # Example shape (1, 5)

# Resize arr2 to match arr1's shape for element-wise multiplication
# Ensure arr2 has the same number of columns as arr1
arr2_resized = np.tile(arr2, (3, 1))  # Repeat arr2 to match arr1's shape
```

```
# Perform element-wise multiplication
result = arr1 * arr2_resized

# Print the arrays and the result
print("arr1:\n", arr1)
print("arr2_resized:\n", arr2_resized)
print("Result of element-wise multiplication:\n", result)
```

```
#question 54. Generate a NumPy array `arr1` with even integers from 2 to 10 and
another array `arr2` with integers from 1
to 5. Perform element-wise division of `arr1` by `arr2`.

import numpy as np

# Generate arr1 with even integers from 2 to 10
arr1 = np.arange(2, 11, 2).reshape((2, 2))  # Example shape (2, 2)

# Generate arr2 with integers from 1 to 5
arr2 = np.arange(1, 6).reshape((2, 3))  # Example shape (2, 3)

# Resize arr1 to match arr2's shape for element-wise division
# Ensure arr1 has the same number of columns as arr2
arr1_resized = np.tile(arr1, (1, 2))  # Repeat arr1 to match arr2's shape

# Perform element-wise division
result = arr1_resized / arr2

# Print the arrays and the result
print("arr1_resized:\n", arr1_resized)
print("arr2:\n", arr2)
print("Result of element-wise division:\n", result)
```

```
#question55. Create a NumPy array `arr1` with integers from 1 to 5 and another
array `arr2` with the same numbers
```

reversed. Calculate the exponentiation of `arr1` raised to the power of `arr2` element-wise.

```
import numpy as np

# Generate arr1 with integers from 1 to 5
arr1 = np.arange(1, 6)

# Generate arr2 with the same numbers reversed
arr2 = np.arange(5, 0, -1)

# Perform element-wise exponentiation
result = arr1 ** arr2

# Print the arrays and the result
print("arr1:", arr1)
print("arr2:", arr2)
print("Result of element-wise exponentiation:\n", result)
```

#question 56. Write a function that counts the occurrences of a specific substring within a NumPy array of strings.

```
import numpy as np

def count_substring_occurrences(arr, substring):
    # Convert the array to a 1D array if it's not already
    arr = arr.ravel()

    # Count occurrences of the substring in each string in the array
    count = sum(np.char.count(arr, substring))

    return count

# Example usage
arr = np.array(['hello', 'world', 'hello', 'numpy', 'hello'])
substring = 'hello'
occurrences = count_substring_occurrences(arr, substring)

print(f"The substring '{substring}' occurs {occurrences} times in the array.")
```

```
#question 57

import numpy as np

def extract_uppercase(arr):
    # Convert the array to a 1D array if it's not already
    arr = arr.ravel()

    # Function to extract uppercase characters from a single string
    def get_uppercase_chars(s):
        return ''.join(c for c in s if c.isupper())

    # Apply the function to each element in the array
    uppercase_chars = np.vectorize(get_uppercase_chars)(arr)

    return uppercase_chars

# Example usage
arr = np.array(['Hello', 'World', 'OpenAI', 'GPT'])
uppercase_chars = extract_uppercase(arr)

print("Uppercase characters extracted from the array:")
print(uppercase_chars)
```

```
#question 58. Write a function that replaces occurrences of a substring in a NumPy
array of strings with a new string.

import numpy as np

def replace_substring(arr, old_substring, new_substring):
    # Convert the array to a 1D array if it's not already
    arr = arr.ravel()

    # Replace old_substring with new_substring in each element of the array
    replaced_arr = np.char.replace(arr, old_substring, new_substring)

    return replaced_arr
```

```
# Example usage
arr = np.array(['apple', 'banana', 'grape', 'pineapple'])
old_substring = 'apple'
new_substring = 'orange'
result = replace_substring(arr, old_substring, new_substring)

print("Array after replacement:")
print(result)
```

```
#question  59

import numpy as np

def concatenate_strings(arr1, arr2):
    # Ensure both arrays are 1D
    arr1 = arr1.ravel()
    arr2 = arr2.ravel()

    # Perform element-wise concatenation
    concatenated_arr = np.char.add(arr1, arr2)

    return concatenated_arr

# Example usage
arr1 = np.array(['Hello', 'World'])
arr2 = np.array(['Open', 'AI'])
result = concatenate_strings(arr1, arr2)

print("Array after concatenation:")
print(result)
```

```
#question 60. Write a function that finds the length of the longest string in a
NumPy array.

arr = np.array(['apple', 'banana', 'grape', 'pineapple'])
arr = np.array(['apple', 'banana', 'grape', 'pineapple'])
```

```
import numpy as np

def find_max_string_length(arr):
    # Ensure the array is 1D
    arr = arr.ravel()

    # Find the length of each string in the array
    lengths = np.char.str_len(arr)

    # Return the maximum length
    max_length = np.max(lengths)

    return max_length

# Example usage
arr = np.array(['apple', 'banana', 'grape', 'pineapple'])
max_length = find_max_string_length(arr)

print("The length of the longest string is:", max_length)
```

question 61. Create a dataset of 100 random integers between 1 and 1000. Compute the mean, median, variance, and
standard deviation of the dataset using NumPy's functions.

```
import numpy as np

# Generate a dataset of 100 random integers between 1 and 1000
dataset = np.random.randint(1, 1001, size=100)

# Compute statistical properties
mean = np.mean(dataset)
median = np.median(dataset)
variance = np.var(dataset)
std_dev = np.std(dataset)

# Print the results
print("Dataset:", dataset)
print("Mean:", mean)
print("Median:", median)
print("Variance:", variance)
print("Standard Deviation:", std_dev)
```

```
#question 62. Generate an array of 50 random numbers between 1 and 100. Find the
25th and 75th percentiles of the
dataset

import numpy as np

# Generate an array of 50 random integers between 1 and 100
dataset = np.random.randint(1, 101, size=50)

# Compute the 25th and 75th percentiles
percentile_25 = np.percentile(dataset, 25)
percentile_75 = np.percentile(dataset, 75)

# Print the results
print("Dataset:", dataset)
print("25th Percentile:", percentile_25)
print("75th Percentile:", percentile_75)
```

```
#question 63. Create two arrays representing two sets of variables. Compute the
correlation coefficient between these
arrays using NumPy's `corrcoef` function.


import numpy as np

# Create two arrays representing two sets of variables
array1 = np.array([1, 2, 3, 4, 5])
array2 = np.array([2, 4, 6, 8, 10])

# Compute the correlation coefficient matrix
correlation_matrix = np.corrcoef(array1, array2)

# Extract the correlation coefficient between the two arrays
correlation_coefficient = correlation_matrix[0, 1]

# Print the results
print("Correlation Matrix:")
print(correlation_matrix)
print("Correlation Coefficient between array1 and array2:",
correlation_coefficient)
```

```
#question 64. Create two matrices and perform matrix multiplication using NumPy's
`dot` function.

import numpy as np

# Define two matrices
matrix1 = np.array([[1, 2, 3],
                    [4, 5, 6]])

matrix2 = np.array([[7, 8],
                    [9, 10],
                    [11, 12]])

# Perform matrix multiplication using np.dot
result = np.dot(matrix1, matrix2)

# Print the result
print("Matrix 1:")
print(matrix1)
print("Matrix 2:")
print(matrix2)
print("Result of Matrix Multiplication:")
print(result)
```

```
#question 65. Create an array of 50 integers between 10 and 1000. Calculate the
10th, 50th (median), and 90th
percentiles along with the first and third quartiles.

import numpy as np

# Generate an array of 50 random integers between 10 and 1000
data = np.random.randint(10, 1001, size=50)
```

```
# Compute percentiles and quartiles
percentile_10 = np.percentile(data, 10)
percentile_50 = np.percentile(data, 50)  # This is also the median
percentile_90 = np.percentile(data, 90)
quartile_1 = np.percentile(data, 25)  # 1st Quartile
quartile_3 = np.percentile(data, 75)  # 3rd Quartile

# Print the results
print("Dataset:", data)
print("10th Percentile:", percentile_10)
print("50th Percentile (Median):", percentile_50)
print("90th Percentile:", percentile_90)
print("1st Quartile (25th Percentile):", quartile_1)
print("3rd Quartile (75th Percentile):", quartile_3)
```

#question 66. Create a NumPy array of integers and find the index of a specific element.

```
import numpy as np

# Create a NumPy array of integers
array = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100])

# Specify the element to find
element_to_find = 40

# Find the index of the specific element
indices = np.where(array == element_to_find)

# Since np.where returns a tuple of arrays, extract the first element
index = indices[0]

# Print the result
print("Array:", array)
print(f"Index of element {element_to_find}:", index)
```

#question 67. Generate a random NumPy array and sort it in ascending order.

```
import numpy as np

# Generate a random NumPy array with integers
array = np.random.randint(1, 100, size=10)  # Random integers between 1 and 100,
array of size 10

# Sort the array in ascending order
sorted_array = np.sort(array)

# Print the results
print("Original Array:", array)
print("Sorted Array:", sorted_array)
```

```
#question 68. Filter elements >20  in the given NumPy array.

import numpy as np

# Define the NumPy array
arr = np.array([12, 25, 6, 42, 8, 30])

# Filter elements greater than 20
filtered_elements = arr[arr > 20]

# Print the results
print("Original Array:", arr)
print("Filtered Elements (>20):", filtered_elements)
```

```
#question-69. Filter elements which are divisible by 3 from a given NumPy array.

import numpy as np

# Define the NumPy array
arr = np.array([1, 5, 8, 12, 15])

# Filter elements that are divisible by 3
divisible_by_3 = arr[arr % 3 == 0]
```

```
# Print the results
print("Original Array:", arr)
print("Elements Divisible by 3:", divisible_by_3)
```

```
#question 70. Filter elements which are ≥ 20 and ≤ 40 from a given NumPy array.

import numpy as np

# Define the NumPy array
arr = np.array([10, 20, 30, 40, 50])

# Filter elements that are >= 20 and <= 40
filtered_elements = arr[(arr >= 20) & (arr <= 40)]

# Print the results
print("Original Array:", arr)
print("Filtered Elements (20 ≤ x ≤ 40):", filtered_elements)
```

```
#question 71. For the given NumPy array, check its byte order using the `dtype`
attribute byteorder.

import numpy as np

# Define the NumPy array
arr = np.array([1, 2, 3])

# Check the byte order
byte_order = arr.dtype.byteorder

# Print the results
print("Array:", arr)
print("Byte Order:", byte_order)

# Interpret the byte order
if byte_order == '=':
    print("Byte order: Native (little-endian or big-endian depending on the
system)")
elif byte_order == '<':
    print("Byte order: Little-endian")
elif byte_order == '>':
    print("Byte order: Big-endian")
elif byte_order == '|':
    print("Byte order: Not applicable (for character data)")
else:
    print("Byte order: Unknown")
```

```
#question 72. For the given NumPy array, perform byte swapping in place using
`byteswap()`.

import numpy as np

# Define the NumPy array with dtype=np.int32
arr = np.array([1, 2, 3], dtype=np.int32)

# Display the original array and its byte order
print("Original Array:", arr)
print("Original Byte Order:", arr.dtype.byteorder)

# Perform byte swapping in place
arr.byteswap(inplace=True)

# Display the array after byte swapping
print("Array after Byte Swapping:", arr)
print("Byte Order after Byte Swapping:", arr.dtype.byteorder)
```

question 73

```
import numpy as np
```

```
# Define the NumPy array with dtype=np.int32
arr = np.array([1, 2, 3], dtype=np.int32)

# Display the original array and its byte order
print("Original Array:", arr)
print("Original Byte Order:", arr.dtype.byteorder)

# Swap the byte order using newbyteorder
arr_swapped = arr.newbyteorder()

# Display the array after byte swapping
print("Array with Swapped Byte Order:", arr_swapped)
print("Byte Order of Swapped Array:", arr_swapped.dtype.byteorder)
```

```
#question 74. For the given NumPy array and swap its byte order conditionally based
on system endianness using
`newbyteorder()`.


import numpy as np
import sys

def swap_byteorder_conditionally(arr):
    # Check the system's endianness
    system_endianness = sys.byteorder

    # Determine the byte order to use based on system endianness
    if system_endianness == 'little':
        # System is little-endian, so convert to big-endian
        return arr.newbyteorder('>')
    else:
        # System is big-endian, so convert to little-endian
        return arr.newbyteorder('<')

# Define the NumPy array with dtype=np.int32
arr = np.array([1, 2, 3], dtype=np.int32)

# Display the original array and its byte order
print("Original Array:", arr)
print("Original Byte Order:", arr.dtype.byteorder)

# Swap byte order conditionally
arr_swapped = swap_byteorder_conditionally(arr)
```

```
# Display the array after byte swapping
print("Array with Conditionally Swapped Byte Order:", arr_swapped)
print("Byte Order of Swapped Array:", arr_swapped.dtype.byteorder)
```

```
#question 75

import numpy as np
import sys

def check_byte_swapping_needed(arr):
    # Get the system's byte order
    system_byteorder = sys.byteorder

    # Get the array's byte order
    array_byteorder = arr.dtype.byteorder

    # Determine if byte swapping is needed
    if array_byteorder == '=':
        # Native byte order; no byte swapping needed
        return False
    elif (array_byteorder == '<' and system_byteorder == 'big') or \
         (array_byteorder == '>' and system_byteorder == 'little'):
        # Byte order of array and system do not match
        return True
    else:
        # Same byte order or other cases
        return False

# Define the NumPy array with dtype=np.int32
arr = np.array([1, 2, 3], dtype=np.int32)

# Check if byte swapping is necessary
byte_swapping_needed = check_byte_swapping_needed(arr)

# Display the results
print("Array Byte Order:", arr.dtype.byteorder)
print("System Byte Order:", sys.byteorder)
print("Is Byte Swapping Needed?", byte_swapping_needed)
```

```
#question 76. Create a NumPy array `arr1` with values from 1 to 10. Create a copy
of `arr1` named `copy_arr` and modify
an element in `copy_arr`. Check if modifying `copy_arr` affects `arr1`.

import numpy as np

# Step 1: Create the original array
arr1 = np.arange(1, 11)  # Values from 1 to 10

# Step 2: Create a copy of arr1
copy_arr = arr1.copy()

# Step 3: Modify an element in the copy
copy_arr[5] = 99  # Changing the 6th element (index 5) to 99

# Step 4: Check if modifying copy_arr affects arr1
print("Original Array (arr1):", arr1)
print("Modified Copy (copy_arr):", copy_arr)
```

```
#question 77. Create a 2D NumPy array `matrix` of shape (3, 3) with random
integers. Extract a slice `view_slice` from
the matrix. Modify an element in `view_slice` and observe if it changes the
original `matrix`.


import numpy as np

# Step 1: Create a 2D NumPy array with random integers
np.random.seed(0)  # For reproducibility
matrix = np.random.randint(1, 100, size=(3, 3))

# Step 2: Extract a slice from the matrix
view_slice = matrix[1:, 1:]  # Slice from row 1 to end, column 1 to end

# Print the original matrix and the slice before modification
print("Original Matrix:")
print(matrix)
print("\nSlice Before Modification:")
print(view_slice)

# Step 3: Modify an element in the slice
```

```
view_slice[0, 0] = 999  # Modify the top-left element of the slice

# Print the modified slice and the original matrix
print("\nSlice After Modification:")
print(view_slice)
print("\nOriginal Matrix After Modification:")
print(matrix)
```

#question 78. Create a NumPy array `array_a` of shape (4, 3) with sequential integers from 1 to 12. Extract a slice
`view_b` from `array_a` and broadcast the addition of 5 to view_b. Check if it alters the original `array_a`.

```
import numpy as np

# Step 1: Create the original array with sequential integers from 1 to 12
array_a = np.arange(1, 13).reshape((4, 3))

# Step 2: Extract a slice from the array
view_b = array_a[1:, 1:]  # Slice from row index 1 to end, column index 1 to end

# Print the original array and the slice before broadcasting
print("Original Array (array_a):")
print(array_a)
print("\nSlice Before Broadcasting:")
print(view_b)

# Step 3: Broadcast the addition of 5 to the slice
view_b += 5  # This is equivalent to view_b = view_b + 5

# Print the modified slice and the original array after broadcasting
print("\nSlice After Broadcasting:")
print(view_b)
print("\nOriginal Array After Broadcasting:")
print(array_a)
```

```
#question 79. Create a NumPy array `orig_array` of shape (2, 4) with values from 1
to 8. Create a reshaped view
`reshaped_view` of shape (4, 2) from orig_array. Modify an element in
`reshaped_view` and check if it
reflects changes in the original `orig_array`.


import numpy as np

# Step 1: Create the original array with values from 1 to 8
orig_array = np.arange(1, 9).reshape((2, 4))

# Step 2: Create a reshaped view of shape (4, 2)
reshaped_view = orig_array.reshape((4, 2))

# Print the original array and reshaped view before modification
print("Original Array (orig_array):")
print(orig_array)
print("\nReshaped View Before Modification:")
print(reshaped_view)

# Step 3: Modify an element in the reshaped view
reshaped_view[0, 0] = 99  # Modify the top-left element of the reshaped view

# Print the reshaped view and the original array after modification
print("\nReshaped View After Modification:")
print(reshaped_view)
print("\nOriginal Array After Modification:")
print(orig_array)
```

```
#question 80. Create a NumPy array `data` of shape (3, 4) with random integers.
Extract a copy `data_copy` of
elements greater than 5. Modify an element in `data_copy` and verify if it affects
the original `data`.

import numpy as np

# Step 1: Create the original array with random integers
np.random.seed(0)  # For reproducibility
data = np.random.randint(1, 10, size=(3, 4))

# Step 2: Extract a copy of elements greater than 5
```

```
data_copy = data[data > 5].copy()  # Use .copy() to ensure a separate copy

# Print the original array and data_copy before modification
print("Original Array (data):")
print(data)
print("\nData Copy Before Modification:")
print(data_copy)

# Step 3: Modify an element in the data_copy
data_copy[0] = 99  # Modify the first element in the copy

# Print the modified data_copy and the original array after modification
print("\nData Copy After Modification:")
print(data_copy)
print("\nOriginal Array After Modification:")
print(data)
```

```
#question 81. Create two matrices A and B of identical shape containing integers
and perform addition and subtraction
operations between them.

import numpy as np

# Step 1: Create two matrices A and B of identical shape with random integers
np.random.seed(0)  # For reproducibility
A = np.random.randint(1, 10, size=(3, 3))  # Matrix A
B = np.random.randint(1, 10, size=(3, 3))  # Matrix B

# Print matrices A and B
print("Matrix A:")
print(A)
print("\nMatrix B:")
print(B)

# Step 2: Perform element-wise addition
addition_result = A + B

# Step 3: Perform element-wise subtraction
subtraction_result = A - B

# Print the results of addition and subtraction
print("\nAddition Result (A + B):")
print(addition_result)
print("\nSubtraction Result (A - B):")
print(subtraction_result)
```

```
#question 82. Generate two matrices `C` (3x2) and `D` (2x4) and perform matrix
multiplication.

import numpy as np

# Step 1: Create matrices C and D
C = np.random.randint(1, 10, size=(3, 2))  # Matrix C with shape (3, 2)
D = np.random.randint(1, 10, size=(2, 4))  # Matrix D with shape (2, 4)

# Print matrices C and D
print("Matrix C:")
print(C)
print("\nMatrix D:")
print(D)

# Step 2: Perform matrix multiplication
# Using np.dot()
result_dot = np.dot(C, D)

# Alternatively, using the @ operator
result_at = C @ D

# Print the results
print("\nMatrix Multiplication Result (using np.dot()):")
print(result_dot)
print("\nMatrix Multiplication Result (using @ operator):")
print(result_at)
```

```
#question 83. Create a matrix `E` and find its transpose.

import numpy as np

# Step 1: Create a matrix E
E = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  # Matrix E of shape (3, 3)

# Print the original matrix E
```

```
print("Matrix E:")
print(E)

# Step 2: Find the transpose of matrix E
E_transposed = E.T  # Using the T attribute

# Alternatively, using the transpose function
E_transposed_func = np.transpose(E)

# Print the transpose result
print("\nTranspose of Matrix E (using T attribute):")
print(E_transposed)
print("\nTranspose of Matrix E (using np.transpose() function):")
print(E_transposed_func)
```

```
#question 84. Generate a square matrix `F` and compute its determinant.

import numpy as np

# Step 1: Generate a square matrix F
F = np.array([[4, 2, 1],
              [3, 5, 2],
              [1, 3, 3]])  # Example 3x3 matrix

# Print the original matrix F
print("Matrix F:")
print(F)

# Step 2: Compute the determinant of matrix F
det_F = np.linalg.det(F)

# Print the determinant result
print("\nDeterminant of Matrix F:")
print(det_F)
```

```
#question 85. Create a square matrix `G` and find its inverse.
```

```
import numpy as np

# Step 1: Create a square matrix G
G = np.array([[4, 7],
              [2, 6]])  # Example 2x2 matrix

# Print the original matrix G
print("Matrix G:")
print(G)

# Step 2: Compute the inverse of matrix G
# Check if the matrix is invertible
if np.linalg.det(G) != 0:
    G_inverse = np.linalg.inv(G)
    # Print the inverse
    print("\nInverse of Matrix G:")
    print(G_inverse)
else:
    print("\nMatrix G is not invertible (its determinant is zero).")
```