Que_1. Create a NumPy array 'arr' of integers from 0 to 5 and print its data type.

```
import numpy as np
arr = np.arange(6) #Create an array from 0 to 5 (inclusive)
print(arr)
print(arr.dtype)
```

Que_2. Given a NumPy array 'arr', check if its data type is float64. arr = np.array([1.5, 2.6, 3.7])

```
arr = np.array([1.5, 2.6, 3.7])
if arr.dtype == np.float64:
    print("This data type of arr is float64")
else:
    print("This data type of arr is not float64")
```

Que_3. Create a NumPy array 'arr' with a data type of complex128 containing three complex numbers.

```
arr = np.array([3+5j,9+8j,2+1j], dtype= np.complex128)
print(arr)
print(arr.dtype)
```

Que_4. Convert an existing NumPy array 'arr' of integers to float32 data type.

```
arr = np.array([2,3,5,6,8]) #create integers array
print(arr.dtype) #output int64

arr_float = arr.astype(np.float32) #convert integers to float32
print(arr_float)
print(arr_float.dtype) #output float32
```

Que_5. Given a NumPy array 'arr' with float64 data type, convert it to float32 to reduce decimal precision.

```
import numpy as np

arr = np.array([1.23456789, 2.34567890, 3.45678901], dtype=np.float64)
print(arr.dtype)  # Output: float64

# Convert the array to float32
arr_float32 = arr.astype(np.float32)
print(arr_float32.dtype)  # Output: float32
```

Que_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 of a NumPy array.

  Args:
    arr: The NumPy array.

  Returns:
    A tuple containing the shape, size, and data of the array.
  """

  shape = arr.shape
  size = arr.size
  data = arr.tolist()

  return shape, size, data

# Example usage
arr = np.array([[1, 2, 3], [4, 5, 6]])
shape, size, data = array_attributes(arr)
print("Shape:", shape)
print("Size:", size)
print("Data:", data)
```

Que_7. 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 a NumPy array.

  Args:
    arr: The NumPy array.

  Returns:
    The dimensionality of the array.
  """

  return arr.ndim

# Example usage
arr = np.array([[1, 2, 3], [4, 5, 6]])
dimension = array_dimension(arr)
print("Dimensionality:", dimension)
```

Que_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 a NumPy array.

  Args:
    arr: The NumPy array.

  Returns:
    A tuple containing the item size and total size in bytes.
  """
```

```
  item_size = arr.itemsize
  total_size = arr.nbytes

  return item_size, total_size

# Example usage
arr = np.array([[1, 2, 3], [4, 5, 6]])
item_size, total_size = item_size_info(arr)
print("Item size:", item_size, "bytes")
print("Total size:", total_size, "bytes")
```

Que_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 a NumPy array.

  Args:
    arr: The NumPy array.

  Returns:
    The strides of the array.
  """

  return arr.strides

# Example usage
arr = np.array([[1, 2, 3], [4, 5, 6]])
strides = array_strides(arr)
print("Strides:", strides)
```

Que_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 a NumPy array.

  Args:
    arr: The NumPy array.

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

  shape = arr.shape
  strides = arr.strides

  return shape, strides

# Example usage
arr = np.array([[1, 2, 3], [4, 5, 6]])
shape, strides = shape_stride_relationship(arr)
print("Shape:", shape)
print("Strides:", strides)
```

Que_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.

  Args:
    n: The number of elements in the array.
```

```
  Returns:
    A NumPy array of zeros with n elements.
  """

  return np.zeros(n)

# Example usage
n = 5
zeros_array = create_zeros_array(n)
print(zeros_array)
```

Que_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.

  Args:
    rows: The number of rows in the matrix.
    cols: The number of columns in the matrix.

  Returns:
    A 2D NumPy array filled with ones.
  """

  return np.ones((rows, cols))

# Example usage
rows = 3
cols = 4
ones_matrix = create_ones_matrix(rows, cols)
print(ones_matrix)
```

Que_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):
  """Generates a NumPy array with a given range.

  Args:
    start: The starting value of the range.
    stop: The ending value of the range (exclusive).
    step: The step size between elements.

  Returns:
    A NumPy array with the specified range.
  """

  return np.arange(start, stop, step)

# Example usage
start = 2
stop = 10
step = 2
array = generate_range_array(start, stop, step)
print(array)
```

Que_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
```

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

  Args:
    start: The starting value of the sequence.
    stop: The ending value of the sequence.
    num: The number of samples to generate.

  Returns:
    A NumPy array with the specified range.
  """

  return np.linspace(start, stop, num)

# Example usage
start = 0.0
stop = 1.0
num = 10
array = generate_linear_space(start, stop, num)
print(array)
```

Que_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`.

```python
import numpy as np

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

  Args:
    n: The size of the matrix.

  Returns:
    A square identity matrix of size n x n.
  """

  return np.eye(n)

# Example usage
n = 3
identity_matrix = create_identity_matrix(n)
print(identity_matrix)
```

Que_16. Write a function that takes a Python list and converts it into a NumPy array.

```python
import numpy as np

def list_to_numpy_array(python_list):
  """Converts a Python list to a NumPy array.

  Args:
    python_list: The Python list to convert.

  Returns:
    A NumPy array containing the elements of the list.
  """

  return np.array(python_list)

# Example usage
my_list = [1, 2, 3, 4, 5]
```

```
numpy_array = list_to_numpy_array(my_list)
print(numpy_array)
```

Que_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
arr = np.array([1, 2, 3, 4, 5])

# Create a new array object with the same data using `view`
arr_view = arr.view()

# Print the original array and the view
print("Original array:", arr)
print("View array:", arr_view)

# Modify the original array
arr[0] = 100

# Print the modified original array and the view
print("Modified original array:", arr)
print("View array:", arr_view)
```

Que)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.

  Args:
```

```
    arr1: The first NumPy array.
    arr2: The second NumPy array.
    axis: The axis along which to concatenate the arrays.

  Returns:
    A new NumPy array containing the concatenated elements.
  """

  return np.concatenate((arr1, arr2), axis=axis)

# Example usage
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# Concatenate along the first axis (rows)
concatenated_arr1 = concatenate_arrays(arr1, arr2, axis=0)
print("Concatenated along rows:", concatenated_arr1)

# Concatenate along the second axis (columns)
concatenated_arr2 = concatenate_arrays(arr1, arr2, axis=1)
print("Concatenated along columns:", concatenated_arr2)
```

Que_19. Create two NumPy arrays with different shapes and concatenate them horizontally using `numpy. concatenate`

```
import numpy as np

# Create two NumPy arrays with different shapes
arr1 = np.array([[1, 2, 3], [4, 5, 6]])  # 2x3 array
arr2 = np.array([[7, 8], [9, 10]])  # 2x2 array

# Concatenate the arrays horizontally
concatenated_arr = np.concatenate((arr1, arr2), axis=1)

print(concatenated_arr)
```

Que_20. Write a function that vertically stacks multiple NumPy arrays given as a list.

```
import numpy as np

def vertical_stack(arrays):
  """Vertically stacks multiple NumPy arrays.

  Args:
    arrays: A list of NumPy arrays.

  Returns:
    A new NumPy array containing the vertically stacked arrays.
  """

  return np.vstack(arrays)

# Example usage
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([[7, 8], [9, 10]])
arr3 = np.array([[11, 12]])

stacked_arr = vertical_stack([arr1, arr2, arr3])
print(stacked_arr)
```

Que_21. Write a Python function using NumPy to create an array of integers within a specified range (inclusive) with a given step size.

```
def create_int_arr(start,stop,step=1):
  """ Create a numpy array of integers within a specified range
  Args:
      start : The starting value of the range
      stop : The stop value of the range (inclusive).
      step : The step size between elements

  Returns:
      A numpy array of integers within the specified range.
```

```
  """

  return np.arange(start, stop+1,step)

# Example usage
start = 2
stop = 10
step = 2
integer_array = create_integer_array(start, stop, step)
print(integer_array)
```

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 with equally spaced values.

  Args:
    start: The starting value of the sequence.
    stop: The ending value of the sequence.
    num: The number of samples to generate.

  Returns:
    A NumPy array with the specified range.
  """

  return np.linspace(start, stop, num)

# Example usage
start = 0.0
stop = 1.0
num = 10
array = generate_linspace_array(start, stop, num)
print(array)
```

Que)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 generate_logspace_array(start, stop, num):
  """Generates a NumPy array with logarithmically spaced values.

  Args:
    start: The starting value of the sequence.
    stop: The ending value of the sequence.
    num: The number of samples to generate.

  Returns:
    A NumPy array with the specified range.
  """

  return np.logspace(np.log10(start), np.log10(stop), num)

# Example usage
start = 1
stop = 1000
num = 5
array = generate_logspace_array(start, stop, num)
print(array)
```

Que)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

# Create a NumPy array with 5 rows and 3 columns
np_array = np.random.randint(1, 101, (5, 3))
```

```
# Create a Pandas DataFrame from the NumPy array
df = pd.DataFrame(np_array)

print(df)
```

Que_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 pandas as pd
import numpy as np

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

  Args:
    df: The Pandas DataFrame.
    column_name: The name of the column to modify.

  Returns:
    A new Pandas DataFrame with negative values replaced.
  """

  # Create a boolean mask for negative values in the specified column
  negative_mask = df[column_name] < 0

  # Replace negative values with zeros using NumPy's where function
  df[column_name] = np.where(negative_mask, 0, df[column_name])

  return df

# Example usage
data = {'A': [1, -2, 3, -4, 5], 'B': [6, 7, -8, 9, 10]}
df = pd.DataFrame(data)

df = replace_negative_values(df, 'A')
print(df)
```

Que_26. Access the 3rd element from the given NumPy array. arr = np.array([10, 20, 30, 40, 50])

```
arr[2]
```

Que_27. Retrieve the element at index (1, 2) from the 2D NumPy array.

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

```
arr_2d[0:1,:2]
```

Que_28. Using boolean indexing, extract elements greater than 5 from the given NumPy array. arr = np.array([3, 8, 2, 10, 5, 7])

```
 import numpy as np

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

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

# Extract elements using boolean indexing
result = arr[mask]

print(result)
```

Que_29. Perform basic slicing to extract elements from index 2 to 5 (inclusive) from the given NumPy array. arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

```
 import numpy as np

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

```
# Extract elements from index 2 to 5 (inclusive)
sliced_arr = arr[2:6]

print(sliced_arr)
```

Que_30. Slice the 2D NumPy array to extract the sub-array `[[2, 3], [5, 6]]` from the given array. arr_2d = np.array([[1, 2, 3], [4, 5, 6],[7, 8, 9]])

```
import numpy as np

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

# Extract the sub-array
sub_array = arr_2d[0:2,1:3]

print(sub_array)
```

Que_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(array, indices):
  """Extracts elements from a 2D NumPy array based on indices.

  Args:
    array: The 2D NumPy array.
    indices: A 2D NumPy array containing the row and column indices of the
elements to extract.

  Returns:
    A 1D NumPy array containing the extracted elements.
  """
```

```
  return array[indices[:, 0], indices[:, 1]]

# Example usage
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

indices = np.array([[0, 1],
                    [1, 2]])

extracted_elements = extract_elements(array, indices)
print(extracted_elements)
```

Que_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(arr, threshold):
  """Filters elements greater than a threshold from a 1D NumPy array.

  Args:
    arr: The 1D NumPy array.
    threshold: The threshold value.

  Returns:
    A new NumPy array containing the elements greater than the threshold.
  """

  mask = arr > threshold
  return arr[mask]

# Example usage
arr = np.array([3, 8, 2, 10, 5, 7])
threshold = 5
filtered_arr = filter_elements_greater_than(arr, threshold)
print(filtered_arr)
```

Que_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_3d(array, row_indices, col_indices, depth_indices):
  """Extracts elements from a 3D NumPy array based on indices.

  Args:
    array: The 3D NumPy array.
    row_indices: A 1D NumPy array containing the row indices.
    col_indices: A 1D NumPy array containing the column indices.
    depth_indices: A 1D NumPy array containing the depth indices.

  Returns:
    A 1D NumPy array containing the extracted elements.
  """

  return array[row_indices, col_indices, depth_indices]

# Example usage
array = np.array([[[1, 2, 3],
                   [4, 5, 6]],
                  [[7, 8, 9],
                   [10, 11, 12]]])

row_indices = np.array([0, 1])
col_indices = np.array([1, 2])
depth_indices = np.array([0, 1])

extracted_elements = extract_elements_3d(array, row_indices, col_indices,
depth_indices)
print(extracted_elements)
```

Que_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 extract_elements_multiple_conditions(arr, condition1, condition2):
  """Extracts elements from a 1D NumPy array based on multiple conditions.

  Args:
    arr: The 1D NumPy array.
    condition1: The first condition to check.
    condition2: The second condition to check.

  Returns:
    A new NumPy array containing the extracted elements.
  """

  mask = (condition1 & condition2)
  return arr[mask]

# Example usage
arr = np.array([3, 8, 2, 10, 5, 7])
condition1 = arr > 5
condition2 = arr % 2 == 0

extracted_elements = extract_elements_multiple_conditions(arr, condition1,
condition2)
print(extracted_elements)
```

Que_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_2d(array, row_indices, col_indices):
  """Extracts elements from a 2D NumPy array based on indices.
```

```
  Args:
    array: The 2D NumPy array.
    row_indices: A 1D NumPy array containing the row indices.
    col_indices: A 1D NumPy array containing the column indices.

  Returns:
    A 1D NumPy array containing the extracted elements.
  """

  return array[row_indices, col_indices]

# Example usage
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

row_indices = np.array([0, 1])
col_indices = np.array([1, 2])

extracted_elements = extract_elements_2d(array, row_indices, col_indices)
print(extracted_elements)
```

Que_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

# Create a 3x3 NumPy array
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

# Add 5 to each of the element in arr
result = arr + 5

print(result)
```

Que_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

# Create two NumPy arrays
arr1 = np.array([[1, 1, 3]])
arr2 = np.array([[4, 4, 6, 7],
                 [8, 8, 10, 11],
                 [12, 13, 14, 15]])

# Multiply each row of arr2 by the corresponding element in arr1 using
broadcasting
result = arr1 * arr2.reshape(4,3) #reshape arr2 to (4,3)

print(result)
```

Que_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

# Create NumPy arrays
arr1 = np.array([[1, 2, 3, 4]])
arr2 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9],
                 [10, 11, 12]])

# Add arr1 to each row of arr2 using broadcasting

result = arr2 + arr1.reshape(4, 1)  # Reshape arr1 to (4, 1)
print(result)
```

Que_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

# Create NumPy arrays
arr1 = np.array([[1],
                 [2],
                 [3]])
arr2 = np.array([[1, 2, 3]])

# Add the arrays using broadcasting
result = arr1 + arr2

print(result)
```

Que_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

# Create NumPy arrays
arr1 = np.array([[1, 2, 3],
                 [4, 5, 6]])
arr2 = np.array([[7, 8],
                 [9, 10]])

# Reshape arr2 to (2, 1) and add a new dimension
arr2_reshaped = np.expand_dims(arr2, axis=1)

# Perform element-wise multiplication
result = arr1 * arr2_reshaped

print(result)
```

Que_41. Calculate column-wise mean for the given array: arr = np.array([[1, 2, 3], [4, 5, 6]])

```
 import numpy as np

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

# Calculate column-wise mean
column_means = np.mean(arr, axis=0)

print(column_means)
```

Que_42. Find maximum value in each row of the given array: arr = np.array([[1, 2, 3], [4, 5, 6]])

```
 import numpy as np

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

# Find the maximum value in each row
row_max = np.max(arr, axis=1)

print(row_max)
```

Que_43. For the given array, find indices of maximum value in each column. arr = np.array([[1, 2, 3], [4, 5, 6]])

```
import numpy as np

arr = np.array([[1, 2, 3],
```

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

# Find the index of the maximum value in each column
max_indices = np.argmax(arr, axis=0)

print(max_indices)
```

Que_44. For the given array, apply custom function to calculate moving sum along rows. arr = np.array([[1, 2, 3], [4, 5, 6]])

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

#calculate the moving sum along rows
moving_sum = arr.cumsum(axis= 1)
print(moving_sum)
```

Que_45. In the given array, check if all elements in each column are even. arr = np.array([[2, 4, 6], [3, 5, 7]])

```
import numpy as np

arr = np.array([[2, 4, 6],
                [3, 5, 7]])

# Check if all elements in each column are even
are_all_even = np.all(arr % 2 == 0, axis=0)

print(are_all_even)
```

Que_46. Given a NumPy array arr, reshape it into a matrix of dimensions `m` rows and `n` columns. Return the reshaped matrix. original_array = np.array([1, 2, 3, 4, 5, 6])

```
def reshape_matrix(original_array, m, n):
    """
    Reshapes a NumPy array into a matrix of specified dimensions.

    Args:
        original_array: The original NumPy array.
        m: The desired number of rows in the reshaped matrix.
        n: The desired number of columns in the reshaped matrix.

    Returns:
        The reshaped matrix.

    Raises:
        ValueError: If the product of `m` and `n` is not equal to the length of
the original array.
    """

    if m * n != len(original_array):
        raise ValueError("The product of m and n must equal the length of the
original array.")

    return original_array.reshape(m, n)

# Example usage:
original_array = np.array([1, 2, 3, 4, 5, 6])
```

```
m = 2
n = 3
reshaped_matrix = reshape_matrix(original_array, m, n)
print(reshaped_matrix)
```

Que_47. Create a function that takes a matrix as input and returns the flattened array. input_matrix = np.array([[1, 2, 3], [4, 5, 6]])

```
import numpy as np

def flatten_matrix(input_matrix):
  """
  Flattens a matrix into a one-dimensional array.

  Args:
    input_matrix: The input matrix.

  Returns:
    The flattened array.
  """

  return input_matrix.flatten()

# Example usage:
input_matrix = np.array([[1, 2, 3], [4, 5, 6]])
flattened_array = flatten_matrix(input_matrix)
print(flattened_array)
```

Que_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=0):
  """
  Concatenates two given arrays along a specified axis.

  Args:
    array1: The first array.
    array2: The second array.
    axis: The axis along which to concatenate.

  Returns:
    The concatenated array.
  """

  return np.concatenate((array1, array2), axis=axis)

# Example usage:
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])

# Concatenate along the first axis (row-wise)
concatenated_array1 = concatenate_arrays(array1, array2, axis=0)
print(concatenated_array1)

# Concatenate along the second axis (column-wise)
concatenated_array2 = concatenate_arrays(array1, array2, axis=1)
print(concatenated_array2)
```

Que_49. Create a function that splits an array into multiple sub-arrays along a specified axis. original_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).

```
import numpy as np

def split_array(arr, axis=0, sections=None):
  """
  Splits a NumPy array into multiple sub-arrays along a specified axis.

  Args:
    arr: The input array.
```

```
    axis: The axis along which to split.
    sections: The number of sections to split the array into (optional).

  Returns:
    A list of split arrays.
  """

  if sections is None:
    sections = arr.shape[axis] // 2  # Default to splitting into two equal parts

  return np.split(arr, sections, axis=axis)

# Example usage:
original_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Split into two equal parts along the second axis (column-wise)
split_arrays = split_array(original_array)
print(split_arrays)

# Split into three sections along the first axis (row-wise)
split_arrays = split_array(original_array, axis=0, sections=3)
print(split_arrays)
```

Que_50. Write a function that inserts and then deletes elements from a given array at specified indices. original_array = np.array([1, 2, 3, 4, 5])[SEP] indices_to_insert = [2, 4][SEP] values_to_insert = [10, 11][SEP] indices_to_delete = [1, 3].

```
import numpy as np
def
insert_and_delete_ar(arr,indices_to_insert,values_to_insert,indices_to_delete):
  """
  Inserts and then deletes elements from a given array at specified indices.

  Args:
    arr: The input array.
    indices_to_insert: A list of indices where to insert the new elements.
    values_to_insert: A list of values to insert.
    indices_to_delete: A list of indices where to delete elements.
```

```
  Returns:
    result
  """
  array = np.insert(arr, indices_to_insert,values_to_insert)
  result = np.delete(array,indices_to_delete)
  return result
#usage
insert_and_delete_ar(original_array,indices_to_insert,values_to_insert,indices_to
_delete)
```

Que_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 array arr1 with random integers
arr1 = np.random.randint(1,200,(1,10))

# Create array arr2 with integers from 1 to 10
arr2 = np.random.randint(1,11,(1,10))
# Perform element-wise addition
result = arr1 + arr2

print("arr1:", arr1)
print("arr2:", arr2)
print("Result:", result)
```

Que_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

# Create array arr1 with sequential integers from 10 to 1
arr1 = np.arange(10, 0, -1)

# Create array arr2 with integers from 1 to 10
arr2 = np.arange(1, 11)

# Subtract arr2 from arr1 element-wise
result = arr1 - arr2

print("arr1:", arr1)
print("arr2:", arr2)
print("Result:", result)
```

Que_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

# Create array arr1 with random integers
arr1 = np.random.randint(10,50,(1,5))
# Create array arr2 with integers from 1 to 5
arr2 = np.arange(1,6)
# Perform element-wise multiplication
result = arr1 * arr2

print("arr1:", arr1)
print("arr2:", arr2)
print("Result:", result)
```

Que_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

# Create array arr1 with even integers from 2 to 10
arr1 = np.arange(2, 11, 2)

# Create array arr2 with integers from 1 to 5
arr2 = np.arange(1, 6)

# Perform element-wise division of arr1 by arr2
result = arr1 / arr2

print("arr1:", arr1)
print("arr2:", arr2)
print("Result:", result)
```

Que_55. 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

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

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

# Calculate the exponentiation of arr1 raised to the power of arr2 element-wise
result = arr1 ** arr2

print("arr1:", arr1)
print("arr2:", arr2)
print("Result:", result)
```

Que_56. Write a function that counts the occurrences of a specific substring within a NumPy array of strings. arr = np.array(['hello', 'world', 'hello', 'numpy', 'hello']).

```
 import numpy as np

def count_substring_occurrences(arr, substring):
  """Counts the occurrences of a specific substring within a NumPy array of strings.

  Args:
    arr: The input NumPy array of strings.
    substring: The substring to search for.

  Returns:
    The total number of occurrences of the substring in the array.
  """

  # Use np.char.count to count occurrences of the substring in each element
  counts = np.char.count(arr, substring)

  # Sum the counts to get the total number of occurrences
  total_count = np.sum(counts)

  return total_count

# Example usage:
arr = np.array(['hello', 'world', 'hello', 'numpy', 'hello'])
substring = 'hello'
total_count = count_substring_occurrences(arr, substring)
print(total_count)  # Output: 3
```

Que_57. Write a function that extracts uppercase characters from a NumPy array of strings. arr = np.array(['Hello', 'World', 'OpenAI', 'GPT']) arr = np.array(['Hello', 'World', 'OpenAI', 'GPT']).

```
def extract_upprcase(arr):
  for i in arr:
```

```
    if np.char.isupper(i) = True
    return i



  import numpy as np

def extract_uppercase(arr):
  """Extracts uppercase characters from a NumPy array of strings.

  Args:
    arr: The input NumPy array of strings.

  Returns:
    A NumPy array containing only the uppercase characters from the original
array.
  """
  # Use np.char.isalpha and np.char.isupper to filter only uppercase
characters

  upper = arr[np.char.isalpha(arr) & np.char.isupper(arr)]

  return upper
```

Que_58. Write a function that replaces occurrences of a substring in a NumPy array of strings with a new string. arr = np.array(['apple', 'banana', 'grape', 'pineapple']).

```
import numpy as np

def replace_substring(arr, old_substring, new_substring):
  """Replaces occurrences of a substring in a NumPy array of strings with a
new string.

  Args:
    arr: The input NumPy array of strings.
    old_substring: The substring to be replaced.
    new_substring: The new substring to replace the old one.

  Returns:
```

```
    A new NumPy array with the replaced substrings.
  """

  # Use np.char.replace to replace occurrences of the old substring with the
new one
  new_arr = np.char.replace(arr, old_substring, new_substring)

  return new_arr

# Example usage:
arr = np.array(['hello world', 'hello again', 'numpy is cool'])
old_substring = 'hello'
new_substring = 'hi'
new_arr = replace_substring(arr, old_substring, new_substring)
print(new_arr)
```

Que_59. Write a function that concatenates strings in a NumPy array element-wise. arr1 = np.array(['Hello', 'World']) arr2 = np.array(['Open', 'AI']).

```
import numpy as np

def concatenate_strings(arr1, arr2):
  """Concatenates two NumPy arrays of strings element-wise.

  Args:
    arr1: The first NumPy array of strings.
    arr2: The second NumPy array of strings.

  Returns:
    A new NumPy array with the concatenated strings.
  """

  # Use np.char.add to concatenate corresponding elements of arr1 and arr2
  result = np.char.add(arr1, arr2)

  return result

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

Que_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_longest_string_length(arr):
  """Finds the length of the longest string in a NumPy array.

  Args:
    arr: The input NumPy array of strings.

  Returns:
    The length of the longest string in the array.
  """
```

```
  # Use np.char.len to get the length of each string in the array
  lengths = np.char.len(arr)

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

  return max_length

# Example usage:
arr = np.array(['apple', 'banana', 'grape', 'pineapple'])
longest_length = find_longest_string_length(arr)
print(longest_length)  # Output: 9
```

Que_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

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

# Compute the mean, median, variance, and standard deviation
mean = np.mean(data)
median = np.median(data)
variance = np.var(data)
std_dev = np.std(data)

print("Mean:", mean)
print("Median:", median)
print("Variance:", variance)
print("Standard Deviation:", std_dev)
```

Que_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 numbers between 1 and 100
data = np.random.randint(1, 101, 50)

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

print("25th percentile:", percentile_25)
print("75th percentile:", percentile_75)
```

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

# Compute the correlation coefficient
correlation_coefficient = np.corrcoef(x, y)[0, 1]

print("Correlation coefficient:", correlation_coefficient)
```

Que_64. Create two matrices and perform matrix multiplication using NumPy's `dot` function.

```python
import numpy as np

# Create 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("Matrix 1:")
print(matrix1)
print("Matrix 2:")
print(matrix2)
print("Result:")
print(result)
```

Que_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.

```python
import numpy as np

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

# Calculate the 10th, 50th (median), and 90th percentiles
percentile_10 = np.percentile(data, 10)
percentile_50 = np.percentile(data, 50)  # Median
percentile_90 = np.percentile(data, 90)

# Calculate the first and third quartiles
quartile_1 = np.percentile(data, 25)
quartile_3 = np.percentile(data, 75)

print("10th percentile:", percentile_10)
print("50th percentile (median):", percentile_50)
print("90th percentile:", percentile_90)
print("First quartile:", quartile_1)
print("Third quartile:", quartile_3)
```

Que_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
arr = np.array([10, 20, 30, 40, 50, 30])

# Find the index of the first occurrence of the element 30
index = np.where(arr == 30)[0][0]

print("Index of the first occurrence of 30:", index)
```

Que_67. Generate a random NumPy array and sort it in ascending order.

```
import numpy as np

# Generate a random NumPy array of size 10
random_array = np.random.randint(0, 100, 10)

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

print("Original array:", random_array)
print("Sorted array:", sorted_array)
```

Que_68. Filter elements >20 in the given NumPy array. arr = np.array([12, 25, 6, 42, 8, 30]).

```
import numpy as np
```

```
arr = np.array([12, 25, 6, 42, 8, 30])

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

print(filtered_arr)
```

Que_69. Filter elements which are divisible by 3 from a given NumPy array. arr = np.array([1, 5, 8, 12, 15]).

```
import numpy as np

arr = np.array([1, 5, 8, 12, 15])

# Filter elements divisible by 3 using a boolean mask
filtered_arr = arr[arr % 3 == 0]

print(filtered_arr)
```

Que_70. Filter elements which are ≥ 20 and ≤ 40 from a given NumPy array. arr = np.array([10, 20, 30, 40, 50]).

```
import numpy as np

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

# Filter elements between 20 and 40 using a boolean mask
filtered_arr = arr[(arr >= 20) & (arr <= 40)]

print(filtered_arr)
```

Que_71. For the given NumPy array, check its byte order using the `dtype` attribute byteorder. arr = np.array([1, 2, 3]).

```
import numpy as np

arr = np.array([1, 2, 3])

byte_order = arr.dtype.byteorder

print("Byte order:", byte_order)
```

Que_72. For the given NumPy array, perform byte swapping in place using `byteswap()`. arr = np.array([1, 2, 3], dtype=np.int32).

```
import numpy as np

arr = np.array([1, 2, 3], dtype=np.int32)

# In-place byte swapping
arr.byteswap(inplace=True)

print(arr)
```

Que_73. For the given NumPy array, swap its byte order without modifying the original array using `newbyteorder()`. arr = np.array([1, 2, 3], dtype=np.int32).

```
import numpy as np

arr = np.array([1, 2, 3], dtype=np.int32)

# Create a new array with the swapped byte order
swapped_arr = arr.newbyteorder()

print("Original array:", arr)
print("Swapped array:", swapped_arr)
```

Que_74. For the given NumPy array and swap its byte order conditionally based on system endianness using `newbyteorder()`. arr = np.array([1, 2, 3], dtype=np.int32).

```
import numpy as np

def swap_byte_order_conditionally(arr):
  """Swaps the byte order of a NumPy array conditionally based on system
endianness.

  Args:
    arr: The input NumPy array.

  Returns:
    A new NumPy array with the swapped byte order if necessary.
  """

  # Determine the system's native byte order
  native_byte_order = np.dtype(arr.dtype).byteorder

  # Swap byte order if the array's byte order is different from the native byte
order
  if native_byte_order == '<':
    swapped_arr = arr.newbyteorder('>')
  elif native_byte_order == '>':
    swapped_arr = arr.newbyteorder('<')
  else:
    swapped_arr = arr

  return swapped_arr

# Example usage:
arr = np.array([1, 2, 3], dtype=np.int32)
swapped_arr = swap_byte_order_conditionally(arr)

print("Original array:", arr)
print("Swapped array:", swapped_arr)
```

Que_75. For the given NumPy array, check if byte swapping is necessary for the current system using `dtype` attribute `byteorder`. arr = np.array([1, 2, 3], dtype=np.int32).

```
import numpy as np

def is_byte_swap_necessary(arr):
  """Checks if byte swapping is necessary for the given NumPy array based on system endianness.

  Args:
    arr: The input NumPy array.

  Returns:
    True if byte swapping is necessary, False otherwise.
  """

  # Determine the system's native byte order
  native_byte_order = np.dtype(arr.dtype).byteorder

  # Check if the array's byte order is different from the native byte order
  if arr.dtype.byteorder != native_byte_order:
    return True
  else:
    return False

# Example usage:
arr = np.array([1, 2, 3], dtype=np.int32)
is_necessary = is_byte_swap_necessary(arr)

if is_necessary:
  print("Byte swapping is necessary.")
else:
  print("Byte swapping is not necessary.")
```

Que_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

# Create a NumPy array with values from 1 to 10
arr1 = np.arange(1, 11)

# Make a copy of arr1 using np.copy()
copy_arr = np.copy(arr1)

# Modify an element in copy_arr
copy_arr[5] = 100

# Print the original array and the modified copy
print("Original array:", arr1)
print("Modified copy:", copy_arr)
```

Que_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

# Create a 2D NumPy array of shape (3, 3) with random integers
matrix = np.random.randint(1, 10, (3, 3))

# Extract a slice from the matrix
view_slice = matrix[1:, 1:]

# Modify an element in the slice
view_slice[0, 0] = 999

# Print the original matrix and the slice
print("Original matrix:")
print(matrix)
```

```
print("Slice:")
print(view_slice)
```

Que_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

# Create a NumPy array of shape (4, 3) with sequential integers from 1 to 12
array_a = np.arange(1, 13).reshape(4, 3)

# Extract a slice from array_a
view_b = array_a[1:, 1:]

# Broadcast the addition of 5 to view_b
view_b += 5

# Check if the original array_a has changed
print("Original array_a:")
print(array_a)

# Output:
# Original array_a:
# [[ 1  2  3]
#  [ 6  7  8]
#  [ 9 10 11]
#  [12 13 14]]

As you can see, the original array_a has been modified after adding 5 to the
view_b slice. This is because slices in NumPy are views into the original array,
meaning they share the same underlying data. Any changes made to the slice will
also be reflected in the original array.

If you want to avoid modifying the original array, you can create a copy of the
slice using np.copy()


copy_b = np.copy(array_a[1:, 1:])
```

```
copy_b += 5

print("Original array_a:")
print(array_a)
```

Que_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

# Create a NumPy array of shape (2, 4) with values from 1 to 8
orig_array = np.arange(1, 9).reshape(2, 4)

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

# Modify an element in the reshaped view
reshaped_view[2, 1] = 999

# Check if the original array has changed
print("Original array:")
print(orig_array)

# Output:
# Original array:
# [[1 2 3 4]
#  [5 6 7 8]]

As you can see, modifying an element in the reshaped_view also modifies the
corresponding element
in the original orig_array. This is because reshaping a NumPy array does not
create a new copy of
the data, but rather creates a new view of the same underlying data. So, any
changes made to the
reshaped view will be reflected in the original array.
```

Que_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

# Create a NumPy array of shape (3, 4) with random integers
data = np.random.randint(1, 10, (3, 4))

# Extract elements greater than 5 as a copy
data_copy = data[data > 5].copy()

# Modify an element in data_copy
data_copy[1] = 999

# Check if the original data has changed
print("Original data:")
print(data)

# Output:
# Original data:
# [[1 2 8 3]
#  [4 5 9 7]
#  [6 1 2 3]]

Because we used copy(), any modifications made to data_copy will not affect the
original data.
This is essential when you want to work with a subset of an array without
unintentionally modifying
the original data.
```

Que_81. Create two matrices A and B of identical shape containing integers and perform addition and subtraction operations between them.

```
import numpy as np

# Create two matrices of the same shape
```

```
A = np.random.randint(1, 10, (3, 3))
B = np.random.randint(1, 10, (3, 3))

# Perform matrix addition
result_addition = A + B

# Perform matrix subtraction
result_subtraction = A - B

print("Matrix A:")
print(A)
print("Matrix B:")
print(B)
print("Addition:")
print(result_addition)
print("Subtraction:")
print(result_subtraction)
```

Que_82. Generate two matrices `C` (3x2) and `D` (2x4) and perform matrix multiplication.

```
import numpy as np

# Create matrix C of shape (3, 2)
C = np.random.randint(1, 10, (3, 2))

# Create matrix D of shape (2, 4)
D = np.random.randint(1, 10, (2, 4))

# Perform matrix multiplication
result = np.dot(C, D)

print("Matrix C:")
print(C)
print("Matrix D:")
print(D)
print("Result:")
print(result)
```

Que_83. Create a matrix `E` and find its transpose.

```
import numpy as np

# Create a matrix E
E = np.random.randint(1, 10, (3, 4))

# Find the transpose of E
transpose_E = E.T

print("Matrix E:")
print(E)
print("Transpose of E:")
print(transpose_E)
```

Que_84. Generate a square matrix `F` and compute its determinant.

```

import numpy as np

# Create a square matrix F
F = np.random.randint(1, 10, (4, 4))

# Compute the determinant of F
determinant = np.linalg.det(F)

print("Matrix F:")
print(F)
print("Determinant of F:", determinant)
```

Que_85. Create a square matrix `G` and find its inverse.

```
import numpy as np
```

```python
# Create a square matrix G
G = np.random.randint(1, 10, (4, 4))

# Find the inverse of G
inverse_G = np.linalg.inv(G)

print("Matrix G:")
print(G)
print("Inverse of G:")
print(inverse_G)
```