*# Create a NumPy array 'arr' of integers from 0 to 5 and print its data type:*

**import** numpy **as** np

arr **=** np**.**arange(6)

print("Array:", arr)

print("Data type:", arr**.**dtype)

Array: [0 1 2 3 4 5]

Data type: int32

In [83]:

*# Given a NumPy array 'arr', check if its data type is float64:*

arr **=** np**.**array([1.5, 2.6, 3.7])

is\_float64 **=** arr**.**dtype **==** np**.**float64

print("Is the array float64?", is\_float64)

Is the array float64? True

In [84]:

*# Create a NumPy array 'arr' with a data type of complex128 containing three complex numbers:*

arr **=** np**.**array([1**+**2j, 3**+**4j, 5**+**6j], dtype**=**np**.**complex128)

print("Array:", arr)

print("Data type:", arr**.**dtype)

Array: [1.+2.j 3.+4.j 5.+6.j]

Data type: complex128

In [85]:

*# Convert an existing NumPy array 'arr' of integers to float32 data type:*

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

arr\_float32 **=** arr**.**astype(np**.**float32)

print("Original array:", arr)

print("Converted array:", arr\_float32)

print("Data type of converted array:", arr\_float32**.**dtype)

Original array: [1 2 3 4 5]

Converted array: [1. 2. 3. 4. 5.]

Data type of converted array: float32

In [86]:

*# Given a NumPy array 'arr' with float64 data type, convert it to float32 to reduce decimal precision:*

arr **=** np**.**array([1.1, 2.2, 3.3], dtype**=**np**.**float64)

arr\_float32 **=** arr**.**astype(np**.**float32)

print("Original array:", arr)

print("Converted array:", arr\_float32)

print("Data type of converted array:", arr\_float32**.**dtype)

Original array: [1.1 2.2 3.3]

Converted array: [1.1 2.2 3.3]

Data type of converted array: float32

In [87]:

*# Write a function array\_attributes that takes a NumPy array as input and returns its shape, size, and data type:*

**def** array\_attributes(arr):

**return** arr**.**shape, arr**.**size, arr**.**dtype

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

print(array\_attributes(arr))

((5,), 5, dtype('int32'))

In [88]:

*# Create a function array\_dimension that takes a NumPy array as input and returns its dimensionality:*

**def** array\_dimension(arr):

**return** arr**.**ndim

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

print(array\_dimension(arr))

2

In [89]:

*# Design a function item\_size\_info that takes a NumPy array as input and returns the item size and the total size in bytes:*

**def** item\_size\_info(arr):

**return** arr**.**itemsize, arr**.**nbytes

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

print(item\_size\_info(arr))

(4, 20)

In [90]:

*# Create a function array\_strides that takes a NumPy array as input and returns the strides of the array:*

**def** array\_strides(arr):

**return** arr**.**strides

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

print(array\_strides(arr))

(12, 4)

In [91]:

*# Design a function shape\_stride\_relationship that takes a NumPy array as input and returns the shape and strides of the array:*

**def** shape\_stride\_relationship(arr):

**return** arr**.**shape, arr**.**strides

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

print(shape\_stride\_relationship(arr))

((2, 3), (12, 4))

In [92]:

*# Create a function create\_zeros\_array that takes an integer n as input and returns a NumPy array of zeros with n elements:*

**def** create\_zeros\_array(n):

**return** np**.**zeros(n)

print(create\_zeros\_array(5))

[0. 0. 0. 0. 0.]

In [93]:

*# 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:*

**def** create\_ones\_matrix(rows, cols):

**return** np**.**ones((rows, cols))

print(create\_ones\_matrix(3, 4))

[[1. 1. 1. 1.]

[1. 1. 1. 1.]

[1. 1. 1. 1.]]

In [94]:

*# 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:*

**def** generate\_range\_array(start, stop, step):

**return** np**.**arange(start, stop, step)

print(generate\_range\_array(1, 10, 2))

[1 3 5 7 9]

In [95]:

*# 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):*

**def** generate\_linear\_space(start, stop, num):

**return** np**.**linspace(start, stop, num)

print(generate\_linear\_space(0.0, 1.0, 5))

[0. 0.25 0.5 0.75 1. ]

In [96]:

*# 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:*

**def** create\_identity\_matrix(n):

**return** np**.**eye(n)

print(create\_identity\_matrix(4))

[[1. 0. 0. 0.]

[0. 1. 0. 0.]

[0. 0. 1. 0.]

[0. 0. 0. 1.]]

In [97]:

*# Write a function that takes a Python list and converts it into a NumPy array:*

**def** list\_to\_array(lst):

**return** np**.**array(lst)

print(list\_to\_array([1, 2, 3, 4, 5]))

[1 2 3 4 5]

In [98]:

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

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

arr\_view **=** arr**.**view()

print("Original array:", arr)

print("View of the array:", arr\_view)

Original array: [1 2 3 4 5]

View of the array: [1 2 3 4 5]

In [99]:

*# Write a function that takes two NumPy arrays and concatenates them along a specified axis:*

**def** concatenate\_arrays(arr1, arr2, axis):

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

arr1 **=** np**.**array([[1, 2], [3, 4]])

arr2 **=** np**.**array([[5, 6], [7, 8]])

print(concatenate\_arrays(arr1, arr2, axis**=**0))

print(concatenate\_arrays(arr1, arr2, axis**=**1))

[[1 2]

[3 4]

[5 6]

[7 8]]

[[1 2 5 6]

[3 4 7 8]]

In [100]:

*# Create two NumPy arrays with different shapes and concatenate them horizontally using numpy.concatenate:*

arr1 **=** np**.**array([[1, 2], [3, 4]])

arr2 **=** np**.**array([[5, 6]])

result **=** np**.**concatenate((arr1, arr2**.**T), axis**=**1)

print(result)

[[1 2 5]

[3 4 6]]

In [101]:

*# Write a function that vertically stacks multiple NumPy arrays given as a list:*

**def** vertical\_stack(arr\_list):

**return** np**.**vstack(arr\_list)

arr1 **=** np**.**array([1, 2, 3])

arr2 **=** np**.**array([4, 5, 6])

arr3 **=** np**.**array([7, 8, 9])

print(vertical\_stack([arr1, arr2, arr3]))

[[1 2 3]

[4 5 6]

[7 8 9]]

In [102]:

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

**def** create\_range\_array(start, end, step):

**return** np**.**arange(start, end **+** 1, step)

print(create\_range\_array(1, 10, 2))

[1 3 5 7 9]

In [103]:

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

**def** generate\_equal\_spaced\_array():

**return** np**.**linspace(0, 1, 10)

print(generate\_equal\_spaced\_array())

[0. 0.11111111 0.22222222 0.33333333 0.44444444 0.55555556

0.66666667 0.77777778 0.88888889 1. ]

In [104]:

*# Write a Python function using NumPy to create an array of 5 logarithmically spaced values between 1 and 1000 (inclusive):*

**def** generate\_log\_spaced\_array():

**return** np**.**logspace(0, 3, 5)

print(generate\_log\_spaced\_array())

[ 1. 5.62341325 31.6227766 177.827941 1000. ]

In [105]:

*# 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** pandas **as** pd

arr **=** np**.**random**.**randint(1, 101, size**=**(5, 3))

df **=** pd**.**DataFrame(arr, columns**=**['A', 'B', 'C'])

print(df)

A B C

0 76 60 45

1 44 20 5

2 68 31 23

3 11 7 83

4 27 39 30

In [106]:

*# 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:*

**def** replace\_negative\_with\_zero(df, column\_name):

df[column\_name] **=** np**.**where(df[column\_name] **<** 0, 0, df[column\_name])

**return** df

data **=** {'A': [1, **-**2, 3], 'B': [**-**4, 5, **-**6]}

df **=** pd**.**DataFrame(data)

print(replace\_negative\_with\_zero(df, 'A'))

A B

0 1 -4

1 0 5

2 3 -6

In [107]:

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

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

print("3rd element:", arr[2])

3rd element: 30

In [108]:

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

print("Element at index (1, 2):", arr\_2d[1, 2])

Element at index (1, 2): 6

In [109]:

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

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

print("Elements greater than 5:", arr[arr **>** 5])

Elements greater than 5: [ 8 10 7]

In [110]:

*# 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]):*

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

print("Elements from index 2 to 5:", arr[2:6])

Elements from index 2 to 5: [3 4 5 6]

In [111]:

*# 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]]):*

arr\_2d **=** np**.**array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

sub\_array **=** arr\_2d[0:2, 1:3]

print("Sub-array [[2, 3], [5, 6]]:", sub\_array)

Sub-array [[2, 3], [5, 6]]: [[2 3]

[5 6]]

In [112]:

*# Extract elements in a specific order from a 2D array based on indices*

**import** numpy **as** np

**def** indices(arr, row\_indices, col\_indices):

**return** arr[row\_indices, col\_indices]

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

row\_indices **=** np**.**array([0, 1, 2])

col\_indices **=** np**.**array([2, 0, 1])

print("Extracted elements:", indices(arr, row\_indices, col\_indices))

Extracted elements: [3 4 8]

In [113]:

*# Filter elements greater than a threshold from a 1D array*

**def** filter\_greater\_than(arr, threshold):

**return** arr[arr **>** threshold]

arr **=** np**.**array([1, 3, 5, 7, 9])

threshold **=** 5

print("Filtered elements:", filter\_greater\_than(arr, threshold))

Filtered elements: [7 9]

In [114]:

*# Extract specific elements from a 3D array using indices*

**def** extract\_from\_3d\_array(arr, idx1, idx2, idx3):

**return** arr[idx1, idx2, idx3]

arr **=** np**.**random**.**random((3, 4, 5))

idx1 **=** np**.**array([0, 1, 2])

idx2 **=** np**.**array([1, 2, 3])

idx3 **=** np**.**array([2, 3, 4])

print("Extracted elements:", extract\_from\_3d\_array(arr, idx1, idx2, idx3))

Extracted elements: [0.28159814 0.56748081 0.4578364 ]

In [115]:

*# Return elements where both conditions are satisfied*

**def** filter\_with\_conditions(arr, cond1, cond2):

**return** arr[(arr **>** cond1) **&** (arr **<** cond2)]

arr **=** np**.**array([1, 3, 5, 7, 9])

cond1 **=** 2

cond2 **=** 8

print("Filtered elements:", filter\_with\_conditions(arr, cond1, cond2))

Filtered elements: [3 5 7]

In [116]:

*# Extract elements from a 2D array using row and column indices*

**def** extract\_elements\_by\_rows\_cols(arr, row\_indices, col\_indices):

**return** arr[row\_indices, col\_indices]

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

row\_indices **=** np**.**array([0, 1, 2])

col\_indices **=** np**.**array([2, 0, 1])

print("Extracted elements:", extract\_elements\_by\_rows\_cols(arr, row\_indices, col\_indices))

Extracted elements: [3 4 8]

In [117]:

*# Add a scalar value of 5 to each element using broadcasting*

**def** add\_scalar(arr, scalar):

**return** arr **+** scalar

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

scalar **=** 5

print("Array after adding scalar:", add\_scalar(arr, scalar))

Array after adding scalar: [[ 6 7 8]

[ 9 10 11]

[12 13 14]]

In [118]:

*# 37. Multiply each row of arr2 by the corresponding element in arr1*

**def** multiply\_rows\_by\_elements(arr1, arr2):

**return** arr1 **\*** arr2

arr1 **=** np**.**array([[1, 2, 3]])

arr2 **=** np**.**array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

print("Multiplied array:", multiply\_rows\_by\_elements(arr1**.**T, arr2))

Multiplied array: [[ 1 2 3 4]

[10 12 14 16]

[27 30 33 36]]

In [119]:

*# Add arr1 to each row of arr2*

**def** add\_arr1\_to\_each\_row(arr1, arr2):

**return** arr1 **+** arr2

arr1 **=** np**.**array([[1, 2, 3, 4]])

arr2 **=** np**.**array([[4, 3, 2], [1, 0, **-**1], [**-**2, **-**3, **-**4], [**-**5, **-**6, **-**7]])

print("Array after addition:", add\_arr1\_to\_each\_row(arr1, arr2**.**T)**.**T)

Array after addition: [[ 5 4 3]

[ 3 2 1]

[ 1 0 -1]

[-1 -2 -3]]

In [120]:

*# . Add two arrays using broadcasting*

**def** add\_with\_broadcasting(arr1, arr2):

**return** arr1 **+** arr2

arr1 **=** np**.**array([[1], [2], [3]])

arr2 **=** np**.**array([[4, 5, 6]])

print("Broadcasted addition:", add\_with\_broadcasting(arr1, arr2))

Broadcasted addition: [[5 6 7]

[6 7 8]

[7 8 9]]

In [121]:

*# Handle shape incompatibility during broadcasting*

**def** multiply\_with\_broadcasting(arr1, arr2):

**try**:

**return** arr1 **\*** arr2

**except** ValueError **as** e:

**return** str(e)

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

arr2 **=** np**.**array([[7, 8], [9, 10]])

print("Broadcasted multiplication:", multiply\_with\_broadcasting(arr1, arr2))

Broadcasted multiplication: operands could not be broadcast together with shapes (2,3) (2,2)

In [122]:

*# Calculate column-wise mean*

**def** column\_wise\_mean(arr):

**return** np**.**mean(arr, axis**=**0)

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

print("Column-wise mean:", column\_wise\_mean(arr))

Column-wise mean: [2.5 3.5 4.5]

In [123]:

*# Find maximum value in each row*

**def** max\_in\_rows(arr):

**return** np**.**max(arr, axis**=**1)

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

print("Max in each row:", max\_in\_rows(arr))

Max in each row: [3 6]

In [124]:

*# Find indices of maximum value in each column*

**def** indices\_max\_in\_columns(arr):

**return** np**.**argmax(arr, axis**=**0)

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

print("Indices of max in each column:", indices\_max\_in\_columns(arr))

Indices of max in each column: [1 1 1]

In [125]:

*# . Calculate moving sum along rows*

**def** moving\_sum(arr, window\_size):

result **=** np**.**apply\_along\_axis(**lambda** x: np**.**convolve(x, np**.**ones(window\_size, dtype**=**int), 'valid'), axis**=**1, arr**=**arr)

**return** result

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

print("Moving sum along rows:", moving\_sum(arr, 2))

Moving sum along rows: [[ 3 5]

[ 9 11]]

In [126]:

*# Check if all elements in each column are even*

**def** check\_even\_columns(arr):

**return** np**.**all(arr **%** 2 **==** 0, axis**=**0)

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

print("All elements even in each column:", check\_even\_columns(arr))

All elements even in each column: [False False False]

In [127]:

*# Reshape array into a matrix*

**def** reshape\_array(arr, m, n):

**return** arr**.**reshape((m, n))

original\_array **=** np**.**array([1, 2, 3, 4, 5, 6])

print("Reshaped array:", reshape\_array(original\_array, 2, 3))

Reshaped array: [[1 2 3]

[4 5 6]]

In [128]:

*# Return flattened array*

**def** flatten\_matrix(matrix):

**return** matrix**.**flatten()

input\_matrix **=** np**.**array([[1, 2, 3], [4, 5, 6]])

print("Flattened array:", flatten\_matrix(input\_matrix))

Flattened array: [1 2 3 4 5 6]

In [129]:

*# Concatenate two arrays along a specified axis*

**def** concatenate\_arrays(array1, array2, axis**=**0):

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

array1 **=** np**.**array([[1, 2], [3, 4]])

array2 **=** np**.**array([[5, 6], [7, 8]])

print("Concatenated arrays along axis 0:", concatenate\_arrays(array1, array2, axis**=**0))

print("Concatenated arrays along axis 1:", concatenate\_arrays(array1, array2, axis**=**1))

Concatenated arrays along axis 0: [[1 2]

[3 4]

[5 6]

[7 8]]

Concatenated arrays along axis 1: [[1 2 5 6]

[3 4 7 8]]

In [130]:

*# Split an array into multiple sub-arrays*

**def** split\_array(original\_array, num\_splits, axis**=**0):

**return** np**.**array\_split(original\_array, num\_splits, axis**=**axis)

original\_array **=** np**.**array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("Split arrays along axis 0:", split\_array(original\_array, 3, axis**=**0))

print("Split arrays along axis 1:", split\_array(original\_array, 3, axis**=**1))

Split arrays along axis 0: [array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]])]

Split arrays along axis 1: [array([[1],

[4],

[7]]), array([[2],

[5],

[8]]), array([[3],

[6],

[9]])]

In [131]:

*# Insert and then delete elements from an array*

**def** insert\_and\_delete\_elements(original\_array, indices\_to\_insert, values\_to\_insert, indices\_to\_delete):

new\_array **=** np**.**insert(original\_array, indices\_to\_insert, values\_to\_insert)

new\_array **=** np**.**delete(new\_array, indices\_to\_delete)

**return** new\_array

original\_array **=** np**.**array([1, 2, 3, 4, 5])

indices\_to\_insert **=** [2, 4]

values\_to\_insert **=** [10, 11]

indices\_to\_delete **=** [1, 3]

print("Array after insertion and deletion:", insert\_and\_delete\_elements(original\_array, indices\_to\_insert, values\_to\_insert, indices\_to\_delete))

Array after insertion and deletion: [ 1 10 4 11 5]

In [132]:

*# Perform element-wise addition between two arrays*

**def** element\_wise\_addition(arr1, arr2):

**return** arr1 **+** arr2

arr1 **=** np**.**random**.**randint(1, 10, size**=**10)

arr2 **=** np**.**arange(1, 11)

print("Element-wise addition", element\_wise\_addition(arr1, arr2))

Element-wise addition [ 6 8 12 13 9 8 15 11 17 13]

In [133]:

*# Perform element-wise division of two arrays*

**def** element\_wise\_division(arr1, arr2):

**return** arr1 **/** arr2

arr1 **=** np**.**arange(2, 12, 2)

arr2 **=** np**.**arange(1, 6)

print("Element-wise division:", element\_wise\_division(arr1, arr2))

Element-wise division: [2. 2. 2. 2. 2.]

In [134]:

*# Calculate element-wise exponentiation*

**def** element\_wise\_exponentiation(arr1, arr2):

**return** np**.**power(arr1, arr2)

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

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

print("Element-wise exponentiation:", element\_wise\_exponentiation(arr1, arr2))

Element-wise exponentiation: [ 1 16 27 16 5]

In [135]:

*# Count occurrences of a specific substring*

**def** count\_substring\_occurrences(arr, substring):

**return** np**.**char**.**count(arr, substring)

arr **=** np**.**array(['hello', 'world', 'hello', 'numpy', 'hello'])

substring **=** 'hello'

print("Occurrences of substring:", count\_substring\_occurrences(arr, substring))

Occurrences of substring: [1 0 1 0 1]

In [136]:

*# Extract uppercase characters*

**def** extract\_uppercase\_chars(arr):

**return** np**.**array([''**.**join([char **for** char **in** string **if** char**.**isupper()]) **for** string **in** arr])

arr **=** np**.**array(['Hello', 'World', 'OpenAI', 'GPT'])

print("Uppercase characters:", extract\_uppercase\_chars(arr))

Uppercase characters: ['H' 'W' 'OAI' 'GPT']

In [137]:

*# Replace occurrences of a substring*

**def** replace\_substring(arr, old\_substring, new\_substring):

**return** np**.**char**.**replace(arr, old\_substring, new\_substring)

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

old\_substring **=** 'apple'

new\_substring **=** 'orange'

print("Replaced substrings:", replace\_substring(arr, old\_substring, new\_substring))

Replaced substrings: ['orange' 'banana' 'grape' 'pineorange']

In [138]:

*# Concatenate strings element-wise*

**def** concatenate\_strings(arr1, arr2):

**return** np**.**char**.**add(arr1, arr2)

arr1 **=** np**.**array(['Hello', 'World'])

arr2 **=** np**.**array(['Open', 'AI'])

print("Concatenated strings:", concatenate\_strings(arr1, arr2))

Concatenated strings: ['HelloOpen' 'WorldAI']

In [139]:

*# Find the length of the longest string*

**def** length\_of\_longest\_string(arr):

**return** np**.**max(np**.**char**.**str\_len(arr))

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

print("Length of the longest string:", length\_of\_longest\_string(arr))

Length of the longest string: 9

In [140]:

*# Create a dataset of 100 random integers between 1 and 1000. Compute the mean, median, variance, and standard deviation.*

**import** numpy **as** np

data\_61 **=** np**.**random**.**randint(1, 1001, size**=**100)

mean\_61 **=** np**.**mean(data\_61)

median\_61 **=** np**.**median(data\_61)

variance\_61 **=** np**.**var(data\_61)

std\_deviation\_61 **=** np**.**std(data\_61)

mean\_61, median\_61, variance\_61, std\_deviation\_61

Out[140]:

(549.76, 560.0, 84921.84239999998, 291.413524737614)

In [141]:

*# Generate an array of 50 random numbers between 1 and 100. Find the 25th and 75th percentiles.*

*# Task 62*

data\_62 **=** np**.**random**.**uniform(1, 100, size**=**50)

percentile\_25\_62 **=** np**.**percentile(data\_62, 25)

percentile\_75\_62 **=** np**.**percentile(data\_62, 75)

percentile\_25\_62, percentile\_75\_62

Out[141]:

(27.97305647216291, 75.8370791398221)

In [142]:

*# Create two arrays representing two sets of variables. Compute the correlation coefficient.*

*# Task 63*

array1\_63 **=** np**.**random**.**rand(100)

array2\_63 **=** np**.**random**.**rand(100)

correlation\_coefficient\_63 **=** np**.**corrcoef(array1\_63, array2\_63)

correlation\_coefficient\_63

Out[142]:

array([[ 1. , -0.12688944],

[-0.12688944, 1. ]])

In [ ]:

*# Create two matrices and perform matrix multiplication.*

*# Task 64*

matrix1\_64 **=** np**.**random**.**randint(1, 10, size**=**(3, 3))

matrix2\_64 **=** np**.**random**.**randint(1, 10, size**=**(3, 3))

matrix\_multiplication\_64 **=** np**.**dot(matrix1\_64, matrix2\_64)

matrix\_multiplication\_64

Out[ ]:

array([[100, 122, 85],

[136, 155, 105],

[126, 129, 83]])

In [ ]:

*# Create an array of 50 integers between 10 and 1000. Calculate percentiles and quartiles.*

*# Task 65*

data\_65 **=** np**.**random**.**randint(10, 1001, size**=**50)

percentile\_10\_65 **=** np**.**percentile(data\_65, 10)

median\_65 **=** np**.**percentile(data\_65, 50)

percentile\_90\_65 **=** np**.**percentile(data\_65, 90)

first\_quartile\_65 **=** np**.**percentile(data\_65, 25)

third\_quartile\_65 **=** np**.**percentile(data\_65, 75)

percentile\_10\_65, median\_65, percentile\_90\_65, first\_quartile\_65, third\_quartile\_65

In [ ]:

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

*# Task 66*

array\_66 **=** np**.**array([5, 10, 15, 20, 25])

index\_66 **=** np**.**where(array\_66 **==** 15)[0][0]

index\_66

In [ ]:

*# generate a random NumPy array and sort it in ascending order.*

*# Task 67*

array\_67 **=** np**.**random**.**rand(10)

sorted\_array\_67 **=** np**.**sort(array\_67)

sorted\_array\_67

In [ ]:

*# Filter elements > 20 in the given NumPy array.*

*# Task 68*

arr\_68 **=** np**.**array([12, 25, 6, 42, 8, 30])

filtered\_arr\_68 **=** arr\_68[arr\_68 **>** 20]

filtered\_arr\_68

In [ ]:

*# Filter elements which are divisible by 3*

*# Task 69*

arr\_69 **=** np**.**array([1, 5, 8, 12, 15])

filtered\_arr\_69 **=** arr\_69[arr\_69 **%** 3 **==** 0]

filtered\_arr\_69

In [ ]:

*# Filter elements which are ≥ 20 and ≤ 40*

*# Task 70*

arr\_70 **=** np**.**array([10, 20, 30, 40, 50])

filtered\_arr\_70 **=** arr\_70[(arr\_70 **>=** 20) **&** (arr\_70 **<=** 40)]

filtered\_arr\_70

In [ ]:

*# Check the byte order of a given NumPy array.*

*# Task 71*

arr\_71 **=** np**.**array([1, 2, 3])

byte\_order\_71 **=** arr\_71**.**dtype**.**byteorder

byte\_order\_71

In [ ]:

*# Perform byte swapping in place.*

*# Task 72*

arr\_72 **=** np**.**array([1, 2, 3], dtype**=**np**.**int32)

arr\_72**.**byteswap(**True**)

arr\_72

In [ ]:

*# Swap byte order without modifying the original array.*

*# Task 73*

arr\_73 **=** np**.**array([1, 2, 3], dtype**=**np**.**int32)

swapped\_arr\_73 **=** arr\_73**.**newbyteorder()

swapped\_arr\_73

In [ ]:

*# Swap byte order conditionally based on system endianness.*

*# Task 74*

arr\_74 **=** np**.**array([1, 2, 3], dtype**=**np**.**int32)

swapped\_arr\_74 **=** arr\_74**.**newbyteorder('=')

swapped\_arr\_74

In [ ]:

*# Check if byte swapping is necessary for the current system.*

*# Task 75*

arr\_75 **=** np**.**array([1, 2, 3], dtype**=**np**.**int32)

byte\_swap\_needed\_75 **=** arr\_75**.**dtype**.**byteorder **not** **in** ('=', '|')

byte\_swap\_needed\_75

In [ ]:

*# Create a copy of arr1 and modify an element in copy\_arr.*

*# Task 76*

arr1\_76 **=** np**.**arange(1, 11)

copy\_arr\_76 **=** arr1\_76**.**copy()

copy\_arr\_76[0] **=** 100

arr1\_76, copy\_arr\_76

In [ ]:

*# Extract a slice from the matrix and modify an element.*

*# Task 77*

matrix\_77 **=** np**.**random**.**randint(1, 10, size**=**(3, 3))

view\_slice\_77 **=** matrix\_77[:2, :2]

view\_slice\_77[0, 0] **=** 100

matrix\_77, view\_slice\_77

In [ ]:

*# Extract a slice and broadcast the addition of 5.*

*# Task 78*

array\_a\_78 **=** np**.**arange(1, 13)**.**reshape(4, 3)

view\_b\_78 **=** array\_a\_78[:, 1:2]

view\_b\_78 **+=** 5

array\_a\_78, view\_b\_78

In [ ]:

*# Create a reshaped view and modify an element.*

*# Task 79*

orig\_array\_79 **=** np**.**arange(1, 9)**.**reshape(2, 4)

reshaped\_view\_79 **=** orig\_array\_79**.**reshape(4, 2)

reshaped\_view\_79[0, 0] **=** 100

orig\_array\_79, reshaped\_view\_79

In [ ]:

*# Extract a copy of elements greater than 5 and modify an element.*

*# Task 80*

data\_80 **=** np**.**random**.**randint(1, 10, size**=**(3, 4))

data\_copy\_80 **=** data\_80[data\_80 **>** 5]**.**copy()

data\_copy\_80[0] **=** 100

data\_80, data\_copy\_80

In [ ]:

*# Perform addition and subtraction operations between two matrices.*

*# Task 81*

A\_81 **=** np**.**random**.**randint(1, 10, size**=**(3, 3))

B\_81 **=** np**.**random**.**randint(1, 10, size**=**(3, 3))

addition\_81 **=** A\_81 **+** B\_81

subtraction\_81 **=** A\_81 **-** B\_81

addition\_81, subtraction\_81

In [ ]:

*# Perform matrix multiplication between two matrices.*

*# Task 82*

C\_82 **=** np**.**random**.**randint(1, 10, size**=**(3, 2))

D\_82 **=** np**.**random**.**randint(1, 10, size**=**(2, 4))

matrix\_multiplication\_82 **=** np**.**dot(C\_82, D\_82)

matrix\_multiplication\_82

In [ ]:

*# Find the transpose of a matrix.*

*# Task 83*

E\_83 **=** np**.**random**.**randint(1, 10, size**=**(3, 3))

transpose\_E\_83 **=** E\_83**.**T

transpose\_E\_83

In [ ]:

*# Compute the determinant of a square matrix.*

*# Task 84*

F\_84 **=** np**.**random**.**randint(1, 10, size**=**(3, 3))

determinant\_F\_84 **=** np**.**linalg**.**det(F\_84)

determinant\_F\_84

In [ ]:

*# Find the inverse of a square matrix.*

*# Task 85*

G\_85 **=** np**.**random**.**randint(1, 10, size**=**(3, 3))

inverse\_G\_85 **=** np**.**linalg**.**inv(G\_85)

inverse\_G\_85