 4.adv\_operations

[11:01 AM] Poornima (Guest)

'''

Broadcasting

Broadcasting Rules:

If the arrays do not have the same rank, prepend the shape of the smaller array with 1s.

Two dimensions are compatible when they are equal, or one of them is 1.

If the arrays have compatible shapes, they are broadcasted to a common shape.

'''

import numpy as np

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

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

# Broadcasting b to the shape of a

result = a + b

print('A+B: ',result)

#Practical Application ==> Broadcasting can simplify code and make operations more efficient.

# Element-wise addition with broadcasting

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

vector = np.array([1, 0, 1])

result = matrix + vector

print('A M + B V',result)

'''

Common Pitfalls

Mismatch in shapes: Ensure arrays have compatible shapes before broadcasting.

Unintended broadcasting: Be careful with operations that might unintentionally

broadcast arrays, leading to unexpected results.

'''

# Mismatch shapes

try:

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

b = np.array([1, 2])

result = a + b

except ValueError as e:

print("ValueError:", e)

'''Memory Layout

C-order and F-order

NumPy arrays can be stored in memory in C-order (row-major) or F-order (column-major).

'''

array = np.array([[1, 2, 3], [4, 5, 6]], order='C')

print("C-order:\n", array)

array = np.array([[1, 2, 3], [4, 5, 6]], order='F')

print("F-order:\n", array)

#Reshaping an array can affect memory layout. Using reshape often creates a view

# instead of a copy, but it can create a copy if necessary.

array = np.arange(6).reshape(2, 3)

print("Original array:\n", array)

reshaped\_array = array.reshape(3, 2)

print("Reshaped array:\n", reshaped\_array)

# Views vs. Copies ==> A view is a new array object that looks at the same data,

# whereas a copy is a new array with its own data.

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

# View

view = array[1:3]

print('view : ',view)

view[0] = 99

print("Original array after view modification:", array)

print('view : ',view)

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

# Copy

copy = array[1:3].copy()

print('copied : ', copy)

copy[0] = 88

print("Original array after copy modification:", array)

print('copied : ', copy)

'''Performance Optimization ==> Vectorization for Performance

Vectorization allows for faster computations by applying operations on entire arrays

rather than using loops.

Avoiding Loops with NumPy

Using NumPy functions instead of Python loops can significantly speed up computations.

'''

# Using loop

array = np.arange(1e6)

result = np.zeros\_like(array)

print(array, '\n', result)

for i in range(len(array)):

result[i] = np.sin(array[i])

# Using vectorization

result = np.sin(array)

print('Vec sin : ', result)

'''Numba for JIT Compilation

Numba is a JIT compiler that translates Python functions to optimized machine code at runtime.

'''

from numba import jit

import numpy as np

@jit(nopython=True)

def sum\_array(arr):

total = 0

for i in range(arr.shape[0]):

total += arr[i]

return total

array = np.arange(1e6)

print(sum\_array(array))

# Applications in Machine Learning

# 1. Data Preprocessing with NumPy

# 2. Feature Engineering and Transformation

data = np.random.rand(100, 3) # Random data

# Normalizing data

mean = np.mean(data, axis=0)

std = np.std(data, axis=0)

normalized\_data = (data - mean) / std

print("Normalized data:\n", normalized\_data)

data = np.random.rand(100, 2)

# Polynomial features

poly\_features = np.hstack((data, data\*\*2, data\*\*3))

print("Polynomial features:\n", poly\_features)

# Simple linear regression using NumPy

X = np.random.rand(100, 1)

y = 3\*X.squeeze() + 2 + np.random.randn(100) \* 0.1

# Adding intercept term

X\_b = np.c\_[np.ones((100, 1)), X]

# Normal equation

theta\_best = np.linalg.inv(X\_b.T.dot(X\_b)).dot(X\_b.T).dot(y)

print("Theta best (parameters):", theta\_best)

Pandas

1.pandas basics

[4:44 PM] Poornima (Guest)

import pandas as pd

'''

Basic Usage and Conventions

Pandas provides two main data structures: DataFrame and Series.

A DataFrame is a 2-dimensional labeled data structure with columns of potentially

different types.

A Series is a 1-dimensional labeled array.

'''

#Creating a Series:

data = [1, 2, 3, 4, 5]

print(type(data))

series = pd.Series(data)

print(series)

print(type(series))

#Creating a DataFrame

data = {

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 35],

'City': ['New York', 'San Francisco', 'Los Angeles']

}

df = pd.DataFrame(data)

print(df)

'''

Basic DataFrame Operations:

Head and Tail: View the first or last few rows of the DataFrame.

'''

print('head \n ',df.head()) # First 5 rows by default

print('tail - 2 \n ',df.tail(2)) # Last 2 rows

print(df.info())

print('desc \n',df.describe())

#Selecting Columns

print(df['Name'])

print(df[['Name', 'City']])

#Filtering Rows

print(df[df['Age'] > 30])

2.data\_structures

'''  
Data Structures in Pandas  
Series  
A Series is a one-dimensional labeled array capable of holding data of  
any type (integer, string, float, python objects, etc.).  
The axis labels are collectively referred to as the index.  
  
Creating Series from Lists, Dictionaries, and Arrays  
'''  
  
import pandas as pd  
  
data\_list = [1, 2, 3, 4, 5]  
series\_from\_list = pd.Series(data\_list)  
print(series\_from\_list)  
  
data\_dict = {'a': 1, 'b': 2, 'c': 3}  
series\_from\_dict = pd.Series(data\_dict)  
print(series\_from\_dict)  
  
import numpy as np  
  
data\_array = np.array([1, 2, 3, 4, 5])  
series\_from\_array = pd.Series(data\_array)  
print(series\_from\_array)  
  
'''  
Series Attributes and Methods  
Attributes, Methods  
'''  
  
print(series\_from\_list.index) # RangeIndex(start=0, stop=5, step=1)  
print(series\_from\_list.values) # array([1, 2, 3, 4, 5])  
print(series\_from\_list.dtype) # dtype('int64')  
  
print(series\_from\_list.head(3)) # First 3 elements  
print(series\_from\_list.tail(2)) # Last 2 elements  
print(series\_from\_list.mean()) # Mean value  
print(series\_from\_list.sum()) # Sum of all values  
print(series\_from\_list.describe()) # Statistical summary  
  
  
#Indexing and Slicing Series  
  
print(series\_from\_list[2])  
print(series\_from\_dict['b'])  
  
print(series\_from\_list[1:4])  
print(series\_from\_list[:3])  
print(series\_from\_list[3:])  
  
#Operations on Series  
  
print(series\_from\_list + 2) # Adding 2 to each element  
print(series\_from\_list \* 2) # Multiplying each element by 2  
  
other\_series = pd.Series([10, 20, 30, 40, 50])  
print(series\_from\_list + other\_series)

'''  
DataFrames  
A DataFrame is a 2-dimensional labeled data structure with columns of   
potentially different types. You can think of it as a table or a spreadsheet in Excel.  
  
Creating DataFrames from Dictionaries, Lists, and NumPy Arrays  
  
'''  
  
data\_dict = {  
'Name': ['Alice', 'Bob', 'Charlie'],  
'Age': [25, 30, 35],  
'City': ['New York', 'San Francisco', 'Los Angeles']  
}  
  
df\_from\_dict = pd.DataFrame(data\_dict)  
print(df\_from\_dict)  
  
data\_list\_dicts = [  
{'Name': 'Alice', 'Age': 25, 'City': 'New York'},  
{'Name': 'Bob', 'Age': 30, 'City': 'San Francisco'},  
{'Name': 'Charlie', 'Age': 35, 'City': 'Los Angeles'}  
]  
  
df\_from\_list\_dicts = pd.DataFrame(data\_list\_dicts)  
print(df\_from\_list\_dicts)  
  
data\_array = np.array([  
['Alice', 25, 'New York'],  
['Bob', 30, 'San Francisco'],  
['Charlie', 35, 'Los Angeles']  
])  
  
df\_from\_array = pd.DataFrame(data\_array, columns=['Name', 'Age', 'City'])  
print(df\_from\_array)  
  
'''  
DataFrame Attributes and Methods  
Attributes, Methods  
  
'''  
  
print(df\_from\_dict.index) # RangeIndex(start=0, stop=3, step=1)  
print(df\_from\_dict.columns) # Index(['Name', 'Age', 'City'], dtype='object')  
print(df\_from\_dict.values) # 2D array of the DataFrame values  
  
  
print(df\_from\_dict.head(2)) # First 2 rows  
print(df\_from\_dict.tail(1)) # Last row  
print(df\_from\_dict.info()) # Information about the DataFrame  
print(df\_from\_dict.describe()) # Statistical summary of numeric columns  
  
  
#Indexing and Slicing DataFrames  
print(df\_from\_dict['Name'])  
print(df\_from\_dict[['Name', 'City']])  
  
print(df\_from\_dict.iloc[0]) # By integer index  
print(df\_from\_dict.loc[0]) # By label index (same as iloc in this case)  
  
print(df\_from\_dict.iloc[0:2]) # First 2 rows  
print(df\_from\_dict.loc[0:1]) # Rows with labels 0 and 1  
  
#Single column selection:  
print(df\_from\_dict['Age'])  
  
#Multiple columns selection:  
print(df\_from\_dict[['Name', 'City']])  
  
#Row selection using labels (loc):  
print(df\_from\_dict.loc[0])  
print(df\_from\_dict.loc[0:2])  
  
#Row selection using integer positions (iloc):  
print(df\_from\_dict.iloc[0])  
print(df\_from\_dict.iloc[0:2])

Assessments:

Day 1:

Task 1:

# Difference between Sums of Odd and Even Digits in a given number(use functions)  
def find\_difference(number):  
 odd\_sum = 0  
 even\_sum = 0  
 for digit in str(number):  
 if int(digit) % 2 == 0:  
 even\_sum += int(digit)  
 else:  
 odd\_sum += int(digit)  
 return odd\_sum - even\_sum  
  
# Example usage:  
number = int(input("enter the number: "))  
print("Difference between sums of odd and even digits:", find\_difference(number))

Task 2

# Use Python's re module to find all occurrences of the word "Python" in a given text.  
import re  
  
def find\_all\_occurrences(text, word):  
 pattern = fr'\b{re.escape(word)}\b'  
 matches = re.findall(pattern, text, re.IGNORECASE)  
 return matches  
  
text = "Hello World!!! The World is very beautiful, we must adopt things by looking around the World"  
word = "World"  
occurrences = find\_all\_occurrences(text, word)  
print(f"Occurrences of the word '{word}': {occurrences}")

Task 3:

# You are tasked with creating a Python program that calculates the square root of a non-negative number entered by the user. The program should handle exceptions such as ValueError and NameError appropriately. Additionally, it should include an else block to print the square root if no exception occurs and a finally block to ensure that the program execution is completed. Write the Python program to fulfill these requirements  
  
  
import math  
  
def calculate\_square\_root():  
 try:  
 number = float(input("Enter a non-negative number: "))  
 if number < 0:  
 raise ValueError("Number must be non-negative")  
 result = math.sqrt(number)  
 except ValueError as ve:  
 print("Error:", ve)  
 except NameError:  
 print("Invalid input. Please enter a number.")  
 else:  
 print("Square root:", result)  
 finally:  
 print("Program execution completed.")  
  
# Example usage:  
calculate\_square\_root()

Day 2:

Task 1:

# Question 1: Matrix Operations with NumPy  
#  
# Create two 3x3 matrices A and B with random integer values between 1 and 10.  
# Compute the following:  
# The sum of A and B.  
# The difference between A and B.  
# The element-wise product of A and B.  
# The matrix product of A and B.  
# The transpose of matrix A.  
# The determinant of matrix A.  
  
import numpy as np  
  
# Create two 3x3 matrices A and B with random integer values between 1 and 10  
A = np.random.randint(1, 10, (3, 3))  
B = np.random.randint(1, 10, (3, 3))  
  
# Compute the sum of A and B  
sum\_result = np.add(A, B)  
  
# Compute the difference between A and B  
difference\_result = np.subtract(A, B)  
  
# Compute the element-wise product of A and B  
elementwise\_product\_result = np.multiply(A, B)  
  
# Compute the matrix product of A and B  
matrix\_product\_result = np.dot(A, B)  
  
# Compute the transpose of matrix A  
transpose\_result = np.transpose(A)  
  
# Compute the determinant of matrix A  
determinant\_result = np.linalg.det(A)  
  
print("Sum of A and B:\n", sum\_result)  
print("\nDifference between A and B:\n", difference\_result)  
print("\nElement-wise product of A and B:\n", elementwise\_product\_result)  
print("\nMatrix product of A and B:\n", matrix\_product\_result)  
print("\nTranspose of matrix A:\n", transpose\_result)  
print("\nDeterminant of matrix A:", determinant\_result)

Task 2:

# Question 2: Solving Linear Equations with SciPy  
#  
# Given the system of equations:  
# 2x + 3y = 8  
# 3x + 4y = 11  
#  
# Represent the system of equations in matrix form AX = B.  
# Use scipy.linalg.solve to find the values of x and y.  
  
import numpy as np  
from scipy.linalg import solve  
  
# Represent the system of equations in matrix form AX = B  
A = np.array([[2, 3], [3, 4]])  
B = np.array([8, 11])  
  
# Use scipy.linalg.solve to find the values of x and y  
solution = solve(A, B)  
x, y = solution  
  
print("Solution: x =", x, "y =", y)

Task 3:

# Question 3: Calculus with SciPy  
#  
# Define the function f(x) = x^3 + 2x^2 + x + 1.  
# Compute the first and second derivatives of f(x) at x = 1.  
# Compute the definite integral of f(x) from x = 0 to x = 2.  
  
from scipy import integrate  
from sympy import symbols, diff  
  
# Define the function f(x) = x^3 + 2x^2 + x + 1  
x = symbols('x')  
f = x\*\*3 + 2\*x\*\*2 + x + 1  
  
# Compute the first and second derivatives of f(x) at x = 1  
first\_derivative = diff(f, x)  
second\_derivative = diff(first\_derivative, x)  
  
first\_derivative\_at\_1 = first\_derivative.subs(x, 1)  
second\_derivative\_at\_1 = second\_derivative.subs(x, 1)  
  
# Compute the definite integral of f(x) from x = 0 to x = 2  
definite\_integral\_result, \_ = integrate.quad(lambda x: f.subs('x', x), 0, 2)  
  
print("First derivative of f(x) at x = 1:", first\_derivative\_at\_1)  
print("Second derivative of f(x) at x = 1:", second\_derivative\_at\_1)  
print("Definite integral of f(x) from x = 0 to x = 2:", definite\_integral\_result)

Task 4:

# Question 4: Descriptive Statistics with NumPy and SciPy  
#  
# Create a dataset with 20 random values between 1 and 100.  
# Compute the following statistics for the dataset:  
# Mean  
# Median  
# Standard deviation  
# Variance  
# Skewness  
# Kurtosis  
  
from scipy.stats import describe  
import numpy as np  
# Create a dataset with 20 random values between 1 and 100  
dataset = np.random.randint(1, 100, 20)  
  
# Compute descriptive statistics for the dataset  
stats = describe(dataset)  
  
print("Mean:", stats.mean)  
print("Median:", np.median(dataset))  
print("Standard deviation:", np.std(dataset))  
print("Variance:", np.var(dataset))  
print("Skewness:", stats.skewness)  
print("Kurtosis:", stats.kurtosis)

Task 5:

# Question 5: Hypothesis Testing with SciPy  
# Generate a sample dataset of 30 random values from a normal distribution with mean 50 and standard deviation 5.  
# Perform a one-sample t-test to check if the sample mean is significantly different from 50."  
  
import numpy as np  
  
from scipy.stats import ttest\_1samp  
  
# Generate a sample dataset of 30 random values from a normal distribution with mean 50 and standard deviation 5  
sample\_dataset = np.random.normal(50, 5, 30)  
  
# Perform a one-sample t-test to check if the sample mean is significantly different from 50  
t\_statistic, p\_value = ttest\_1samp(sample\_dataset, 50)  
  
print("T-statistic:", t\_statistic)  
print("P-value:", p\_value)

Day 3

Task 1

# Question 1: One-Sample t-Test  
#  
# Perform a one-sample t-test to determine if the sample mean is significantly different from a known population mean.  
#  
# Generate a sample dataset of 30 random values from a normal distribution with a mean of 60 and a standard deviation of 10.  
# Perform a one-sample t-test to check if the sample mean is significantly different from 50.  
  
from scipy.stats import ttest\_1samp  
import numpy as np  
  
# Generate a sample dataset of 30 random values from a normal distribution with a mean of 60 and a standard deviation of 10  
sample\_dataset = np.random.normal(60, 10, 30)  
  
# Perform a one-sample t-test to check if the sample mean is significantly different from 50  
t\_statistic, p\_value = ttest\_1samp(sample\_dataset, 50)  
  
print("One-Sample t-Test:")  
print("T-statistic:", t\_statistic)  
print("P-value:", p\_value)

Task 2

# Question 2: Two-Sample t-Test  
# Perform a two-sample t-test to compare the means of two independent samples.  
#  
# Generate two sample datasets each with 25 random values from normal distributions with means of 55 and 60, and a standard deviation of 8.  
# Perform an independent two-sample t-test to check if the means of the two samples are significantly different.  
  
  
import numpy as np  
from scipy.stats import ttest\_ind  
  
# Generate two sample datasets  
sample\_dataset1 = np.random.normal(55, 8, 25)  
sample\_dataset2 = np.random.normal(60, 8, 25)  
  
# Perform an independent two-sample t-test to check if the means of the two samples are significantly different  
t\_statistic, p\_value = ttest\_ind(sample\_dataset1, sample\_dataset2)  
  
print("Two-Sample t-Test:")  
print("T-statistic:", t\_statistic)  
print("P-value:", p\_value)

Task 3

# Question 3: Chi-Squared Test  
# Objective: Perform a Chi-Squared test for independence.  
#  
# Create a contingency table with observed frequencies for two categorical variables.  
#  
# |-----------| Category A | Category B |  
# | Group 1 | 10 | 20 |  
# | Group 2 | 15 | 25 |  
#  
# Perform a Chi-Squared test to determine if there is a significant association between the two categorical variables.  
  
import numpy as np  
from scipy.stats import chi2\_contingency  
  
# Create a contingency table with observed frequencies  
observed\_frequencies = np.array([[10, 20],  
 [15, 25]])  
  
# Perform a Chi-Squared test  
chi2\_statistic, p\_value, \_, \_ = chi2\_contingency(observed\_frequencies)  
  
print("Chi-Squared Test:")  
print("Chi-Squared Statistic:", chi2\_statistic)  
print("P-value:", p\_value)

Task 4

# Question 4: One-Way ANOVA  
# Objective: Perform a one-way ANOVA to compare means across multiple groups.  
#  
# Generate three sample datasets each with 20 random values from normal distributions with means of 50, 55, and 60, and a standard deviation of 10.  
# Perform a one-way ANOVA to check if there are any significant differences in means across the three groups.  
  
import numpy as np  
from scipy.stats import f\_oneway  
  
# Generate three sample datasets  
sample\_dataset1 = np.random.normal(50, 10, 20)  
sample\_dataset2 = np.random.normal(55, 10, 20)  
sample\_dataset3 = np.random.normal(60, 10, 20)  
  
# Perform a one-way ANOVA  
f\_statistic, p\_value = f\_oneway(sample\_dataset1, sample\_dataset2, sample\_dataset3)  
  
print("One-Way ANOVA:")  
print("F-statistic:", f\_statistic)  
print("P-value:", p\_value)

Task 5

# Question 5: Post-hoc Test using Tukey's HSD  
# Objective: Perform a post-hoc test using Tukey's HSD to identify which groups are significantly different.  
#  
# Use the same datasets generated in the one-way ANOVA exercise.  
# Perform Tukey's HSD test to find out which pairs of group means are significantly different.  
  
import numpy as np  
from statsmodels.stats.multicomp import pairwise\_tukeyhsd  
  
# Generate three sample datasets  
sample\_dataset1 = np.random.normal(50, 10, 20)  
sample\_dataset2 = np.random.normal(55, 10, 20)  
sample\_dataset3 = np.random.normal(60, 10, 20)  
  
  
# Combine all datasets into one  
combined\_data = np.concatenate([sample\_dataset1, sample\_dataset2, sample\_dataset3])  
  
# Create corresponding group labels  
group\_labels = ['Group 1'] \* 20 + ['Group 2'] \* 20 + ['Group 3'] \* 20  
  
# Perform Tukey's HSD test  
tukey\_results = pairwise\_tukeyhsd(combined\_data, group\_labels)  
  
print("Tukey's HSD Test:")  
print(tukey\_results)

Day 4

Task 1

# Exercise 1: Create different types of NumPy arrays and perform basic manipulations.  
#  
# a. Create a 1-dimensional array:  
# Create a 1-dimensional array of integers from 0 to 9.  
# Print the array and its shape.  
#  
# b. Create a 2-dimensional array:  
# Create a 2-dimensional array (3x3) with values from 1 to 9.  
# Print the array, its shape, and the sum of all elements.  
#  
# c. Reshape the array:  
# Reshape the 1-dimensional array from step 1 into a 2x5 array.  
# Print the reshaped array and its shape.  
  
  
  
import numpy as np  
  
# Exercise 1a  
arr\_1d = np.arange(10)  
print("1D Array:")  
print(arr\_1d)  
print("Shape:", arr\_1d.shape)  
  
# Exercise 1b  
arr\_2d = np.arange(1, 10).reshape(3, 3)  
print("\n2D Array:")  
print(arr\_2d)  
print("Shape:", arr\_2d.shape)  
print("Sum of all elements:", np.sum(arr\_2d))  
  
# Exercise 1c  
reshaped\_arr = arr\_1d.reshape(2, 5)  
print("\nReshaped Array:")  
print(reshaped\_arr)  
print("Shape:", reshaped\_arr.shape)

Task 2

# Exercise 2: Perform Basic and Advanced Array Operations  
#  
# a. Array arithmetic:  
# Create two 1-dimensional arrays of integers from 1 to 5 and 6 to 10.  
# Perform element-wise addition, subtraction, multiplication, and division and Print the results.  
#  
# b. Indexing and slicing:  
# Create a 5x5 array with values from 1 to 25.  
# Extract the subarray consisting of the first two rows and columns.  
# Print the extracted subarray.  
#  
# c. Boolean indexing:  
# Create a 1-dimensional array of integers from 10 to 19.  
# Extract elements greater than 15.  
# Print the resulting array.  
  
import numpy as np  
  
# Exercise 2a  
arr1 = np.arange(1, 6)  
arr2 = np.arange(6, 11)  
print("\nArray 1:", arr1)  
print("Array 2:", arr2)  
print("Addition:", arr1 + arr2)  
print("Subtraction:", arr1 - arr2)  
print("Multiplication:", arr1 \* arr2)  
print("Division:", arr1 / arr2)  
  
# Exercise 2b  
arr\_5x5 = np.arange(1, 26).reshape(5, 5)  
subarray = arr\_5x5[:2, :2]  
print("\nSubarray:")  
print(subarray)  
  
# Exercise 2c  
arr\_1d\_10to19 = np.arange(10, 20)  
print("\nArray 10 to 19:", arr\_1d\_10to19)  
print("Elements > 15:", arr\_1d\_10to19[arr\_1d\_10to19 > 15])

Task 3

# Exercise 3: Use NumPy for Mathematical and Statistical Calculations  
#  
# a. Mathematical functions:  
# Create an array of 10 evenly spaced values between 0 and 2π.  
# Compute the sine, cosine, and tangent of each value.  
# Print the results.  
#  
# b. Statistical functions:  
# Create a 3x3 array with random integers between 1 and 100.  
# Compute the mean, median, standard deviation, and variance.  
# Print the results.  
  
import numpy as np  
  
# Exercise 3a  
evenly\_spaced\_values = np.linspace(0, 2 \* np.pi, 10)  
print("\nEvenly Spaced Values:", evenly\_spaced\_values)  
print("Sine:", np.sin(evenly\_spaced\_values))  
print("Cosine:", np.cos(evenly\_spaced\_values))  
print("Tangent:", np.tan(evenly\_spaced\_values))  
  
# Exercise 3b  
random\_array = np.random.randint(1, 101, size=(3, 3))  
print("\nRandom Array:")  
print(random\_array)  
print("Mean:", np.mean(random\_array))  
print("Median:", np.median(random\_array))  
print("Standard Deviation:", np.std(random\_array))  
print("Variance:", np.var(random\_array))

Task 4

# Exercise 4: Implement Broadcasting and Vectorized Operations  
#  
# a. Broadcasting:  
# Create a 3x1 array with values from 1 to 3.  
# Create a 1x3 array with values from 4 to 6.  
# Add the two arrays using broadcasting.  
# Print the resulting array.  
#  
# b. Vectorized operations:  
# Create two large arrays of size 1,000,000 with random values.  
# Compute the element-wise product of the two arrays.  
# Print the time taken for the computation using vectorized operations.  
import time  
  
import numpy as np  
  
# Exercise 4a  
arr\_3x1 = np.arange(1, 4).reshape(3, 1)  
arr\_1x3 = np.arange(4, 7).reshape(1, 3)  
result\_broadcasting = arr\_3x1 + arr\_1x3  
print("\nBroadcasting Result:")  
print(result\_broadcasting)  
  
# Exercise 4b  
large\_arr1 = np.random.random(1000000)  
large\_arr2 = np.random.random(1000000)  
start\_time = time.time()  
result\_vectorized = large\_arr1 \* large\_arr2  
end\_time = time.time()  
print("\nTime taken for vectorized operation:", end\_time - start\_time, "seconds")

Task 5

# Exercise 5: Optimize Performance Using Vectorization and Numba  
#  
# a. Vectorization:  
# Create a function to compute the element-wise square of an array using a for loop.  
# Create another function to perform the same computation using NumPy vectorization.  
# Compare the performance of the two functions using a large array of size 1,000,000.  
#  
# b. Numba:  
# Use the @numba.jit decorator to optimize the function from step 1 that uses a for loop.  
  
# Compare the performance of the Numba-optimized function with the vectorized NumPy function.  
  
  
import time  
# import numba  
  
import numpy as np  
  
# Exercise 5a  
def square\_with\_loop(arr):  
 result = np.empty\_like(arr)  
 for i in range(len(arr)):  
 result[i] = arr[i] \*\* 2  
 return result  
  
large\_array = np.random.random(1000000)  
  
start\_time = time.time()  
result\_loop = square\_with\_loop(large\_array)  
end\_time = time.time()  
print("\nTime taken for loop computation:", end\_time - start\_time, "seconds")  
  
start\_time = time.time()  
result\_vectorized = large\_array \*\* 2  
end\_time = time.time()  
print("Time taken for vectorized computation:", end\_time - start\_time, "seconds")  
  
# Exercise 5b  
# @numba.jit  
def square\_with\_numba(arr):  
 result = np.empty\_like(arr)  
 for i in range(len(arr)):  
 result[i] = arr[i] \*\* 2  
 return result  
  
start\_time = time.time()  
result\_numba = square\_with\_numba(large\_array)  
end\_time = time.time()  
print("Time taken for Numba-optimized computation:", end\_time - start\_time, "seconds")