Q.1) Implement a Python class MaxHeap that supports the following operations: insert, delete, and get\_max. Ensure the operations maintain the properties of a max-heap.

class MaxHeap:

    def \_\_init\_\_(self):

        self.heap = []

    def insert(self, value):

        self.heap.append(value)

        self.\_heapify\_up(len(self.heap) - 1)

    def delete(self):

        if len(self.heap) == 0:

            return None

        if len(self.heap) == 1:

            return self.heap.pop()

        root\_value = self.heap[0]

        self.heap[0] = self.heap.pop()

        self.\_heapify\_down(0)

        return root\_value

    def get\_max(self):

        if not self.heap:

            return None

        return self.heap[0]

    def \_heapify\_up(self, index):

        parent\_index = (index - 1) // 2

        if index > 0 and self.heap[index] > self.heap[parent\_index]:

            self.heap[index], self.heap[parent\_index] = self.heap[parent\_index], self.heap[index]

            self.\_heapify\_up(parent\_index)

    def \_heapify\_down(self, index):

        largest = index

        left\_child\_index = 2 \* index + 1

        right\_child\_index = 2 \* index + 2

        if left\_child\_index < len(self.heap) and self.heap[left\_child\_index] > self.heap[largest]:

            largest = left\_child\_index

        if right\_child\_index < len(self.heap) and self.heap[right\_child\_index] > self.heap[largest]:

            largest = right\_child\_index

        if largest != index:

            self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]

            self.\_heapify\_down(largest)

# Example usage:

heap = MaxHeap()

heap.insert(10)

heap.insert(20)

heap.insert(15)

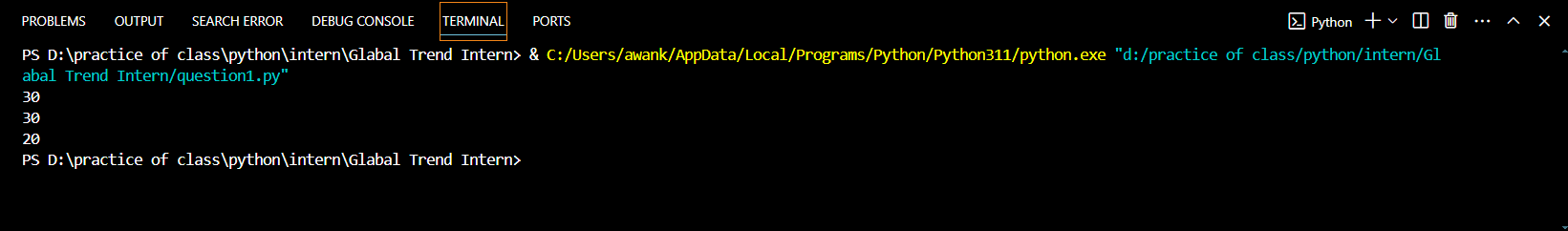
heap.insert(30)

print(heap.get\_max())

print(heap.delete())

print(heap.get\_max())

output:-



# 2) Write a Python function that takes a list of URLs, attempts to download their content, and retries up to 3 times if an error occurs. Use appropriate error handling to manage different types of exceptions.

import requests

from time import sleep

def download\_content(urls):

    results = {}

    max\_retries = 3

    for url in urls:

        success = False

        attempts = 0

        while not success and attempts < max\_retries:

            try:

                response = requests.get(url)

                response.raise\_for\_status()

                results[url] = response.text

                success = True

            except requests.exceptions.HTTPError as http\_err:

                print(f"HTTP error occurred: {http\_err} - {url}")

            except requests.exceptions.ConnectionError as conn\_err:

                print(f"Connection error occurred: {conn\_err} - {url}")

            except requests.exceptions.Timeout as timeout\_err:

                print(f"Timeout error occurred: {timeout\_err} - {url}")

            except requests.exceptions.RequestException as req\_err:

                print(f"Error occurred: {req\_err} - {url}")

            if not success:

                attempts += 1

                if attempts < max\_retries:

                    print(f"Retrying {url} ({attempts}/{max\_retries})...")

                    sleep(1)  # Wait for 1 second before retrying

        if not success:

            results[url] = None

            print(f"Failed to download content from {url} after {max\_retries} attempts")

    return results

# Example usage:

urls = [

    "https://www.google.com",

    "https://www.python.org",

]

content = download\_content(urls)

for url, data in content.items():

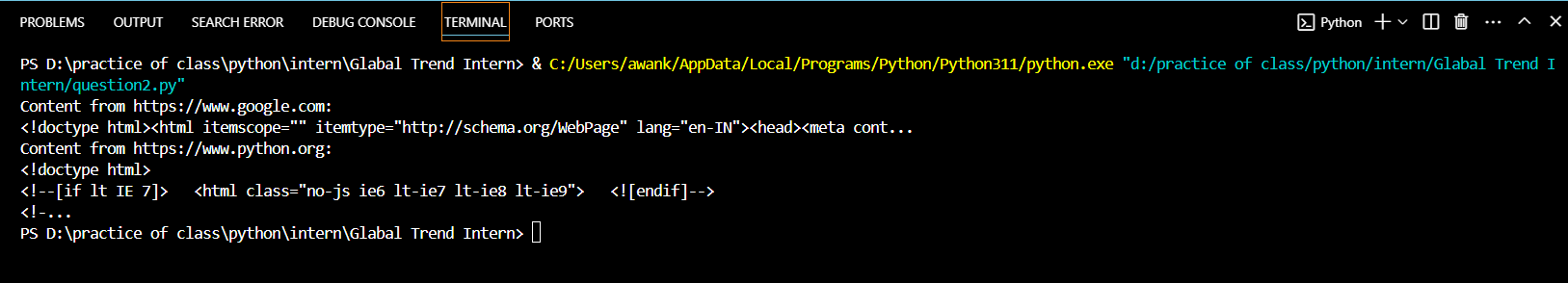
    if data:

        print(f"Content from {url}:\n{data[:100]}...")  # Print first 100 characters

    else:

        print(f"Failed to retrieve content from {url}")

output:-



#3) Write a Python script that trains a simple linear regression model using scikit-learn. Use a dataset of your choice, split it into training and testing sets, and evaluate the model's performance.

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.datasets import fetch\_california\_housing

import warnings

# any warnings

warnings.filterwarnings(action='ignore', category=FutureWarning)

# Load the dataset

california = fetch\_california\_housing()

data = pd.DataFrame(california.data, columns=california.feature\_names)

data['MedHouseVal'] = california.target

# Define the feature matrix (X) and the target vector (y)

X = data.drop('MedHouseVal', axis=1)

y = data['MedHouseVal']

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a linear regression model

model = LinearRegression()

# Train the model

model.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = model.predict(X\_test)

# Evaluate the model's performance

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Mean Squared Error: {mse:.2f}")

print(f"R^2 Score: {r2:.2f}")

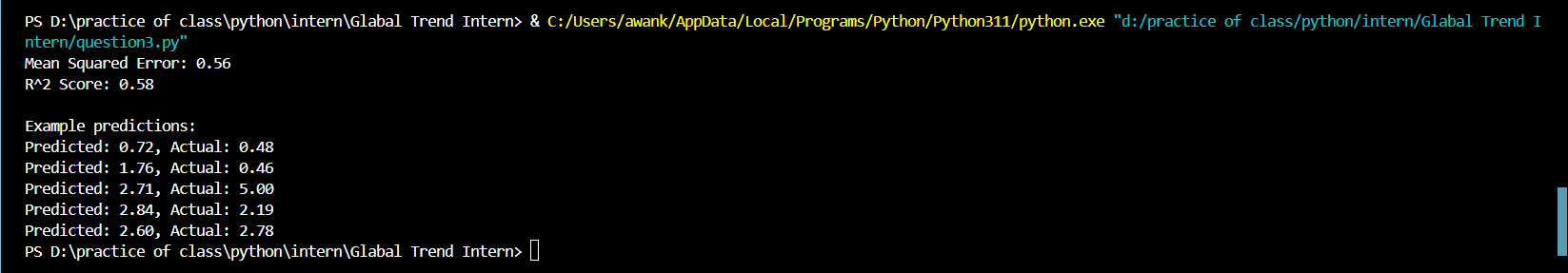
# Example usage:

print("\nExample predictions:")

for i in range(5):

    print(f"Predicted: {y\_pred[i]:.2f}, Actual: {y\_test.values[i]:.2f}")

output:-



# Using pandas, write a Python function to clean and preprocess a given DataFrame, which involves handling missing values, normalizing numerical columns, and encoding categorical columns.

import pandas as pd

from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

from sklearn.impute import SimpleImputer

def preprocess\_dataframe(df):

    # Separate numerical and categorical columns

    numerical\_cols = df.select\_dtypes(include=['float64', 'int64']).columns

    categorical\_cols = df.select\_dtypes(include=['object', 'category']).columns

    # Handle missing values for numerical columns (fill with median)

    num\_imputer = SimpleImputer(strategy='median')

    df[numerical\_cols] = num\_imputer.fit\_transform(df[numerical\_cols])

    # Handle missing values for categorical columns (fill with most frequent)

    cat\_imputer = SimpleImputer(strategy='most\_frequent')

    df[categorical\_cols] = cat\_imputer.fit\_transform(df[categorical\_cols])

    # Normalize numerical columns using Min-Max scaling

    scaler = MinMaxScaler()

    df[numerical\_cols] = scaler.fit\_transform(df[numerical\_cols])

    # Encode categorical columns using one-hot encoding

    df = pd.get\_dummies(df, columns=categorical\_cols, drop\_first=True)

    return df

# Example usage:

# Assume `data` is a pandas DataFrame that you want to preprocess

# data = pd.read\_csv('your\_dataset.csv')

# cleaned\_data = preprocess\_dataframe(data)

# print(cleaned\_data.head())

#5) Write a Python function to compute the nth Fibonacci number using recursion.

def fibonacci(n):

    # Base cases

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    # Recursive case

    else:

        return fibonacci(n - 1) + fibonacci(n - 2)

# Example usage:

n = int(input("Enter the number: "))

print(f"The {n}th Fibonacci number is: {fibonacci(n)}")

output:-



# 6) Write a Python function that divides two numbers and handles the case where the divisor is zero by returning a custom error message.

def safe\_divide(numerator, denominator):

    try:

        result = numerator / denominator

        return result

    except ZeroDivisionError:

        return "Error: Division by zero is not allowed."

# Example usage:

num = 10

den = 0

print(f"Division result: {safe\_divide(num, den)}")

den = 2

print(f"Division result: {safe\_divide(num, den)}")

output:-



# 7) Write a Python decorator that measures the execution time of a function and logs it. Apply this decorator to a function that performs a computationally expensive task.

import time

import logging

# Configure logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def measure\_time(func):

    def wrapper(\*args, \*\*kwargs):

        start\_time = time.time()

        result = func(\*args, \*\*kwargs)

        end\_time = time.time()

        execution\_time = end\_time - start\_time

        logging.info(f"Execution time of {func.\_\_name\_\_}: {execution\_time:.4f} seconds")

        return result

    return wrapper

@measure\_time

def expensive\_computation(n):

    result = 0

    for i in range(n):

        result += sum([j \*\* 2 for j in range(1000)])

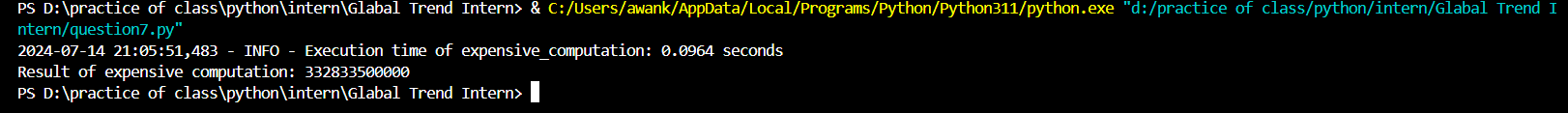
    return result

# Example usage:

n = 1000

print(f"Result of expensive computation: {expensive\_computation(n)}")

output:-



# 8) Write a Python function that takes two numbers and an operator (as a string) and performs the corresponding arithmetic operation (addition, subtraction, multiplication, or division).

def arithmetic\_operation(num1, num2, operator):

    if operator == '+':

        return num1 + num2

    elif operator == '-':

        return num1 - num2

    elif operator == '\*':

        return num1 \* num2

    elif operator == '/':

        try:

            return num1 / num2

        except ZeroDivisionError:

            return "Error: Division by zero is not allowed."

    else:

        return "Error: Invalid operator. Please use one of '+', '-', '\*', '/'."

# Example usage:

num1 = 10

num2 = 5

print(f"{num1} + {num2} = {arithmetic\_operation(num1, num2, '+')}")

print(f"{num1} - {num2} = {arithmetic\_operation(num1, num2, '-')}")

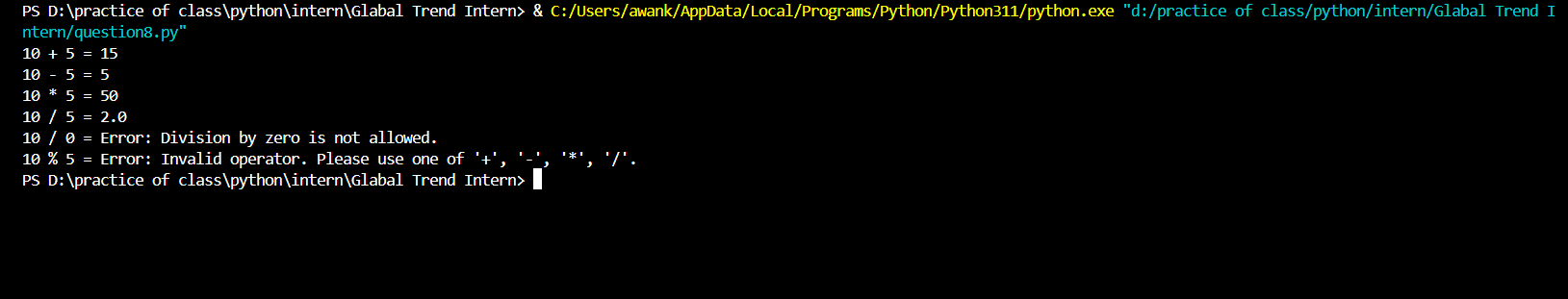
print(f"{num1} \* {num2} = {arithmetic\_operation(num1, num2, '\*')}")

print(f"{num1} / {num2} = {arithmetic\_operation(num1, num2, '/')}")

print(f"{num1} / 0 = {arithmetic\_operation(num1, 0, '/')}")

print(f"{num1} % {num2} = {arithmetic\_operation(num1, num2, '%')}")

output :-



# 9) Write a Python function that generates a random password. The password should contain a mix of uppercase letters, lowercase letters, digits, and special characters.

import string

import random

def generate\_random\_password(length=12):

    # Define characters to choose from

    lowercase\_letters = string.ascii\_lowercase

    uppercase\_letters = string.ascii\_uppercase

    digits = string.digits

    special\_characters = string.punctuation  # includes !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~

    # Combine all characters

    all\_characters = lowercase\_letters + uppercase\_letters + digits + special\_characters

    # Generate password

    password = ''.join(random.choice(all\_characters) for \_ in range(length))

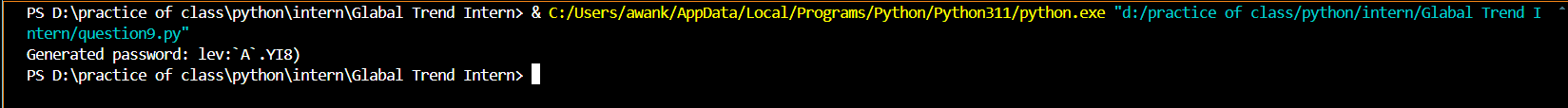
    return password

# Example usage:

password = generate\_random\_password()

print(f"Generated password: {password}")

output:-



# Write a Python function that takes a 2D list (matrix) and returns its transpose.

def transpose\_matrix(matrix):

    if not matrix:

        return []

    num\_rows = len(matrix)

    num\_cols = len(matrix[0])

    # Initialize the transpose matrix with appropriate dimensions

    transpose = [[0] \* num\_rows for \_ in range(num\_cols)]

    # Compute the transpose

    for i in range(num\_rows):

        for j in range(num\_cols):

            transpose[j][i] = matrix[i][j]

    return transpose

# Example usage:

matrix = [

    [1, 2, 3],

    [4, 5, 6],

    [7, 8, 9]

]

print("Original Matrix:")

for row in matrix:

    print(row)

print("\nTranspose Matrix:")

transpose = transpose\_matrix(matrix)

for row in transpose:

    print(row)

output:-

