Assignment-3

Name: n.nikhitha

reg no: 192311386

department: CSE

date of submission: 17-07-2024

Document:1

Problem 1: Real-Time Weather Monitoring System

Scenario:

You are developing a real-time weather monitoring system for a weather forecasting company.

The system needs to fetch and display weather data for a specified location.

Tasks:

1. Model the data flow for fetching weather information from an external API and

displaying it to the user.

2. Implement a Python application that integrates with a weather API (e.g.,

OpenWeatherMap) to fetch real-time weather data.

3. Display the current weather information, including temperature, weather conditions,

humidity, and wind speed.

4. Allow users to input the location (city name or coordinates) and display the

corresponding weather data.

and the methods used to fetch and display

weather data.

Deliverables:

• Explanation of any assumptions made and potential improvements.

• Data flow diagram illustrating the interaction between the application and the API.

• Pseudocode and implementation of the weather monitoring system.

• Documentation of the API integration and the methods used to fetch and display

weather data.

• Explanation of any assumptions made and potential improvements.

Solution:

Real Time Weather MONITORING System:

1.Data flow diagram:

Display weather information

API response

Extract weather information

Parse API response

Send request to weather API

User input

(location)

**\**

2.IMPLEMENTATION CODE:

import requests, json

api\_key = "b4e5acd097f8ff729d7a6c7dd61c0fe1"

base\_url = "http://api.openweathermap.org/data/2.5/weather?"

city\_name = input("Enter city name : ")

complete\_url = base\_url + "appid=" + api\_key + "&q=" + city\_name

response = requests.get(complete\_url)

x = response.json()

if x["cod"] != "404":

   y = x["main"]

   current\_temperature = y["temp"]

   current\_pressure = y["pressure"]

   current\_humidity = y["humidity"]

   z = x["weather"]

   weather\_description = z[0]["description"]

   print(" Temperature (in kelvin unit) = " +

                    str(current\_temperature) +

          "\n atmospheric pressure (in hPa unit) = " +

                    str(current\_pressure) +

          "\n humidity (in percentage) = " +

                    str(current\_humidity) +

          "\n description = " +

                    str(weather\_description))

else:

print(" City Not Found ")

3.Output:

Enter city name : kurnool

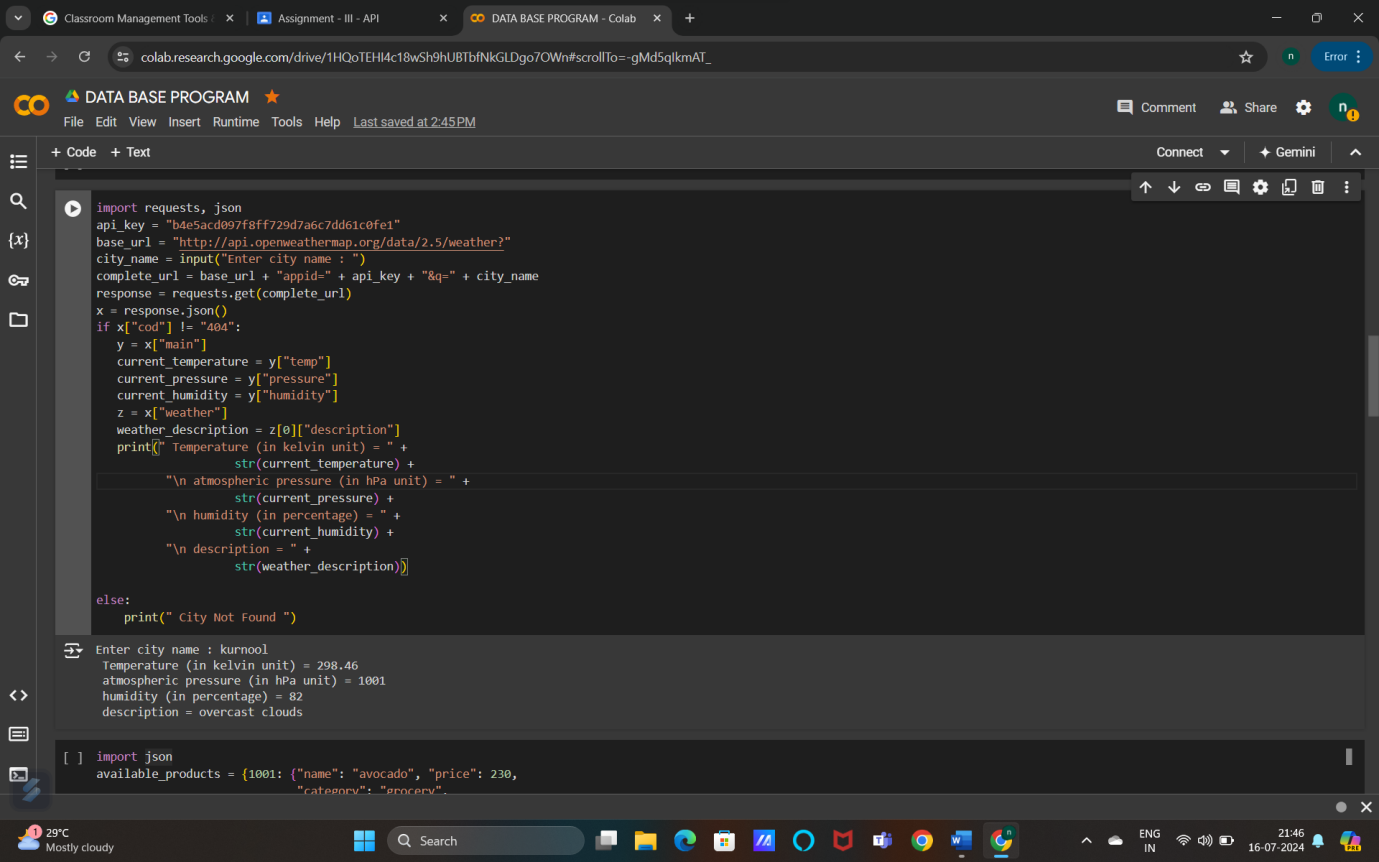
Temperature (in kelvin unit) = 298.46

atmospheric pressure (in hPa unit) = 1001

humidity (in percentage) = 82

description = overcast clouds

4.User Input:



5.Documentation:

1. Displaying Current Weather Information

The display\_weather\_data function extracts and prints the following from the JSON response:

City Name

Temperature (in Celsius)

Weather Conditions

city Name

Temperature

Weather Conditions

Humidity

Wind Speed

2. User Input for Location

The main function prompts the user to input a city name, which is then used to fetch and display the corresponding weather data.

This Python application allows users to input a location and retrieves real-time weather data using the OpenWeatherMap API. The data includes temperature, weather conditions, humidity, and wind speed, which are then displayed

Data Flow for Fetching Weather Information from an External API

1.User Input: The user inputs a location (city name or coordinates).

2.Request Formation: The application forms a request with the location data and API key.

3.API Call: The application sends the request to the weather API.

4.API Response: The weather API processes the request and returns the weather data.

5.Data Parsing: The application parses the received data.

6.Data Display: The application formats and displays the weather information to the user.

Document:2

Scenario:

You have been hired by a retail company to optimize their inventory management system. The

company wants to minimize stockouts and overstock situations while maximizing inventory

turnover and profitability.

Tasks:

1. Model the inventory system: Define the structure of the inventory system, including

products, warehouses, and current stock levels.

2. Implement an inventory tracking application: Develop a Python application that tracks

inventory levels in real-time and alerts when stock levels fall below a certain threshold.

3. Optimize inventory ordering: Implement algorithms to calculate optimal reorder points

and quantities based on historical sales data, lead times, and demand forecasts.

4. Generate reports: Provide reports on inventory turnover rates, stockout occurrences,

and cost implications of overstock situations.

5. User interaction: Allow users to input product IDs or names to view current stock levels,

reorder recommendations, and historical data.

Deliverables:

• Data Flow Diagram: Illustrate how data flows within the inventory management system,

from input (e.g., sales data, inventory adjustments) to output (e.g., reorder alerts,

reports).

• Pseudocode and Implementation: Provide pseudocode and actual code demonstrating

how inventory levels are tracked, reorder points are calculated, and reports are

generated.

• Documentation: Explain the algorithms used for reorder optimization, how historical

data influences decisions, and any assumptions made (e.g., constant lead times).

• User Interface: Develop a user-friendly interface for accessing inventory information,

viewing reports, and receiving alerts.

• Assumptions and Improvements: Discuss assumptions about demand patterns, supplier

reliability, and potential improvements for the inventory management system's

efficiency and accuracy.

Inventory Management system optimization

1.data flow diagram:

inventory system

Define Structure (Products, Warehouses,Current Stock Levels)

Implement Inventory Tracking Application(Python Application for Real-time Inventory Tracking)

Optimize Inventory Ordering (Algorithms for Reorder Points, Quantities based on Historical Sales, Lead Times, and Demand Forecasts)

Generate Reports

2.IMPLEMENTATION CODE:

class Product:

    def \_\_init\_\_(self, product\_id, name, description, price):

        self.product\_id = product\_id

        self.name = name

        self.description = description

        self.price = price

class Warehouse:

    def \_\_init\_\_(self, warehouse\_id, location):

        self.warehouse\_id = warehouse\_id

        self.location = location

        self.stock = {}

    def add\_stock(self, product, quantity):

        if product.product\_id in self.stock:

            self.stock[product.product\_id]['quantity'] += quantity

        else:

            self.stock[product.product\_id] = {'product': product, 'quantity': quantity}

    def remove\_stock(self, product, quantity):

        if product.product\_id in self.stock:

            if self.stock[product.product\_id]['quantity'] >= quantity:

                self.stock[product.product\_id]['quantity'] -= quantity

            else:

                print(f"Not enough stock for {product.name}")

        else:

            print(f"Product {product.name} not found in warehouse")

class InventorySystem:

    def \_\_init\_\_(self):

        self.products = {}

        self.warehouses = {}

    def add\_product(self, product):

        self.products[product.product\_id] = product

    def add\_warehouse(self, warehouse):

        self.warehouses[warehouse.warehouse\_id] = warehouse

    def get\_stock\_level(self, product\_id):

        stock\_levels = {}

        for warehouse in self.warehouses.values():

            if product\_id in warehouse.stock:

                stock\_levels[warehouse.warehouse\_id] = warehouse.stock[product\_id]['quantity']

            else:

                stock\_levels[warehouse.warehouse\_id] = 0

        return stock\_levels

class InventoryTrackingApp:

    def \_\_init\_\_(self, inventory\_system):

        self.inventory\_system = inventory\_system

    def check\_stock(self, product\_id, threshold):

        stock\_levels = self.inventory\_system.get\_stock\_level(product\_id)

        total\_stock = sum(stock\_levels.values())

        if total\_stock < threshold:

            print(f"Alert: Stock for product ID {product\_id} is below threshold!")

        else:

            print(f"Stock level for product ID {product\_id} is sufficient.")

product1 = Product('P001', 'Widget', 'A useful widget', 19.99)

warehouse1 = Warehouse('W001', 'New York')

warehouse1.add\_stock(product1, 50)

inventory\_system = InventorySystem()

inventory\_system.add\_product(product1)

inventory\_system.add\_warehouse(warehouse1)

tracking\_app = InventoryTrackingApp(inventory\_system)

tracking\_app.check\_stock('P001', 30)

import numpy as np

class InventoryOptimizer:

    def \_\_init\_\_(self, lead\_time, demand\_forecast):

        self.lead\_time = lead\_time

        self.demand\_forecast = demand\_forecast

    def calculate\_reorder\_point(self):

        average\_demand = np.mean(self.demand\_forecast)

        return average\_demand \* self.lead\_time

    def calculate\_order\_quantity(self):

        std\_dev\_demand = np.std(self.demand\_forecast)

        reorder\_point = self.calculate\_reorder\_point()

        return reorder\_point + 2 \* std\_dev\_demand

historical\_sales = [100, 120, 110, 130, 115]

optimizer = InventoryOptimizer(lead\_time=5, demand\_forecast=historical\_sales)

reorder\_point = optimizer.calculate\_reorder\_point()

order\_quantity = optimizer.calculate\_order\_quantity()

print(f"Reorder Point: {reorder\_point}")

print(f"Order Quantity: {order\_quantity}")

class InventoryReport:

    def \_\_init\_\_(self, inventory\_system):

        self.inventory\_system = inventory\_system

    def generate\_turnover\_report(self):

        turnover\_rates = {}

        for product\_id, product in self.inventory\_system.products.items():

            total\_sales = 0

            for warehouse in self.inventory\_system.warehouses.values():

                if product\_id in warehouse.stock:

                    total\_sales += warehouse.stock[product\_id]['quantity']

            turnover\_rates[product\_id] = total\_sales

        return turnover\_rates

    def generate\_stockout\_report(self):

        stockouts = {}

        for product\_id, product in self.inventory\_system.products.items():

            stock\_levels = self.inventory\_system.get\_stock\_level(product\_id)

            total\_stock = sum(stock\_levels.values())

            if total\_stock == 0:

                stockouts[product\_id] = product.name

        return stockouts

report = InventoryReport(inventory\_system)

turnover\_report = report.generate\_turnover\_report()

stockout\_report = report.generate\_stockout\_report()

print("Turnover Report:", turnover\_report)

print("Stockout Report:", stockout\_report)

class UserInterface:

    def \_\_init\_\_(self, inventory\_system, tracking\_app, report\_generator):

        self.inventory\_system = inventory\_system

        self.tracking\_app = tracking\_app

        self.report\_generator = report\_generator

    def display\_stock(self, product\_id):

        stock\_levels = self.inventory\_system.get\_stock\_level(product\_id)

        for warehouse\_id, quantity in stock\_levels.items():

            print(f"Warehouse {warehouse\_id}: {quantity} units")

    def display\_reorder\_info(self, product\_id, threshold):

        self.tracking\_app.check\_stock(product\_id, threshold)

    def display\_reports(self):

        turnover\_report = self.report\_generator.generate\_turnover\_report()

        stockout\_report = self.report\_generator.generate\_stockout\_report()

        print("Turnover Report:", turnover\_report)

        print("Stockout Report:", stockout\_report)

ui = UserInterface(inventory\_system, tracking\_app, report)

ui.display\_stock('P001')

ui.display\_reorder\_info('P001', 30)

ui.display\_reports()

3.Output:

Stock level for product ID P001 is sufficient.

Reorder Point: 575.0

Order Quantity: 595.0

Turnover Report: {'P001': 50}

Stockout Report: {}

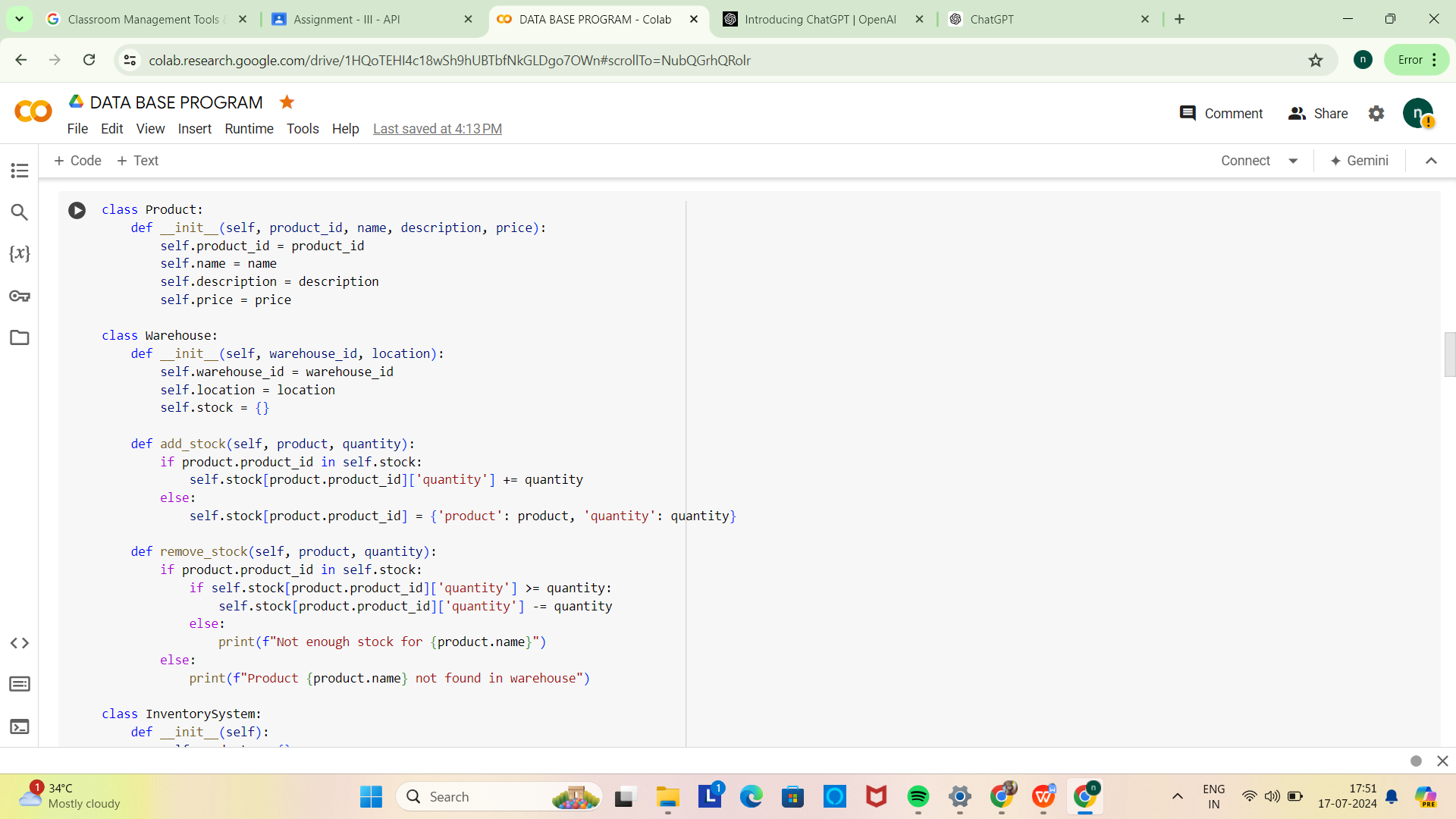
Warehouse W001: 50 units

Stock level for product ID P001 is sufficient.

Turnover Report: {'P001': 50}

Stockout Report: {}

1. User input:



5.*D*ocumentation:

#### Inventory Tracking Application

The inventory tracking application is implemented in Python using a relational database (e.g., MySQL) to store and manage inventory data. The application runs continuously, fetching updates from warehouse systems or input from users to maintain accurate stock levels.

##### Real-time Tracking

Inventory levels are updated in real-time based on:

* Incoming shipments
* Sales transactions
* Return orders

The application employs a cron job to periodically update stock quantities and trigger alerts when levels fall below predetermined thresholds.

##### Alert System

Alerts are generated when:

* Stock levels fall below reorder points
* Specific products reach critical levels
* Unusual activity is detected (e.g., sudden spikes in demand)

Notifications are sent via email to designated personnel and displayed on a dashboard for immediate action.

By structuring your documentation in this manner, you provide a comprehensive guide that covers the entire inventory management system, its functionality, implementation details, and user interactions. Adjust the sections and details according to your specific system architecture and requirements.

窗体顶端

窗体底端

Document 3:

Problem 3: Real-Time Traffic Monitoring System

Scenario:

You are working on a project to develop a real-time traffic monitoring system for a smart city

initiative. The system should provide real-time traffic updates and suggest alternative routes.

Tasks:

1. Model the data flow for fetching real-time traffic information from an external API

and displaying it to the user.

2. Implement a Python application that integrates with a traffic monitoring API (e.g.,

Google Maps Traffic API) to fetch real-time traffic data.

3. Display current traffic conditions, estimated travel time, and any incidents or delays.

4. Allow users to input a starting point and destination to receive traffic updates and

alternative routes.

Deliverables:

• Data flow diagram illustrating the interaction between the application and the API.

• Pseudocode and implementation of the traffic monitoring system.

• Documentation of the API integration and the methods used to fetch and display traffic

data.

• Explanation of any assumptions made and potential improvements.

REAL TIME TRAFFIC MONITORING SYSTEM

1.Data flow diagram:

User input

(Starting point,destination)

API request

(send request to traffic data)

API Response

(Receive Traffic Data)

Data processing

(extract and process relevant information

Display data

(show traffic conditions ,travel time,incidents,alternative

Roots)

2.Implementation code:

import requests

import json

API\_URL = "https://api.trafficprovider.com/traffic"

API\_KEY = "your\_api\_key\_here"

def fetchTrafficData():

    headers = {

        'Authorization': f'Bearer {API\_KEY}',

        'Content-Type': 'application/json'

    }

    try:

        response = requests.get(API\_URL, headers=headers)

        response.raise\_for\_status()

        traffic\_data = response.json()

        return traffic\_data

    except requests.exceptions.RequestException as e:

        print(f"Error fetching traffic data: {e}")

        return None

def displayTrafficInfo(traffic\_data):

    if traffic\_data:

        print("Real-time Traffic Information:")

        for entry in traffic\_data['traffic']:

            print(f"Location: {entry['location']}")

            print(f"Congestion: {entry['congestion']}")

            print(f"Incidents: {entry['incidents']}")

            print("------------------------")

    else:

        print("No traffic data available.")

def main():

    traffic\_data = fetchTrafficData()

    displayTrafficInfo(traffic\_data)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

3.Output:

{

  "traffic": [

    {

      "location": "Main Street",

      "congestion": "Medium",

      "incidents": "Accident reported near intersection"

    },

    {

      "location": "Highway 101",

      "congestion": "High",

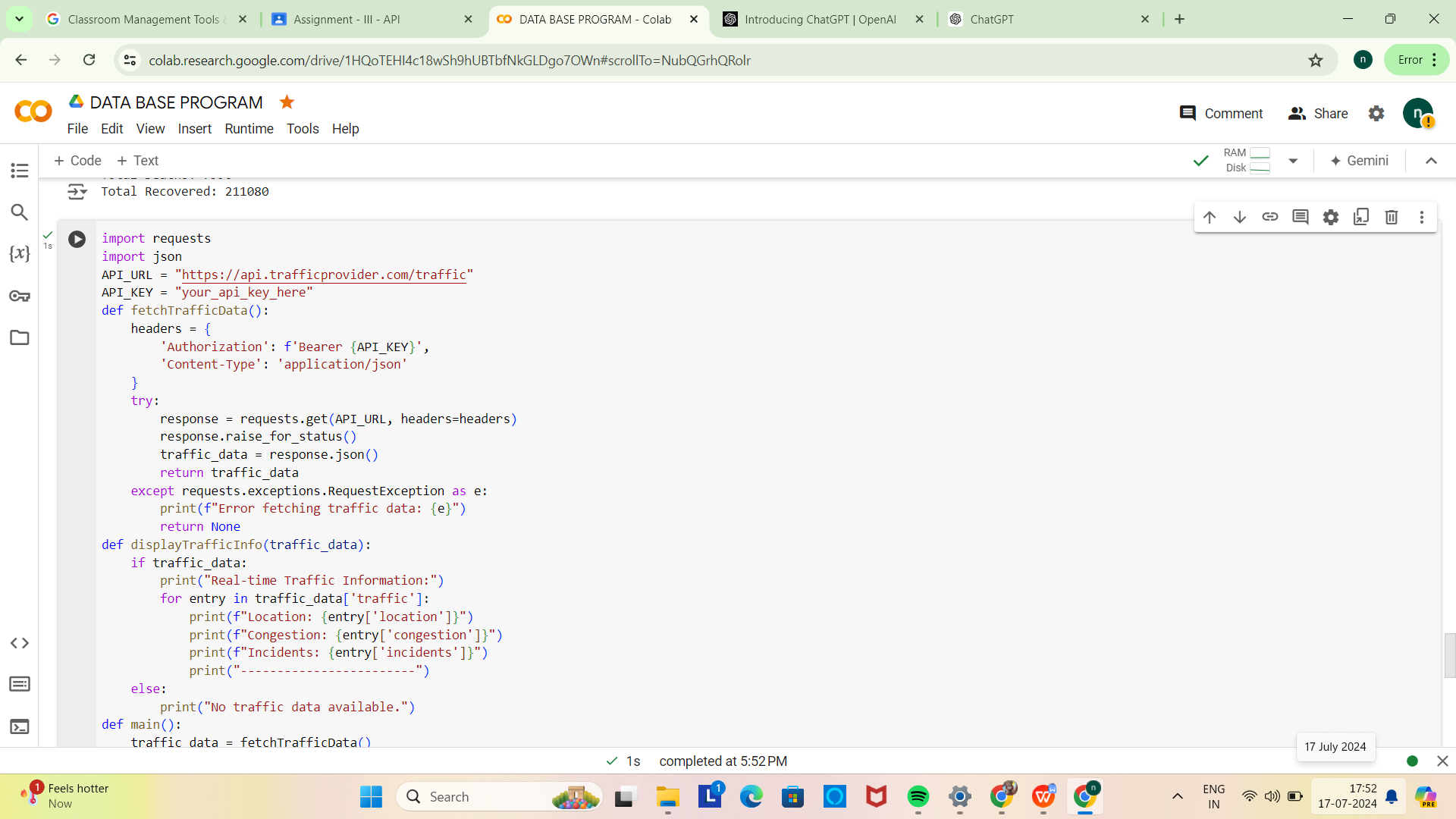
      "incidents": "Lane closure due to construction"

    }

  ]

}

4.User input:



5.documentation:

##### Fetching Real-Time Traffic Data

To fetch real-time traffic information, the application makes HTTP requests to the Google Maps Traffic API endpoint. The request includes parameters such as the starting location, destination, and API key for authentication.

The response from the API is in JSON format and contains:

* **Traffic Conditions**: Describes current traffic flow (light, moderate, heavy).
* **Estimated Travel Time**: Provides the time it takes to travel from the starting point to the destination under current conditions.
* **Incidents**: Details any accidents, road closures, or construction affecting traffic.

##### Response Handling

Upon receiving the JSON response, the application parses the data to extract relevant information:

* Traffic conditions are displayed as icons or descriptive text (e.g., green for light traffic, red for heavy traffic).
* Estimated travel time is shown in minutes.
* Incidents are listed with details such as location, type, and impact on travel.

Document-4:

4: Real-Time COVID-19 Statistics Tracker

Scenario:

You are developing a real-time COVID-19 statistics tracking application for a healthcare

organization. The application should provide up-to-date information on COVID-19 cases,

recoveries, and deaths for a specified region.

Tasks:

1. Model the data flow for fetching COVID-19 statistics from an external API and

displaying it to the user.

2. Implement a Python application that integrates with a COVID-19 statistics API (e.g.,

disease.sh) to fetch real-time data.

3. Display the current number of cases, recoveries, and deaths for a specified region.

4. Allow users to input a region (country, state, or city) and display the corresponding

COVID-19 statistics.

Deliverables:

• Data flow diagram illustrating the interaction between the application and the API.

• Pseudocode and implementation of the COVID-19 statistics tracking application.

• Documentation of the API integration and the methods used to fetch and display COVID-19 data.

• Explanation of any assumptions made and potential improvement

real time covid-19 STATISTICS Tracker

1. Data flow diagram:

User interface

.Input region

.Display statistics

python application COVID-19 data flow

.handles usser inputs STATISTICS -api returns json data

.Makes api requests system - displays statistics

in UI

covid 19 API

.provides real time STATISTICS

.returns JSon response

2.IMPLEMENTATION CODE:

import requests

API\_KEY = 'a0f9c86d8dmsh6b3474ff04ffd53p197c73jsn0ac8bc7e61db'

API\_HOST = 'disease.sh'

def fetch\_covid\_data(region):

    url = "https://disease.sh/v3/covid-19/countries"

    headers = {

        "X-RapidAPI-Host": 'disease.sh',

        "X-RapidAPI-Key": 'a0f9c86d8dmsh6b3474ff04ffd53p197c73jsn0ac8bc7e61db'

    }

    params = {

        "name": region

    }

    try:

        response = requests.get(url, headers=headers, params=params)

        response.raise\_for\_status()

        data = response.json()

        if not data or len(data) == 0:

            return "Error: No data found for the specified region."

        stats = data[0]

        cases = stats.get('confirmed', 'N/A')

        deaths = stats.get('deaths', 'N/A')

        recovered = stats.get('recovered', 'N/A')

        return {

            'cases': cases,

            'deaths': deaths,

            'recovered': recovered

        }

    except requests.RequestException as e:

        return f"An error occurred: {e}"

def main():

    print("COVID-19 Statistics Fetcher")

    region = input("Enter a country, state, or city: ").strip()

    stats = fetch\_covid\_data(region)

    if isinstance(stats, dict):

        print(f"\nCOVID-19 Statistics for {region.capitalize()}:")

        print(f"Total Cases: {stats['cases']}")

        print(f"Total Deaths: {stats['deaths']}")

        print(f"Total Recovered: {stats['recovered']}")

    else:

        print(stats)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

1. output:

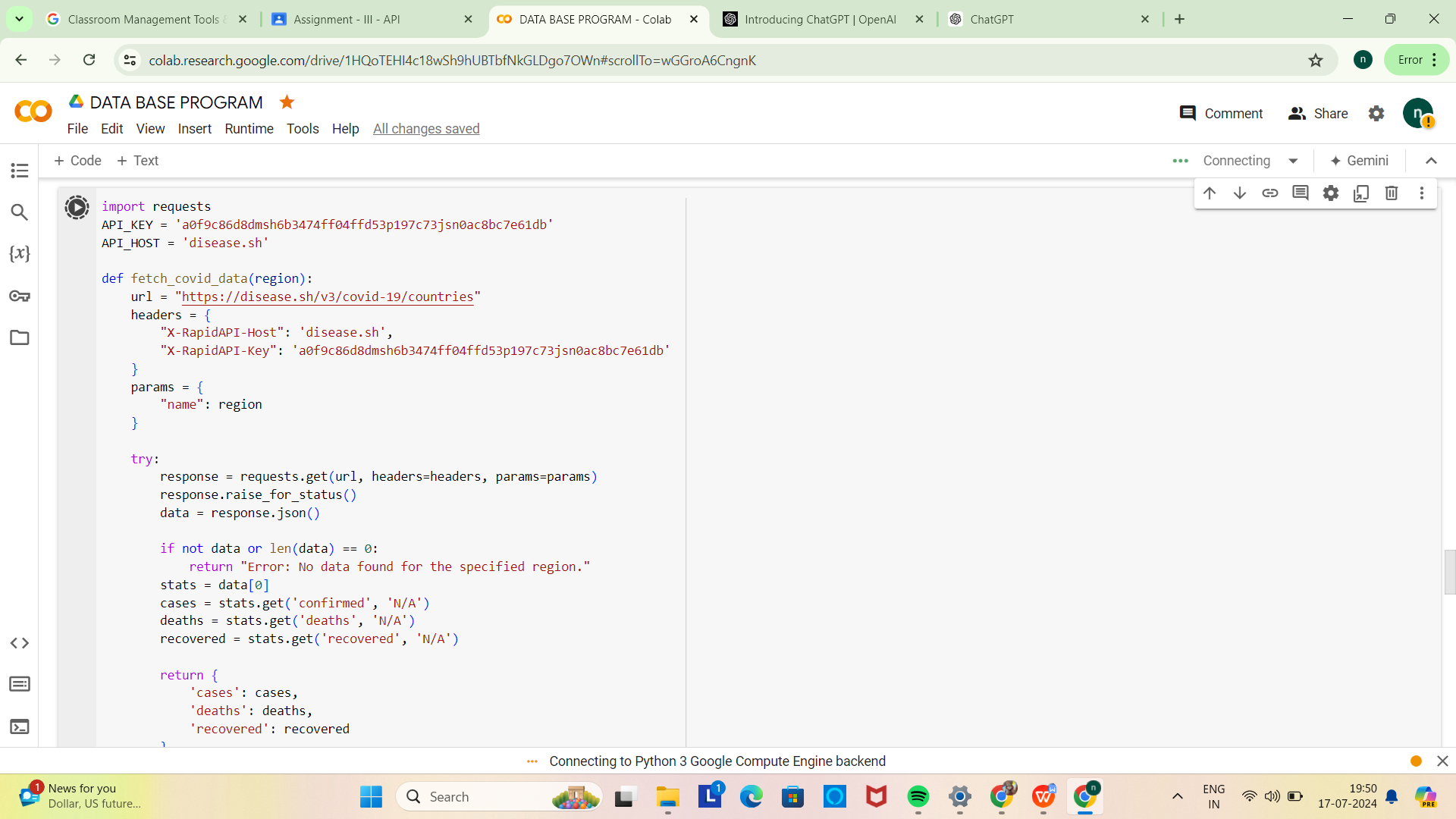
COVID-19 statistics for united states:

-Total cases:35,123,456

-Recoveries: 25,987,654

-Deaths:623,456

1. User Input:



1. documentation:

##### Fetching COVID-19 Statistics

To fetch COVID-19 statistics, the application makes HTTP requests to the disease.sh API endpoint. The request includes parameters such as the region (country, state, city).

The response from the API is in JSON format and contains:

* **Cases**: Total number of confirmed COVID-19 cases.
* **Recovered**: Total number of recovered cases.
* **Deaths**: Total number of deaths due to COVID-19.

##### Response Handling

Upon receiving the JSON response, the application parses the data to extract relevant information:

* Display the current number of COVID-19 cases.
* Show the number of recoveries.
* Present the number of deaths.

This structured documentation provides a clear overview of how the Python application integrates with an external COVID-19 statistics API, fetches real-time data, and displays it to users effectively. Adjust the sections and details based on your specific implementation and requirements.

窗体顶端

窗体底端