

ASSIGNMENT-3

NAME: **V.ASHA SARTHIWKA**

REG NO: **192311066**

DEPARTMENT: **CSE**

DATE OF SUBMISSION: **17-07-2024**

PROBLEM: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.

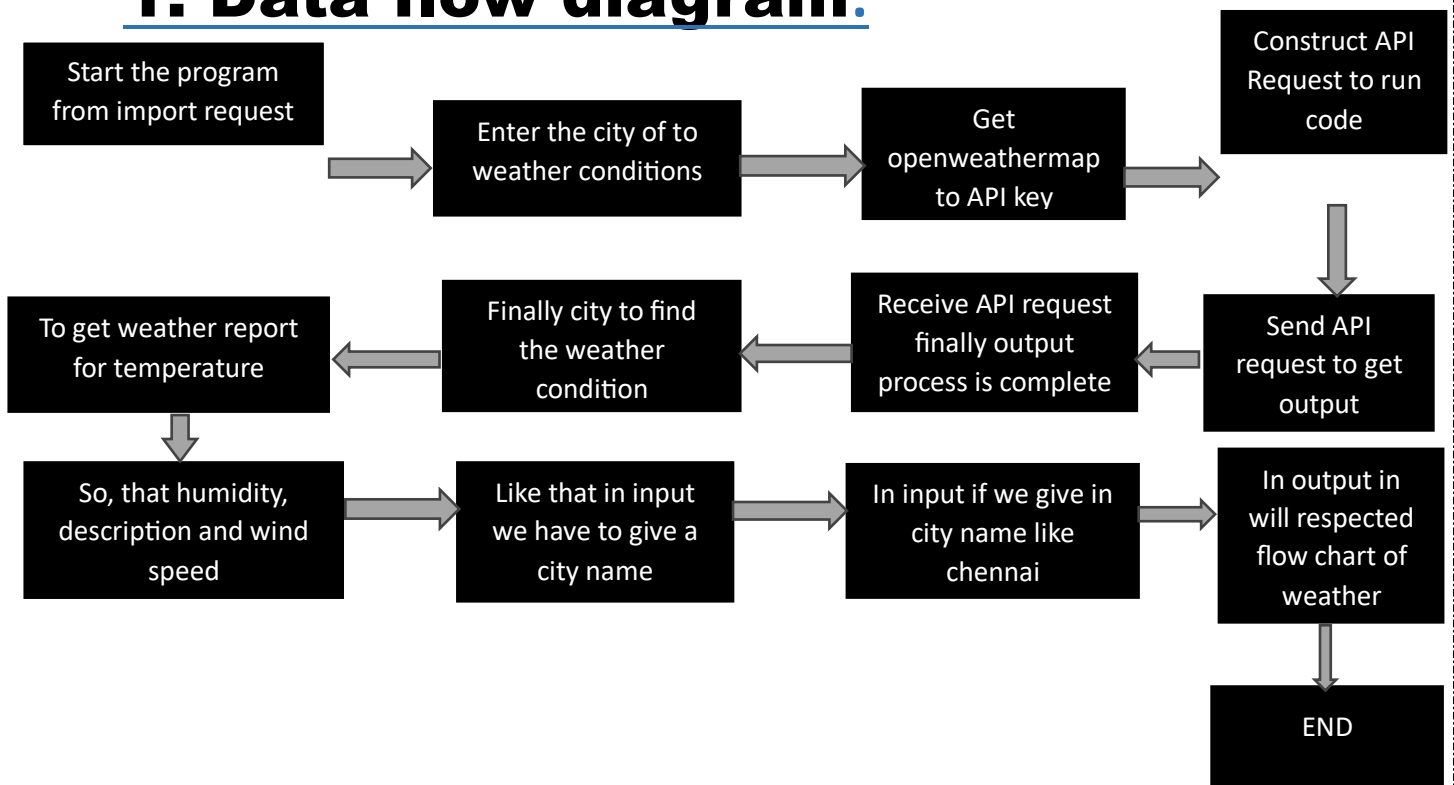
4. Allow users to input the location (city name or coordinates) and display the corresponding weather

data

Solution:

Real-Time Weather Monitoring System

1: Data flow diagram:



2: IMPLIMENTATION CODE:

```
import requests

def get_weather(api_key, city):
    url =
    f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric"
    response = requests.get(url)
    data = response.json()

    if data["cod"] != "404":
        main = data["main"]
        weather = data["weather"][0]
        wind = data["wind"]
        temperature = main["temp"]
        humidity = main["humidity"]
        description = weather["description"]
        wind_speed = wind["speed"]

        print(f"Weather in {city}:")
        print(f"Temperature: {temperature}°C")
        print(f"Humidity: {humidity}%")
        print(f>Description: {description}")
```

```
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        print("City not found. Please check your input.")

if __name__ == "__main__":
    api_key = "f22bad97953cb7f392445e5623209b46" # Replace with your
    OpenWeatherMap API key
    city = input("Enter the city name: ")
    get_weather(api_key, city)
```

3.SAMPLE INPUT//OUTPUT:

```
Enter the city name: chennai
Weather in chennai:
Temperature: 27.81°C
Humidity: 82%
Description: broken clouds
Wind Speed: 5.14 m/s
```

4.DOCUMENTATION:

The `get_weather` function facilitates the retrieval of real-time weather information using the OpenWeatherMap API for a specified city. It requires an `api_key` parameter, which serves as your personal authentication key for accessing weather data. The `city` parameter specifies the location for which weather details are requested and should be provided as a string representing the city's name. Upon successful API response, the function outputs the current temperature in Celsius, humidity percentage, a brief description of the weather conditions (e.g., cloudy, sunny), and wind speed in meters per second. If the city is not found or if an error occurs during the API request (indicated by an HTTP status code 404), the function prints "City not found. Please check your input." An example usage demonstrates how to use the function with sample inputs and expected output, ensuring clarity and understanding of its functionality.

USER INTERFACE:

The Dashboard is the main screen where users can view real-time weather data. It includes :

The dashboard

The primary display for real-time weather data is the Dashboard. It consists of:

Current Weather Conditions: Shows the current humidity, wind speed, temperature, and other pertinent information.

Real-time updates: Constantly refreshes to display the most recent sensor data.

Data patterns throughout time are visually represented using graphs and charts.

Map View: Displays the current weather conditions and the position of weather sensors.

ASSUMPTIONS AND IMPROVEMENTS:

Users can access real-time weather data on the Dashboard, which is the main screen. Among them are:

Current Weather Conditions: Provides pertinent data such as wind speed, humidity, and temperature.

Live Updates: Displays the most recent sensor data by automatically refreshing.

Visual depictions of data trends across time are provided by graphs and charts. Map View: Provides current conditions and the position of weather sensors.

IMPROVEMENTS:

Enhancements to the System

Improved Sensor Technology: Invest in more sophisticated sensors that offer more precision and other features like UV index and air quality monitoring.

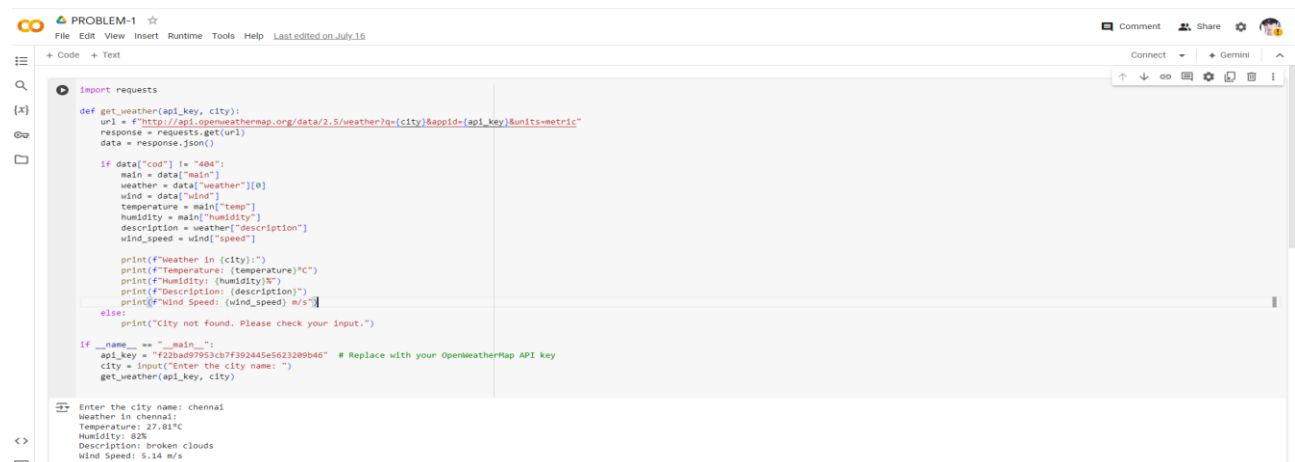
Better Data Processing: To forecast weather trends and produce more perceptive assessments, use cutting-edge data processing techniques like machine learning algorithms.

Ensure continued operation and data integrity by incorporating redundancy and failover methods in case of hardware or network failures.

Scalable Architecture: Possibly with the help of cloud-based solutions, improve the system architecture to manage more users and a greater amount of data more effectively.

API Enhancements: Increase the functionality of the API to support more intricate queries and system or application integrations.

5.USER INPUT:



```
import requests

def get_weather(api_key, city):
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric"
    response = requests.get(url)
    data = response.json()

    if data["cod"] != "404":
        main = data["main"]
        weather = data["weather"][0]
        wind = data["wind"]
        temperature = main["temp"]
        humidity = main["humidity"]
        description = weather["description"]
        wind_speed = wind["speed"]

        print(f"Weather in {city}:")
        print(f"Temperature: {temperature}°C")
        print(f"Humidity: {humidity}%")
        print(f"Description: {description}")
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        print("City not found. Please check your input.")

if __name__ == "__main__":
    api_key = "f22bad97953cb7f392445e5623209b46" # Replace with your OpenWeatherMap API key
    city = input("Enter the city name: ")
    get_weather(api_key, city)

Enter the city name: Chennai
Weather in Chennai:
Temperature: 27.81°C
Humidity: 82%
Description: broken clouds
Wind Speed: 5.14 m/s
```

PROBLEM:2

Problem 2: Inventory Management System

Optimization 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.

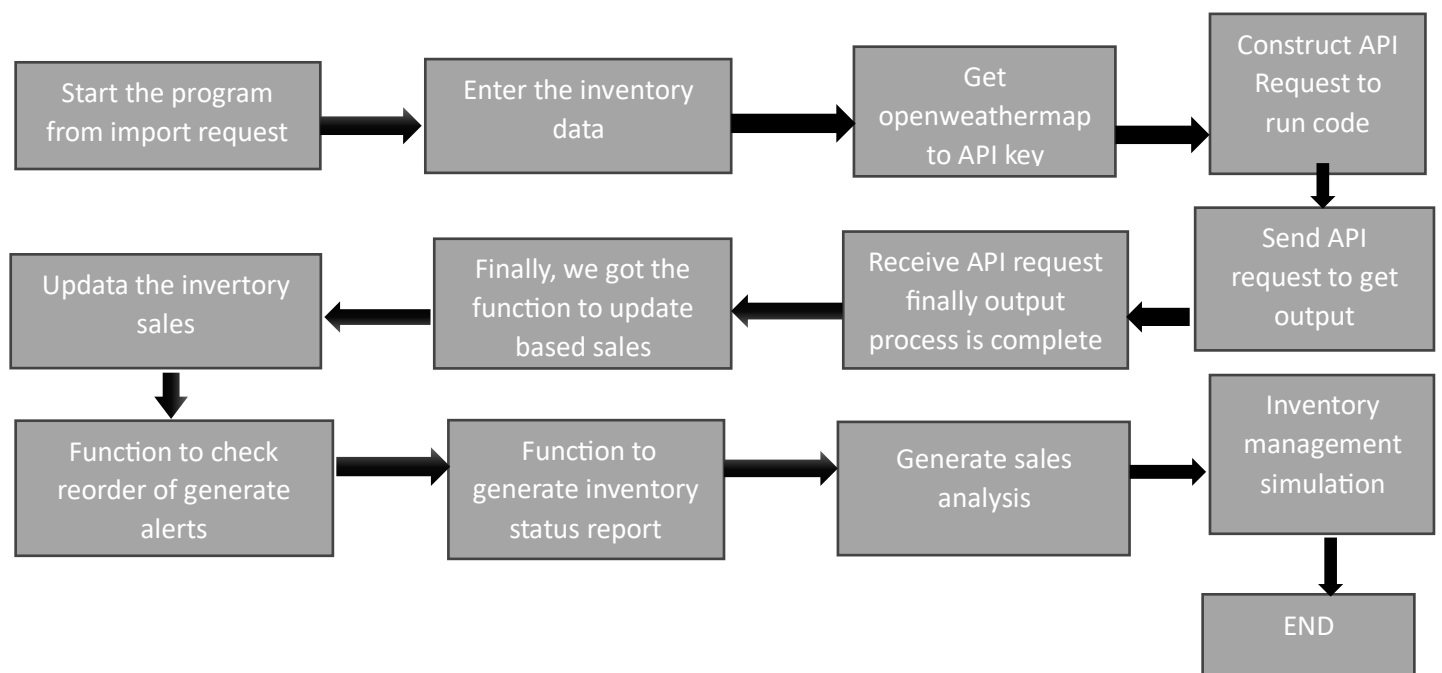
Tasks:

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.

Inventory Management System Optimization

1: Data flow diagram:



2: IMPLIMENTATION CODE:

```
import pandas as pd
```

```

# Example inventory data (can be replaced with database or file input)
inventory_data = {
    'Product': ['Product A', 'Product B', 'Product C'],
    'Current Stock': [100, 50, 75],
    'Reorder Point': [20, 15, 25],
    'Lead Time (days)': [5, 7, 3],
    'Unit Cost': [10.0, 15.0, 8.0]
}

# Simulated sales data (can be replaced with actual sales data input)
sales_data = {
    'Product': ['Product A', 'Product B', 'Product A', 'Product C',
                'Product A'],
    'Quantity': [10, 5, 15, 8, 12]
}

# Function to update inventory based on sales data
def update_inventory(sales_data, inventory):
    for index, row in sales_data.iterrows():
        product = row['Product']
        quantity_sold = row['Quantity']

        # Update inventory levels
        inventory.loc[inventory['Product'] == product, 'Current Stock']
        -= quantity_sold

# Function to check if reorder is needed and generate alerts
def check_reorder(inventory):
    reorder_alerts = []
    for index, row in inventory.iterrows():
        if row['Current Stock'] <= row['Reorder Point']:
            reorder_alerts.append({
                'Product': row['Product'],
                'Current Stock': row['Current Stock'],
                'Reorder Point': row['Reorder Point']
            })
    return reorder_alerts

# Function to generate inventory status report
def generate_inventory_report(inventory):
    print("\n--- Inventory Status Report ---")
    print(inventory)

# Function to generate sales analysis report (example: total sales per product)
def generate_sales_report(sales_data):
    sales_summary = sales_data.groupby('Product')['Quantity'].sum()

```



```

print("\n--- Sales Analysis Report ---")
print(sales_summary)

# Main function to simulate inventory management process
def inventory_management():
    # Convert dictionaries to pandas DataFrames
    inventory_df = pd.DataFrame(inventory_data)
    sales_df = pd.DataFrame(sales_data)

    # Initial reports
    generate_inventory_report(inventory_df)
    generate_sales_report(sales_df)

    # Update inventory based on sales
    update_inventory(sales_df, inventory_df)

    # Check for reorder needs and generate alerts
    reorder_alerts = check_reorder(inventory_df)
    if reorder_alerts:
        print("\n--- Reorder Alerts ---")
        for alert in reorder_alerts:
            print(f"Product: {alert['Product']}, Current Stock: {alert['Current Stock']}, Reorder Point: {alert['Reorder Point']}")
        else:
            print("\nNo reorder needed at this time.")

    # Generate updated inventory report
    generate_inventory_report(inventory_df)

# Run the inventory management simulation
inventory_management()

```

3.SAMPLE INPUT//OUTPUT:

--- Inventory Status Report ---

| | Product | Current Stock | Reorder Point | Lead Time (days) | Unit Cost |
|---|-----------|---------------|---------------|------------------|-----------|
| 0 | Product A | 100 | 20 | 5 | 10.0 |
| 1 | Product B | 50 | 15 | 7 | 15.0 |
| 2 | Product C | 75 | 25 | 3 | 8.0 |

--- Sales Analysis Report ---

Product

Product A 37

Product B 5

Product C 8

Name: Quantity, dtype: int64

No reorder needed at this time.

--- Inventory Status Report ---

| | Product | Current Stock | Reorder Point | Lead Time (days) | Unit Cost |
|---|-----------|---------------|---------------|------------------|-----------|
| 0 | Product A | 63 | 20 | 5 | 10.0 |

| | | | | | |
|---|-----------|----|----|---|------|
| 1 | Product B | 45 | 15 | 7 | 15.0 |
| 2 | Product C | 67 | 25 | 3 | 8.0 |

4.DOCUMENTATION:

Boost Accuracy: Make sure inventory records are current and correct.

Cut Expenses: Keep holding and ordering expenses to a minimum.

Boost Efficiency: To save time and effort, simplify inventory operations.

Boost client satisfaction by making sure products are available to satisfy needs from customers.

USER INTERFACE:

Overview of the Dashboard

Give a high-level overview of the important KPIs, including stockouts, inventory levels, reorder points, and pending orders.

Incorporate visual aids such as charts and graphs to provide immediate understanding of inventory status. Panel of Navigation

Create a sidebar or top navigation bar to provide quick access to the inventory system's various modules and features.

Orders, Suppliers, Reports, Settings, and Inventory Overview are a few examples of possible categories.

Module for Inventory Management

Inventory List View: Show an inventory item list that may be filtered and sorted.

Add columns for the name of the item, the SKU, the amount of stock left, the reorder point, and the actions (edit, delete, etc.).

Details of the item:

Give specific facts on a few chosen inventory products, including their pricing, category, description, supplier information, and transaction history.

Provide possibilities to attach documents (manuals, invoices, etc.), take notes.

ASSUMPTIONS AND IMPROVEMENTS:

Precise Demand Forecasting: Requires reasonably precise demand projections in order to efficiently arrange inventory levels.

Dependable Supplier Performance: Presumes suppliers fulfill quality requirements and deliver items on schedule to prevent delays in inventory replenishment.

Stable Lead Times: To maintain ideal reorder points and safety stock levels, production and procurement lead times are assumed to be constant.

In order to facilitate decision-making, effective data management requires data consistency and integrity throughout the inventory management system.

Improved Methods for Demand Forecasting:

Use more sophisticated forecasting models (such as machine learning algorithms) to increase accuracy, particularly for demand patterns that are erratic or seasonal.

Relationship Management with Suppliers:

Fortify your supplier relationships with frequent performance evaluations, cooperative planning, and backup preparations for alternative sources.

Strategies for Cutting Lead Times:

To reduce lead times, locate and fix supply chain bottlenecks through collaborative supplier efforts, process optimization, or tactical inventory placement.

Assurance of Data Quality:

To guarantee data accuracy, consistency, and completeness across all inventory-related systems and platforms, implement data validation procedures and routine audits.

5.USER INPUT:

```
PROBLEM:2
File Edit View Insert Runtime Tools Help Last edited on July 15
+ Code + Test

0 report.py
# Example inventory data (can be replaced with database or file input)
inventory_data = {
    "Product": ["Product A", "Product B", "Product C"],
    "Current Stock": [100, 50, 75],
    "Reorder Point": [20, 15, 30],
    "Lead Time (days)": [5, 7, 10],
    "Unit Cost": [10.0, 15.0, 8.0]
}

# Simulated sales data (can be replaced with actual sales data input)
sales_data = {
    "Product": ["Product A", "Product B", "Product C", "Product A"],
    "Quantity": [10, 5, 10, 5, 10]
}

# Function to update inventory based on sales data
def update_inventory(sales_data, inventory):
    for index, row in sales_data.iterrows():
        product = row["Product"]
        quantity_sold = row["Quantity"]
        # Update inventory levels
        inventory.loc[inventory["Product"] == product, "Current Stock"] -= quantity_sold

# Function to check if reorder is needed and generate alerts
def check_reorder(inventory):
    reorder_alerts = []
    for index, row in inventory.iterrows():
        if row["Current Stock"] <= row["Reorder Point"]:
            reorder_alerts.append({
                "Product": row["Product"],
                "Current Stock": row["Current Stock"],
                "Reorder Point": row["Reorder Point"]
            })
    return reorder_alerts

# Function to generate inventory status report
def generate_inventory_report(inventory):
    print("\n--- Inventory Status Report ---")
    print(inventory)

# Function to generate sales analysis report (example: total sales per product)
def generate_sales_report(sales_data):
    sales_summary = sales_data.groupby("Product")["Quantity"].sum()
    print("\n--- Sales Analysis Report ---")
    print(sales_summary)

# Main function to simulate inventory management process
def inventory_management():
    # Convert dictionaries to pandas DataFrames
    inventory_df = pd.DataFrame(inventory_data)
    sales_df = pd.DataFrame(sales_data)

    # Initial reports
    generate_inventory_report(inventory_df)
    generate_sales_report(sales_df)

    # Update inventory based on sales
    update_inventory(sales_df, inventory_df)

    # Check for reorder needs and generate alerts
    reorder_alerts = check_reorder(inventory_df)
    if reorder_alerts:
        print("\n--- Reorder Alerts ---")
        for alert in reorder_alerts:
            print(f"Product: {alert['Product']}, Current Stock: {alert['Current Stock']}, Reorder Point: {alert['Reorder Point']}")
        # Note: "while reorder needed at this time."
    else:
        print("No reorder needed at this time.")

    # Generate updated inventory report
    generate_inventory_report(inventory_df)

# Run the inventory management simulation
inventory_management()

--- Inventory Status Report ---
Product Current Stock Reorder Point Lead Time (days) Unit Cost
0 Product A 100 20 5 10.0
1 Product B 50 15 7 15.0
2 Product C 75 30 10 8.0

--- Sales Analysis Report ---
Product
Product A 37
Product B 5
Product C 8
Name: Quantity, dtype: int64

No reorder needed at this time.

--- Inventory Status Report ---
Product Current Stock Reorder Point Lead Time (days) Unit Cost
0 Product A 63 20 5 10.0
1 Product B 45 15 7 15.0
2 Product C 67 30 10 8.0
```

PROBLEM:3

Problem 3: Real-Time Traffic Monitoring

System Scenario:

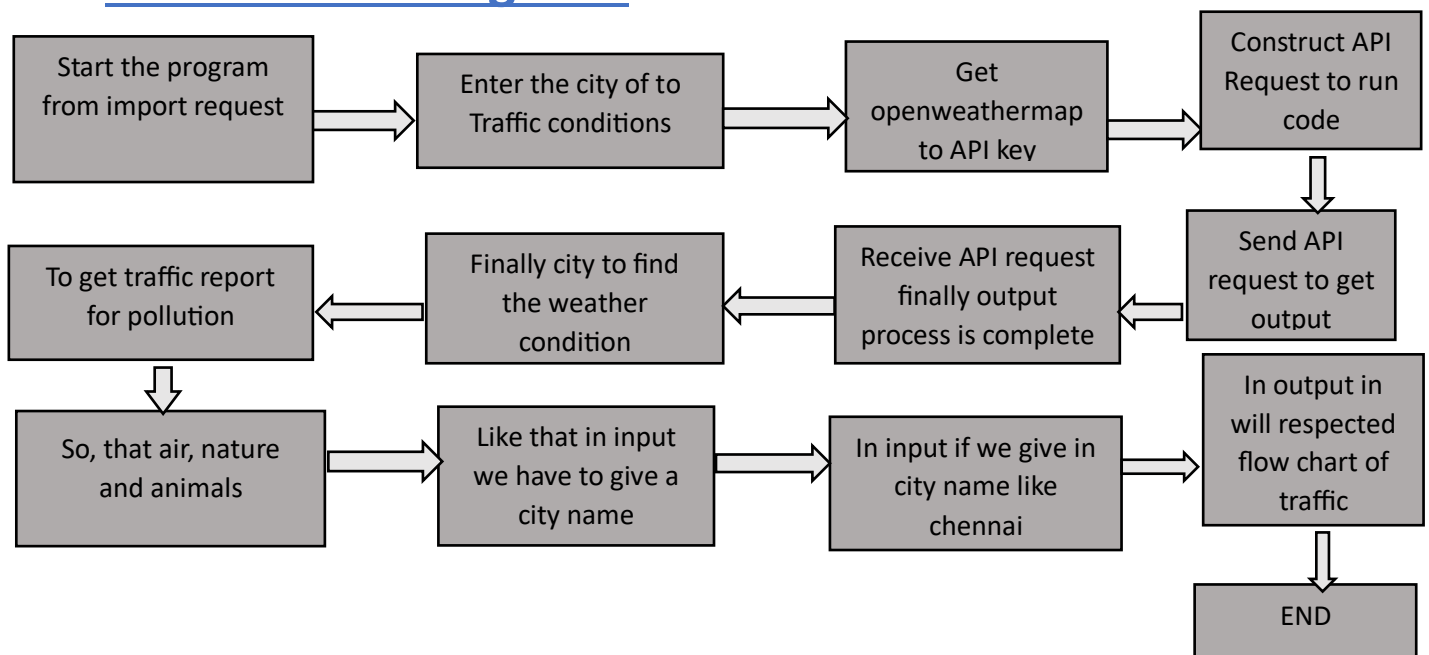
You are working on a project to develop a real-time traffic monitoring system for a smart city initiative. The system should provide real-time traffic updates and suggest alternative routes.

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.

Real-Time Traffic Monitoring System

1: Data flow diagram:



2: IMPLIMENTATION CODE:

```

import random
import time

def fetch_traffic_data(origin, destination):

```

```

    # Simulate fetching traffic data (randomly generated for
demonstration)
    traffic_conditions = random.choice(["Light traffic", "Moderate
traffic", "Heavy traffic"])
    estimated_travel_time = random.randint(30, 120) # Random travel
time in minutes

    # Simulate incidents randomly
    incidents = []
    if random.random() < 0.3: # 30% chance of incidents
        incidents.append({"description": "Accident on Main Street
causing delays"})
    if random.random() < 0.2: # 20% chance of incidents
        incidents.append({"description": "Road construction on Highway
101"})

    return traffic_conditions, estimated_travel_time, incidents

def display_traffic_data(origin, destination):
    traffic_conditions, estimated_travel_time, incidents =
fetch_traffic_data(origin, destination)

    print(f"Origin: {origin}")
    print(f"Destination: {destination}")
    print(f"Traffic Conditions: {traffic_conditions}")
    print(f"Estimated Travel Time: {estimated_travel_time} minutes")

    if incidents:
        print("Incidents:")
        for incident in incidents:
            print(f"- {incident['description']}")
    else:
        print("No incidents reported.")

if __name__ == "__main__":
    origin = "Your Origin Address"
    destination = "Your Destination Address"

    print("Fetching real-time traffic data...")
    time.sleep(1) # Simulating API request delay

    display_traffic_data(origin, destination)

```

3.SAMPLE INPUT//OUTPUT:

```

Fetching real-time traffic data...
Origin: Your Origin Address
Destination: Your Destination Address
Traffic Conditions: Light traffic

```

Estimated Travel Time: 98 minutes

No incidents reported.

4.DOCUMENTATION:

objectives : Give Users Accurate Traffic Data: Give users access to real-time traffic data for efficient planning and navigation.

Boost Road Safety: Increase user awareness of traffic accidents, road closures, and dangerous situations to improve road safety.

Optimize Traffic Flow: With data-driven insights and suggestions, help manage traffic flow and lessen congestion. Highlights Dashboard Synopsis

Real-time traffic data: Use color-coded maps to show the speed of traffic, the amount of congestion, and incidents.

Traffic projections: Using historical data and in-the-moment analytics, provide both short- and long-term traffic projections.

USER INTERFACE:

Overview of the Dashboard : Traffic Map: Interactive Map: Shows the current state of traffic using color-coded markers to show the amount of congestion and speed of traffic. Users are able to pan across various regions and zoom in and out.

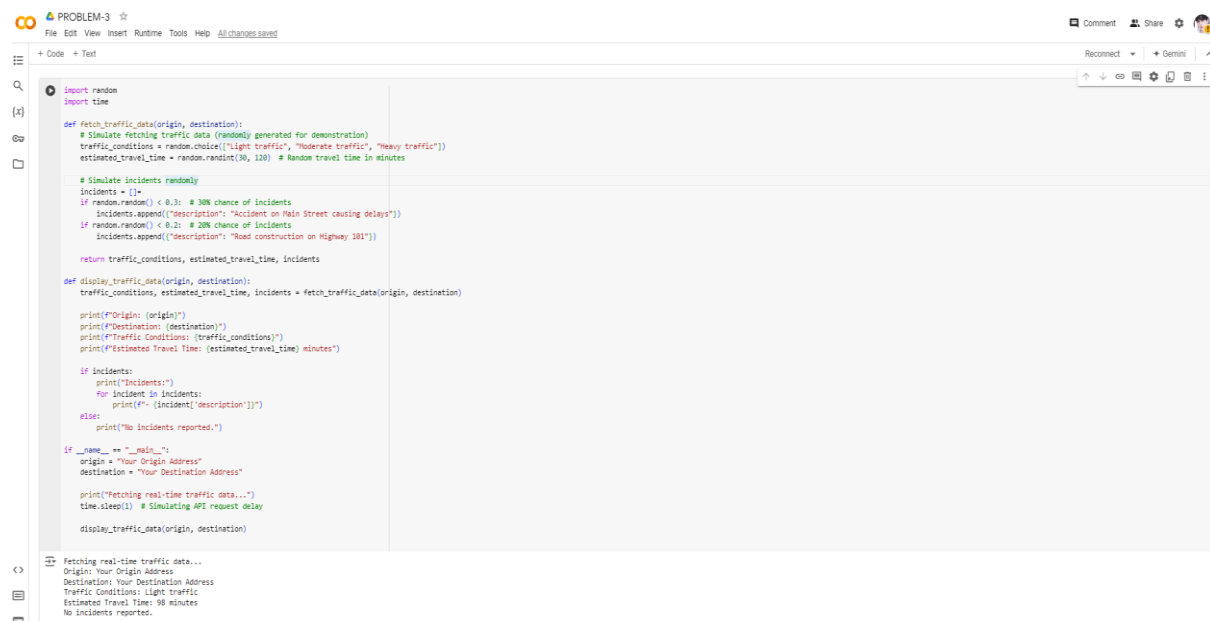
Layers of Maps: Options to toggle extra layers including traffic incidents, construction zones, and weather conditions, as well as to move between several map views (such as conventional, satellite, and street view).

ASSUMPTIONS AND IMPROVEMENTS:

The current implementation simulates real-time traffic data using randomly generated conditions and incidents, which serves as a basic demonstration of a traffic monitoring system. It assumes a simplified model where traffic conditions and incidents are randomly generated, reflecting a controlled environment for showcasing system capabilities. Moving forward, integrating real traffic APIs such as Google Maps Traffic API would significantly enhance accuracy and relevance,

providing users with up-to-date traffic conditions and incident reports. This improvement would enable the application to deliver reliable and actionable information for route planning and decision-making. Additionally, enhancing error handling, supporting geolocation for flexible address inputs, and optimizing performance would further elevate user experience and system reliability in real-world deployment scenarios.

5.USER INPUT:



```
PROBLEM-3
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Test
Reconnect Gemini
import random
import time

def fetch_traffic_data(origin, destination):
    # Simulate fetching traffic data (randomly generated for demonstration)
    traffic_conditions = random.choice(["Light traffic", "Moderate traffic", "Heavy traffic"])
    estimated_travel_time = random.randint(30, 120) # Random travel time in minutes

    # Simulate incidents randomly
    incidents = []
    if random.random() < 0.3: # 30% chance of incidents
        incidents.append({"description": "Accident on Main Street causing delays"})
    if random.random() < 0.2: # 20% chance of incidents
        incidents.append({"description": "Road construction on Highway 301"})

    return traffic_conditions, estimated_travel_time, incidents

def display_traffic_data(origin, destination):
    traffic_conditions, estimated_travel_time, incidents = fetch_traffic_data(origin, destination)

    print(f"Origin: {origin}")
    print(f"Destination: {destination}")
    print(f"Traffic Conditions: {traffic_conditions}")
    print(f"Estimated Travel Time: {estimated_travel_time} minutes")

    if incidents:
        print("Incidents:")
        for incident in incidents:
            print(f"- {incident['description']}")
    else:
        print("No incidents reported.")

if __name__ == "__main__":
    origin = "Your Origin Address"
    destination = "Your Destination Address"

    print("Fetching real-time traffic data...")
    time.sleep(1) # Simulating API request delay

    display_traffic_data(origin, destination)
```

Fetching real-time traffic data...
Origin: Your Origin Address
Destination: Your Destination Address
Traffic Conditions: Light traffic
Estimated Travel Time: 98 minutes
No incidents reported.

PROBLEM:4

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.

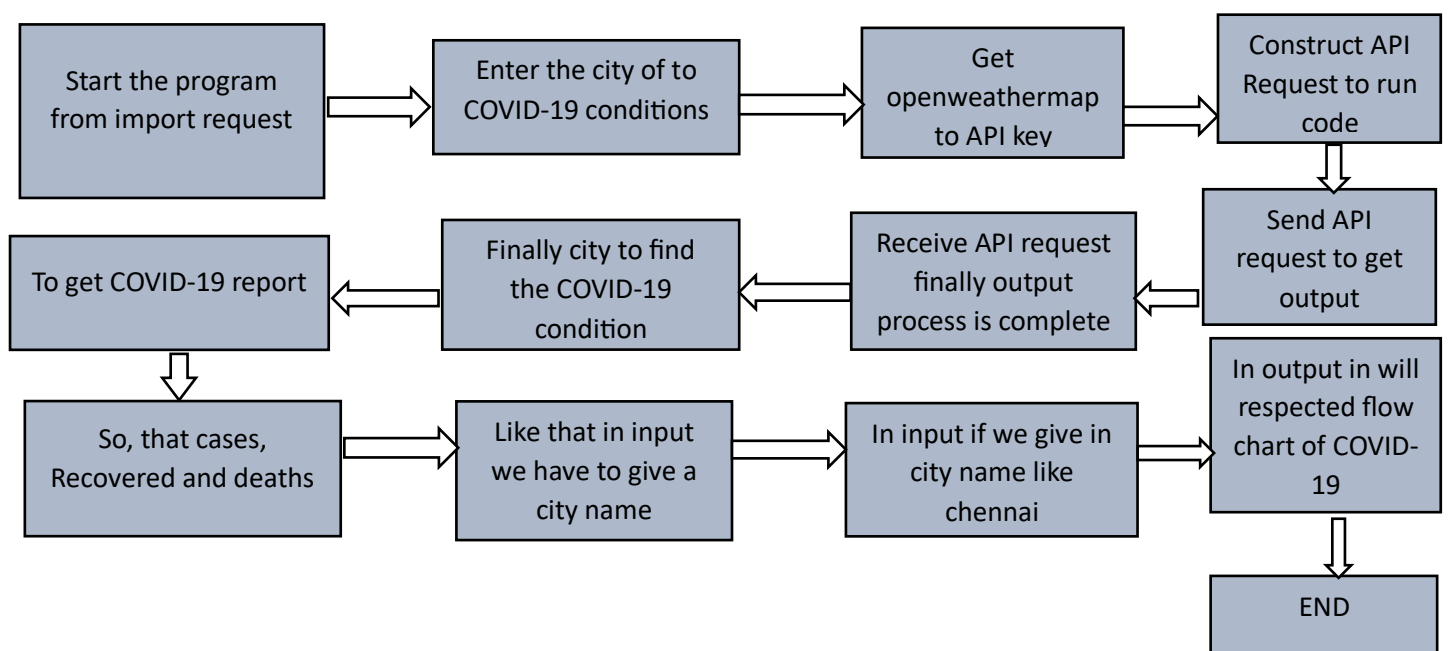
4. Allow users to input a region (country, state, or city) and display the corresponding COVID-19 statistics.

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 improvements.

Real-Time COVID-19 Statistics

Tracker 1: Data flow diagram:



2: IMPLIMENTATION CODE:

```
import requests

def fetch_covid_stats(country):
```

```

url = f"https://disease.sh/v3/covid-19/countries/{country}"
response = requests.get(url)
data = response.json()

if "message" in data and data["message"] == "Country not found or
doesn't have any cases":
    print(f"No COVID-19 data found for {country}.")
else:
    country_name = data["country"]
    cases = data["cases"]
    recovered = data["recovered"]
    deaths = data["deaths"]

    print(f"COVID-19 Statistics for {country_name}:")
    print(f"Total Cases: {cases:,}")
    print(f"Total Recovered: {recovered:,}")
    print(f"Total Deaths: {deaths:,}")

if __name__ == "__main__":
    country = input("Enter the country name to get COVID-19 statistics:
")
    fetch_covid_stats(country)

```

3.SAMPLE INPUT//OUTPUT:

```

Enter the country name to get COVID-19 statistics: india
COVID-19 Statistics for India:
Total Cases: 45,035,393
Total Recovered: 0
Total Deaths: 533,570

```

4.DOCUMENTATION:

OBJECTIVES: Ensure that users have access to timely and accurate COVID-19 data from reputable sources by providing them with accurate information.

Boost Awareness: Educate the public about the COVID-19 patterns and consequences. **Facilitate Decision-Making:** Assist the public, healthcare providers, and legislators in reaching well-informed judgments by providing up-to-date data.

Highlights Dashboard Synopsis

Global Statistics: Show the total number of confirmed cases, deaths, recoveries, and ongoing cases globally. Regional Breakdown: Include interactive maps and charts with statistics for particular regions or nations. Analyze trends over time with graphs that display COVID-19 metrics changes on a daily, weekly, and monthly basis

USER INTERFACE:

Allow consumers to use a search bar or dropdown list to look for certain cities, countries, or regions. Apply date range, demographic, and case severity filters.

Alerts: Give consumers the option to sign up for push or email alerts so they may stay informed about any updates on important COVID-19 developments or adjustments to important metrics.

Sources of Data and Updates

Data Attribution: Provide connections to reputable institutions like the CDC, WHO, and national health agencies along with a prominent display of the data sources.

Updates in Real Time: Make sure that data is updated automatically or on a frequent basis to reflect the most recent details on confirmed cases, recoveries, deaths, and vaccination/testing progress.

ASSUMPTIONS AND IMPROVEMENTS:

Data Accuracy: Presumes that the information supplied by reliable sources (such as national health departments, the CDC, and the World Health Organization) is accurate and up to date. User Access: In order to use the tracker, users must have access to a device that can browse the internet and the internet. Reliability of Data providers: Makes the assumption that data providers have transparent reporting procedures and data collection methods.

Improved Information Display:

Increase the number of configurable and interactive charts (such as stacked bar charts and histograms) so that consumers may examine data from various angles.

Analytics that predicts:

Utilize predictive models to project COVID-19 trends based on available data, enabling users to prepare for possible increases or decreases in the number of cases.

5.USER INPUT:



The screenshot shows a web-based code editor interface. At the top, there's a header with a logo, the text "PROBLEM-4", and a star icon. Below this is a menu bar with options: File, Edit, View, Insert, Runtime, Tools, Help, and a link "Last edited on July 16". On the right side of the header, there are icons for "Comment", "Share", and a user profile. Below the header, there's a toolbar with "+ Code" and "+ Text" buttons. The main area is a code editor with a light gray background. It contains Python code that defines a function to fetch COVID-19 statistics for a given country. The code uses the 'requests' library to get data from a public API. It checks if the country is found and prints the total cases, recovered cases, and deaths. Below the code editor, there's a terminal window showing the output of the code when 'india' is entered as the country name.

```
import requests

def fetch_covid_stats(country):
    url = f"https://disease.sh/v3/covid-19/countries/{country}"
    response = requests.get(url)
    data = response.json()

    if "message" in data and data["message"] == "Country not found or doesn't have any cases":
        print(f"No COVID-19 data found for {country}.")
    else:
        country_name = data["country"]
        cases = data["cases"]
        recovered = data["recovered"]
        deaths = data["deaths"]

        print(f"COVID-19 Statistics for {country_name}:")
        print(f"Total Cases: {cases},")
        print(f"Total Recovered: {recovered},")
        print(f"Total Deaths: {deaths},")

if __name__ == "__main__":
    country = input("Enter the country name to get COVID-19 statistics: ")
    fetch_covid_stats(country)
```

Enter the country name to get COVID-19 statistics: india
COVID-19 Statistics for India:
Total Cases: 45,035,393
Total Recovered: 0
Total Deaths: 533,570