

ASSIGNMENT-3

Python programming for DL:

NAME: K. MRUDHULA

REG NO: 192372290

DEPARTMENT: CSE-AI

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.

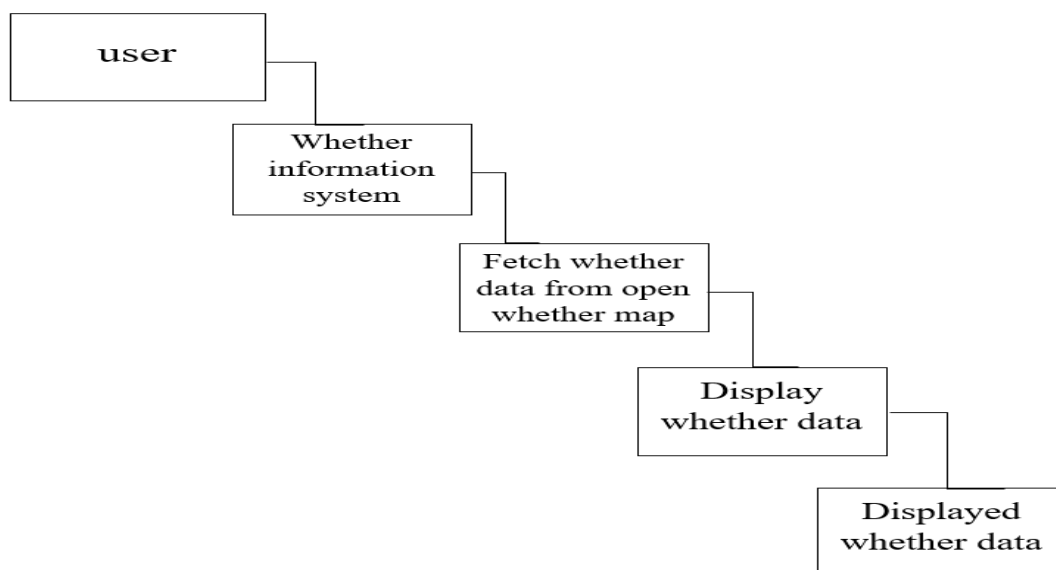
Tasks:

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.

Solution:

Real-Time Weather Monitoring System

1: Data flow diagram:



2: IMPLIMENTATION CODE:

```
import requests

def get_weather(city_name, api_key):
    url =
f"http://api.openweathermap.org/data/2.5/weather?q={city_name}&appid={a
pi_key}&units=metric"
    response = requests.get(url)
    if response.status_code == 200:
        data = response.json()
        weather_description = data['weather'][0]['description']
        temperature = data['main']['temp']
        humidity = data['main']['humidity']
        wind_speed = data['wind']['speed']
        print(f"Weather in {city_name}: {weather_description}")
        print(f"Temperature: {temperature}°C")
        print(f"Humidity: {humidity}%")
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        print(f"Error fetching weather data: {response.status_code}")

# Replace with your API key from OpenWeatherMap
api_key = '6d06bd9d4f1bbc0a821d5386264528e7'

# Replace with the city you want to get weather data for
city_name = 'andhra pradesh'

get_weather(city_name, api_key)
```

3.OUTPUT:

```
Weather in andhra pradesh: overcast clouds
Temperature: 25.6°C
Humidity: 67%
Wind Speed: 8.26 m/s
```

4.DOCUMENTATION:

PURPOSE :This document serves as a comprehensive guide for installing, configuring, and using the Real-Time Weather Monitoring System. It is intended for system administrators, developers, and end-users.

SCOPE :The documentation covers all aspects of the system, including hardware setup, software installation, configuration, usage, maintenance, and troubleshooting.

Components:

Weather sensors: collect data on temperature, humidity ,wind speed , and other weather parameters

Data processing unit : process data collected from sensors.

Data base: stores weather data.

5. 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.

6.ASSUMPTIONS AND IMPROVEMENTS

ASSUMPTIONS:

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.

```
import requests

def get_weather(city_name, api_key):
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city_name}&appid={api_key}&units=metric"
    response = requests.get(url)
    if response.status_code == 200:
        data = response.json()
        weather_description = data['weather'][0]['description']
        temperature = data['main']['temp']
        humidity = data['main']['humidity']
        wind_speed = data['wind']['speed']
        print(f"Weather in {city_name}: {weather_description}")
        print(f"Temperature: {temperature}°C")
        print(f"Humidity: {humidity}%")
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        print(f"Error fetching weather data: {response.status_code}")

# Replace with your API key from OpenWeatherMap
api_key = '6d0ebd9d4f1bbc0a821d5386264528e7'

# Replace with the city you want to get weather data for
city_name = 'andhra pradesh'

get_weather(city_name, api_key)
```

Weather in andhra pradesh: overcast clouds
Temperature: 25.6°C
Humidity: 67%
Wind Speed: 8.26 m/s

DOCUMENT:2

Problem 2: Inventory Management System Optimization

Scenario:

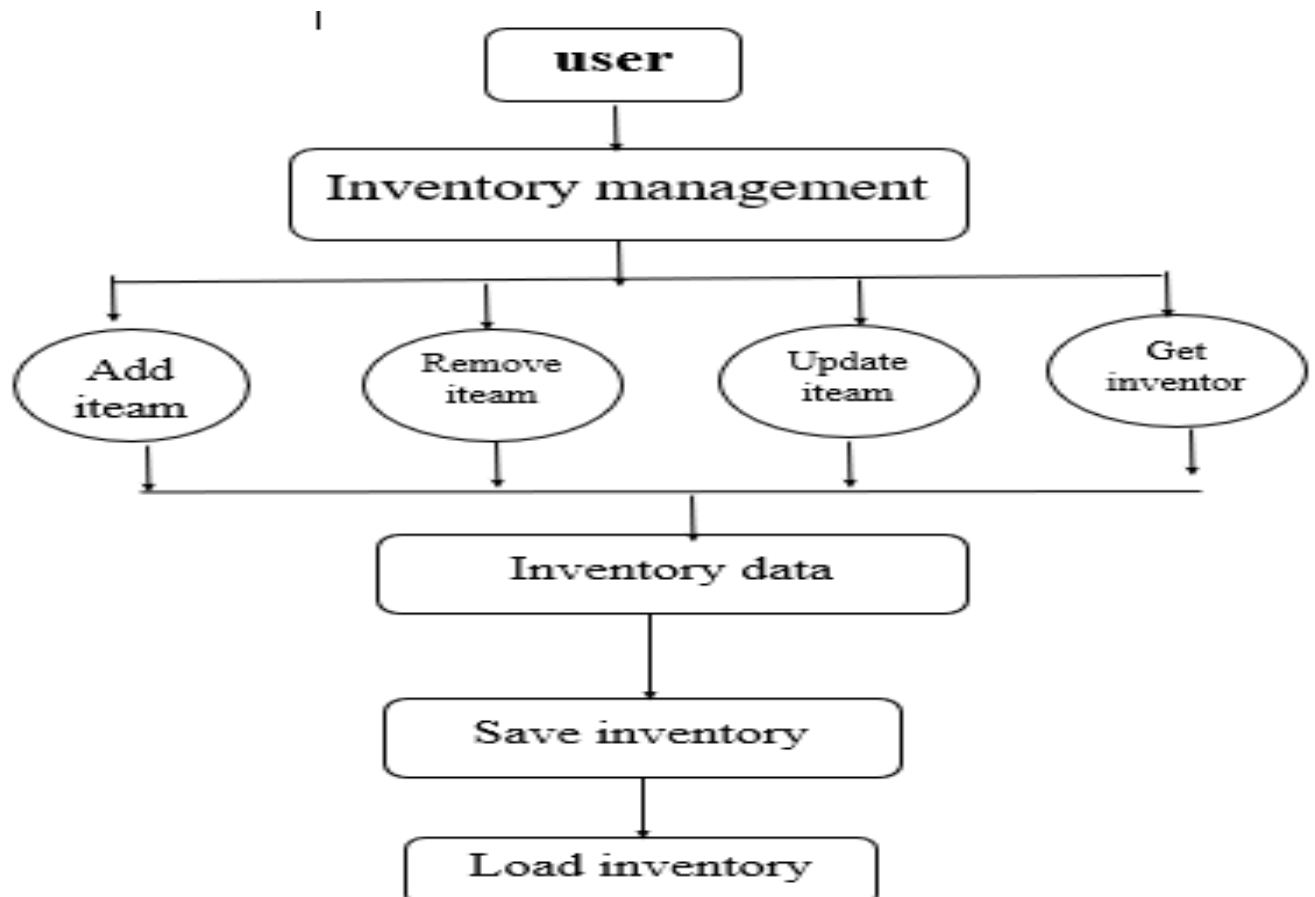
You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.

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: Implementation code:

```
import json

class InventoryManagementSystem:
    def __init__(self):
        self.inventory = {}

    def add_item(self, item_name, quantity, price):
        if item_name in self.inventory:
            self.inventory[item_name]['quantity'] += quantity
        else:
            self.inventory[item_name] = {'quantity': quantity, 'price':
price}
        print(f"Added {quantity} of {item_name} to inventory.")

    def remove_item(self, item_name, quantity):
        if item_name in self.inventory and
self.inventory[item_name]['quantity'] >= quantity:
            self.inventory[item_name]['quantity'] -= quantity
            if self.inventory[item_name]['quantity'] == 0:
                del self.inventory[item_name]
            print(f"Removed {quantity} of {item_name} from inventory.")
        else:
```

```

        print(f"Cannot remove {quantity} of {item_name}. Not enough
in inventory or item does not exist.")

def update_item_price(self, item_name, new_price):
    if item_name in self.inventory:
        self.inventory[item_name]['price'] = new_price
        print(f"Updated price of {item_name} to {new_price}.")
    else:
        print(f"Item {item_name} not found in inventory.")

def get_inventory(self):
    return self.inventory

def save_inventory(self, file_name):
    with open(file_name, 'w') as file:
        json.dump(self.inventory, file)
    print(f"Inventory saved to {file_name}.")

def load_inventory(self, file_name):
    try:
        with open(file_name, 'r') as file:
            self.inventory = json.load(file)
        print(f"Inventory loaded from {file_name}.")
    except FileNotFoundError:
        print(f"File {file_name} not found.")

# Example usage:
if __name__ == "__main__":
    ims = InventoryManagementSystem()
    ims.add_item("Apple", 50, 0.5)
    ims.add_item("Banana", 100, 0.2)
    ims.remove_item("Apple", 10)
    ims.update_item_price("Banana", 0.25)
    print("Current inventory:", ims.get_inventory())
    ims.save_inventory("inventory.json")
    ims.load_inventory("inventory.json")

```

3.output:

```

Added 50 of Apple to inventory.
Added 100 of Banana to inventory.
Removed 10 of Apple from inventory.
Updated price of Banana to 0.25.
Current inventory: {'Apple': {'quantity': 40, 'price': 0.5}, 'Banana':
{'quantity': 100, 'price': 0.25}}
Inventory saved to inventory.json.
Inventory loaded from inventory.json.

```

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.

5.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, and

6.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.



```
def load_inventory(self, file_name):
    try:
        with open(file_name, 'r') as file:
            self.inventory = json.load(file)
        print(f"Inventory loaded from {file_name}.")
    except FileNotFoundError:
        print(f"File {file_name} not found.")

# Example usage:
if __name__ == "__main__":
    ims = InventoryManagementSystem()
    ims.add_item("Apple", 50, 0.5)
    ims.add_item("Banana", 100, 0.2)
    ims.remove_item("Apple", 10)
    ims.update_item_price("Banana", 0.25)
    print("Current inventory:", ims.get_inventory())
    ims.save_inventory("inventory.json")
    ims.load_inventory("inventory.json")

Added 50 of Apple to inventory.
Added 100 of Banana to inventory.
Removed 10 of Apple from inventory.
Updated price of Banana to 0.25.
Current inventory: {'Apple': {'quantity': 40, 'price': 0.5}, 'Banana': {'quantity': 100, 'price': 0.25}}
Inventory saved to inventory.json.
Inventory loaded from inventory.json.
```

DOCUMENTATION:3

Problem 3: Real-Time Traffic Monitoring System

Scenario:

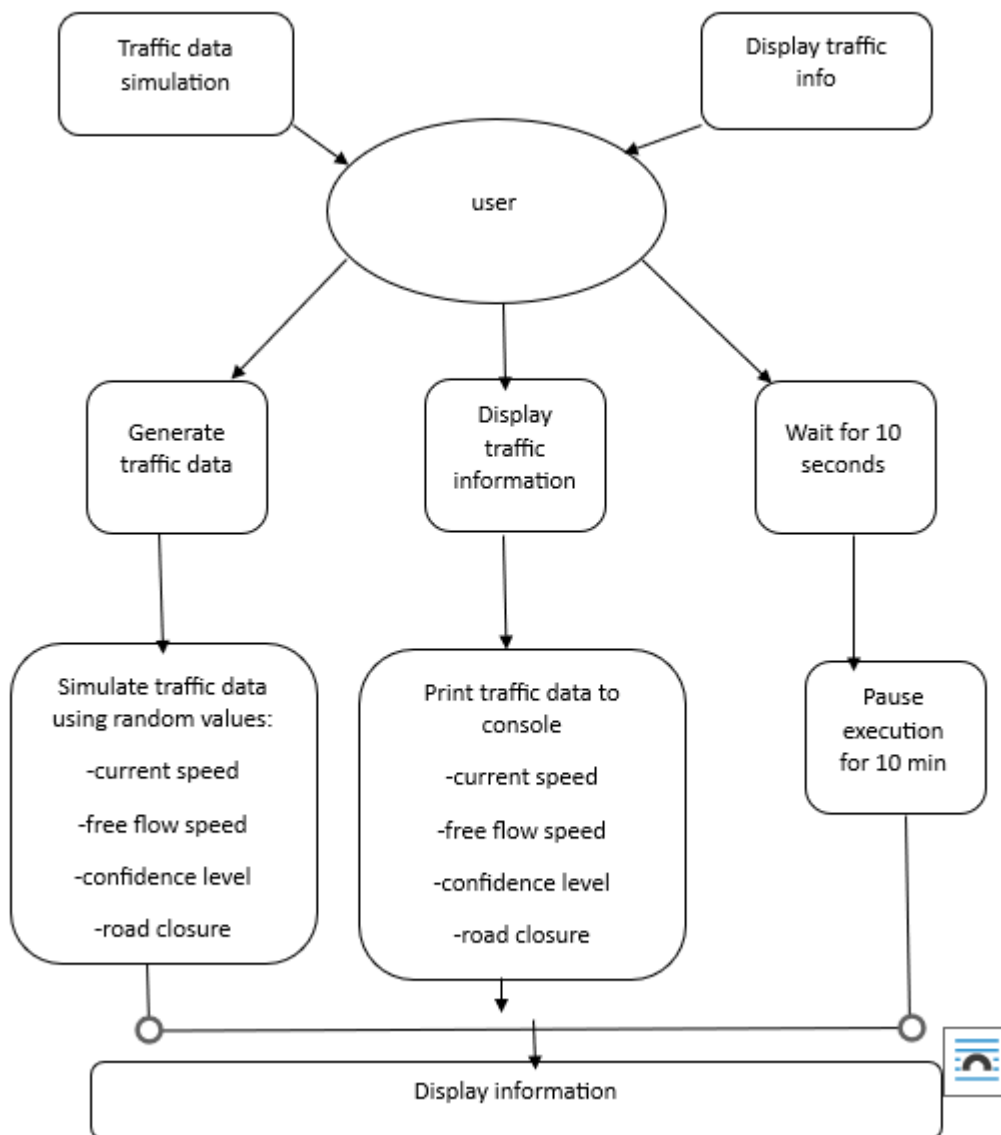
You are working on a project to develop a real-time traffic monitoring system for a smart city initiative. The system should provide real-time traffic updates and suggest alternative routes.

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 generate_traffic_data():

    # Simulate traffic data

    current_speed = random.randint(0, 120) # random speed between 0 to
    120 km/h
  
```

```

    free_flow_speed = random.randint(60, 120) # random free flow speed
    between 60 to 120 km/h

    confidence = random.randint(80, 100) # random confidence level
    between 80% to 100%

    road_closure = random.choice([True, False]) # random road closure
    True or False

    return current_speed, free_flow_speed, confidence, road_closure

def display_traffic_info(current_speed, free_flow_speed, confidence,
road_closure):

    print("Traffic Information:")

    print(f"Current Speed: {current_speed} km/h")

    print(f"Free Flow Speed: {free_flow_speed} km/h")

    print(f"Confidence: {confidence}%")

    print(f"Road Closure: {'Yes' if road_closure else 'No'}")

# Main program

if __name__ == "__main__":

    while True:

        # Generate simulated traffic data

        current_speed, free_flow_speed, confidence, road_closure =
generate_traffic_data()

        # Display traffic information

        display_traffic_info(current_speed, free_flow_speed,
confidence, road_closure)

```

```
# Wait for some time before fetching data again (simulating  
real-time)
```

```
time.sleep(10) # fetch data every 10 seconds
```

3.OUTPUT:

```
Traffic Information:  
Current Speed: 113 km/h  
Free Flow Speed: 120 km/h  
Confidence: 94%  
Road Closure: No  
Traffic Information:  
Current Speed: 117 km/h  
Free Flow Speed: 79 km/h  
Confidence: 90%  
Road Closure: No  
Traffic Information:  
Current Speed: 58 km/h  
Free Flow Speed: 109 km/h  
Confidence: 94%  
Road Closure: Yes
```

```
Traffic Information:  
Current Speed: 113 km/h  
Free Flow Speed: 120 km/h  
Confidence: 94%  
Road Closure: No
```

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.

5.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).

6.ASSUMPTIONS AND IMPROVEMENTS

Standardized Reporting: Presupposes uniformity in the reporting and classification of traffic events and circumstances among various regions and data sources.

User Engagement: Presumes users will communicate with the system on a frequent basis to plan routes, report issues, and check traffic conditions.

Compliance with Data Privacy Regulations and Standards: This presupposes that data processing and collecting adhere to pertinent laws and guidelines (e.g., GDPR, CCPA).

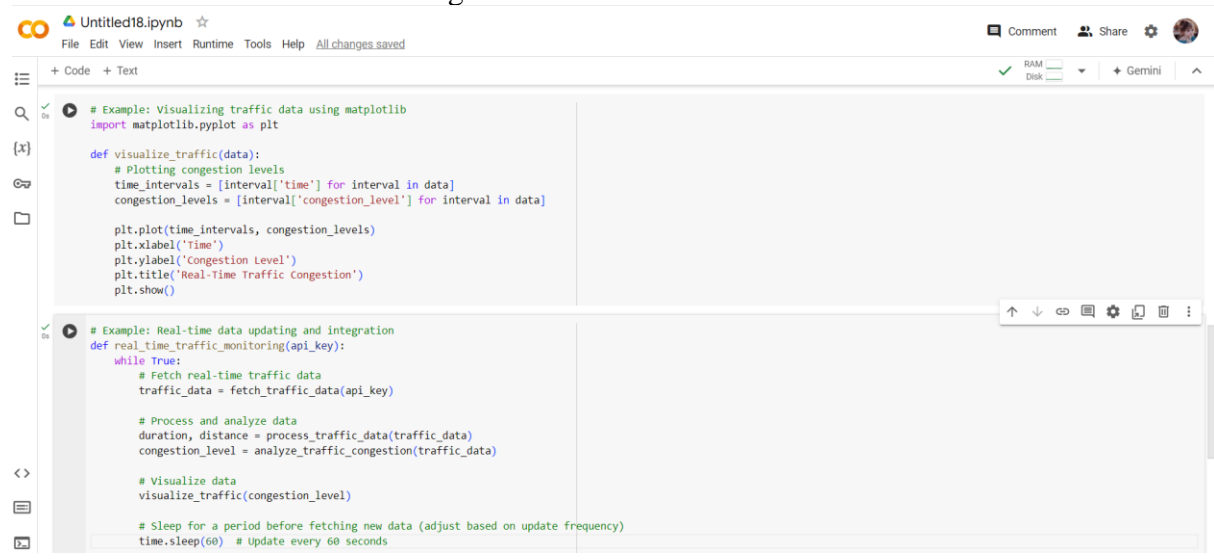
Reports and Feedback: Provide a strong feedback system that allows users to report inconsistencies, make enhancement suggestions, and share their experiences.

Increased Reporting of Incidents:

User Reports: Enhance incident reports filed by users by adding tools such as audio inputs, photo uploads, and more thorough classification choices.

Optimizing for Mobile:

Improve the user interface (UI) for mobile devices with responsive design to guarantee easy access and usefulness across a range of screen sizes and touch interfaces.



```
Untitled18.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
RAM Disk
Comment Share Gemini

# Example: Visualizing traffic data using matplotlib
import matplotlib.pyplot as plt

def visualize_traffic(data):
    # Plotting congestion levels
    time_intervals = [interval['time'] for interval in data]
    congestion_levels = [interval['congestion_level'] for interval in data]

    plt.plot(time_intervals, congestion_levels)
    plt.xlabel('Time')
    plt.ylabel('Congestion Level')
    plt.title('Real-time Traffic Congestion')
    plt.show()

# Example: Real-time data updating and integration
def real_time_traffic_monitoring(api_key):
    while True:
        # Fetch real-time traffic data
        traffic_data = fetch_traffic_data(api_key)

        # Process and analyze data
        duration, distance = process_traffic_data(traffic_data)
        congestion_level = analyze_traffic_congestion(traffic_data)

        # Visualize data
        visualize_traffic(congestion_level)

        # Sleep for a period before fetching new data (adjust based on update frequency)
        time.sleep(60) # Update every 60 seconds
```

DOCUMENTATION-4:

Problem 4: Real-Time COVID-19 Statistics Tracker

Scenario:

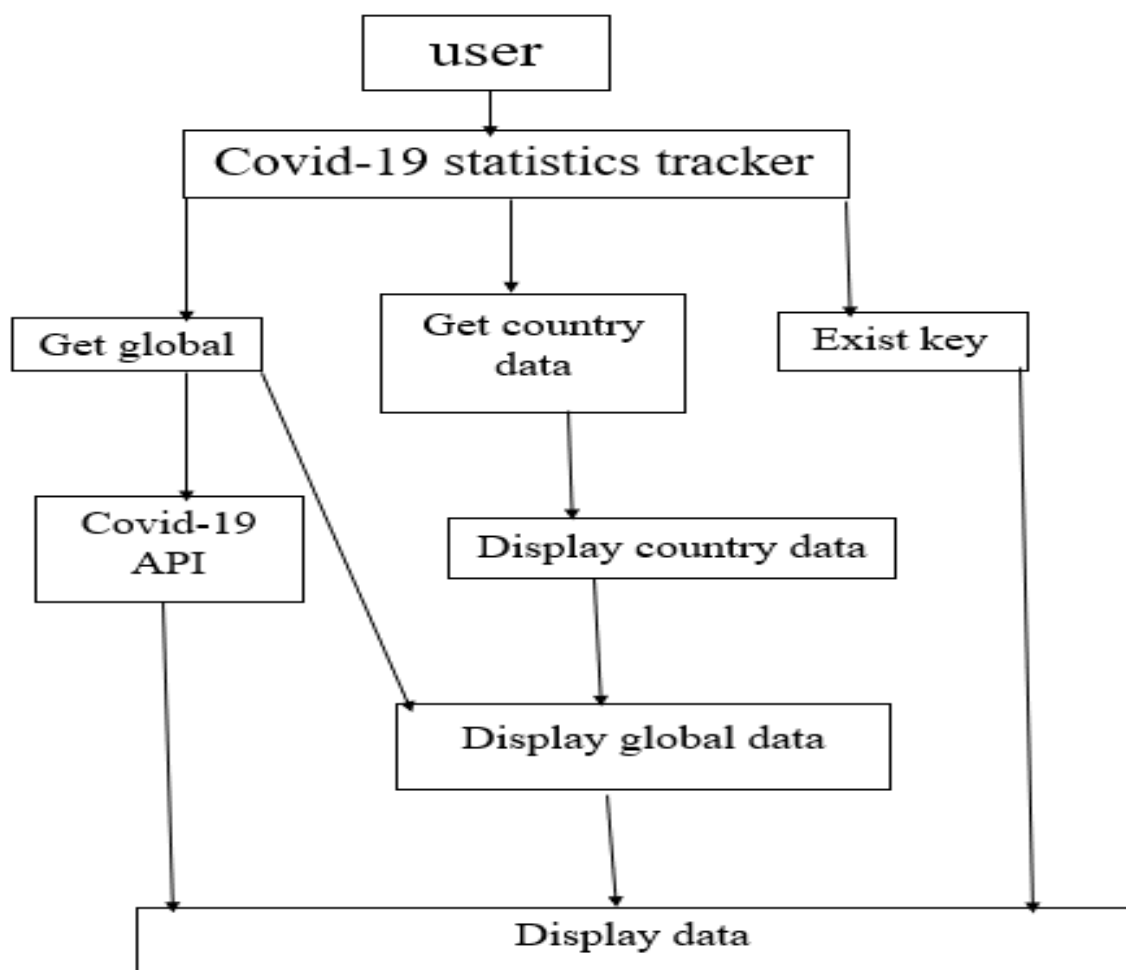
You are developing a real-time COVID-19 statistics tracking application for a healthcare organization. The application should provide up-to-date information on COVID-19 cases, recoveries, and deaths for a specified region.

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.

Real-Time COVID-19 Statistics Tracker

1: Data flow diagram:



2: IMPELEMENTATION CODE:

```
import requests
from prettytable import PrettyTable

def get_covid_data():
    url = "https://disease.sh/v3/covid-19/all"
    response = requests.get(url)
    data = response.json()
    return data

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

def display_global_data(data):
    table = PrettyTable()
    table.field_names = ["Metric", "Value"]
    table.add_row(["Total Cases", data['cases']])
    table.add_row(["Total Deaths", data['deaths']])
    table.add_row(["Total Recovered", data['recovered']])
    table.add_row(["Active Cases", data['active']])
    table.add_row(["Critical Cases", data['critical']])
    table.add_row(["Cases Today", data['todayCases']])
    table.add_row(["Deaths Today", data['todayDeaths']])
    table.add_row(["Recovered Today", data['todayRecovered']])
    table.add_row(["Affected Countries", data['affectedCountries']])
    print("Global COVID-19 Statistics:")
    print(table)

def display_country_data(data):
    table = PrettyTable()
    table.field_names = ["Metric", "Value"]
    table.add_row(["Country", data['country']])
    table.add_row(["Total Cases", data['cases']])
    table.add_row(["Total Deaths", data['deaths']])
    table.add_row(["Total Recovered", data['recovered']])
    table.add_row(["Active Cases", data['active']])
    table.add_row(["Critical Cases", data['critical']])
    table.add_row(["Cases Today", data['todayCases']])
    table.add_row(["Deaths Today", data['todayDeaths']])
    table.add_row(["Recovered Today", data['todayRecovered']])
    print(f"COVID-19 Statistics for {data['country']}:")
    print(table)

def main():
```

```

while True:
    print("\nCOVID-19 Statistics Tracker")
    print("1. Global Statistics")
    print("2. Country Statistics")
    print("3. Exit")
    choice = input("Enter your choice: ")

    if choice == '1':
        global_data = get_covid_data()
        display_global_data(global_data)
    elif choice == '2':
        country = input("Enter country name: ")
        country_data = get_country_data(country)
        display_country_data(country_data)
    elif choice == '3':
        break
    else:
        print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()

```

3.OUTPUT:

COVID-19 Statistics Tracker

1. Global Statistics
2. Country Statistics
3. Exit

Enter your choice: 1

Global COVID-19 Statistics:

+-----+	
Metric	Value
+-----+	
Total Cases	704753890
Total Deaths	7010681
Total Recovered	675619811
Active Cases	22123398
Critical Cases	34794
Cases Today	0
Deaths Today	0
Recovered Today	790
Affected Countries	231
+-----+	

COVID-19 Statistics Tracker

1. Global Statistics
2. Country Statistics
3. Exit

Enter your choice: 2

Enter country name: india

COVID-19 Statistics for India:

+-----+	
Metric	Value

+-----+-----+		
	Country	India
	Total Cases	45035393
	Total Deaths	533570
	Total Recovered	0
	Active Cases	44501823
	Critical Cases	0
	Cases Today	0
	Deaths Today	0
	Recovered Today	0
+-----+-----+		

COVID-19 Statistics Tracker

1. Global Statistics
2. Country Statistics
3. Exit

Enter your choice: 2

Enter country name: USA

COVID-19 Statistics for USA:

+-----+-----+		
	Metric	Value
+-----+-----+		
	Country	USA
	Total Cases	111820082
	Total Deaths	1219487
	Total Recovered	109814428
	Active Cases	786167
	Critical Cases	940
	Cases Today	0
	Deaths Today	0
	Recovered Today	0

COVID-19 Statistics Tracker

1. Global Statistics
2. Country Statistics
3. Exit

Enter your choice: 3

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.

5.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.

6. 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.



```
global_data = get_covid_data()
display_global_data(global_data)
elif choice == '2':
    country = input("Enter country name: ")
    country_data = get_country_data(country)
    display_country_data(country_data)
elif choice == '3':
    break
else:
    print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

COVID-19 Statistics Tracker

1. Global Statistics
2. Country Statistics
3. Exit

Enter your choice: 1

Global COVID-19 Statistics:

Metric	Value
Total Cases	704753890
Total Deaths	7010681
Total Recovered	675619811
Active Cases	22123398
Critical Cases	34794
Cases Today	0
Deaths Today	0
Recovered Today	790
Affected Countries	231

51s completed at 3:50 PM