ASSIGNMENT-3

NAME: K. NAVYA

REGNO: 192371022

SUBJECT: PYTHON PROGRAMMING

SUB CODE: CSA0809

DATE OF SUB: 17-07-2024

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., Open Weather Map) to fetch real-time weather data.

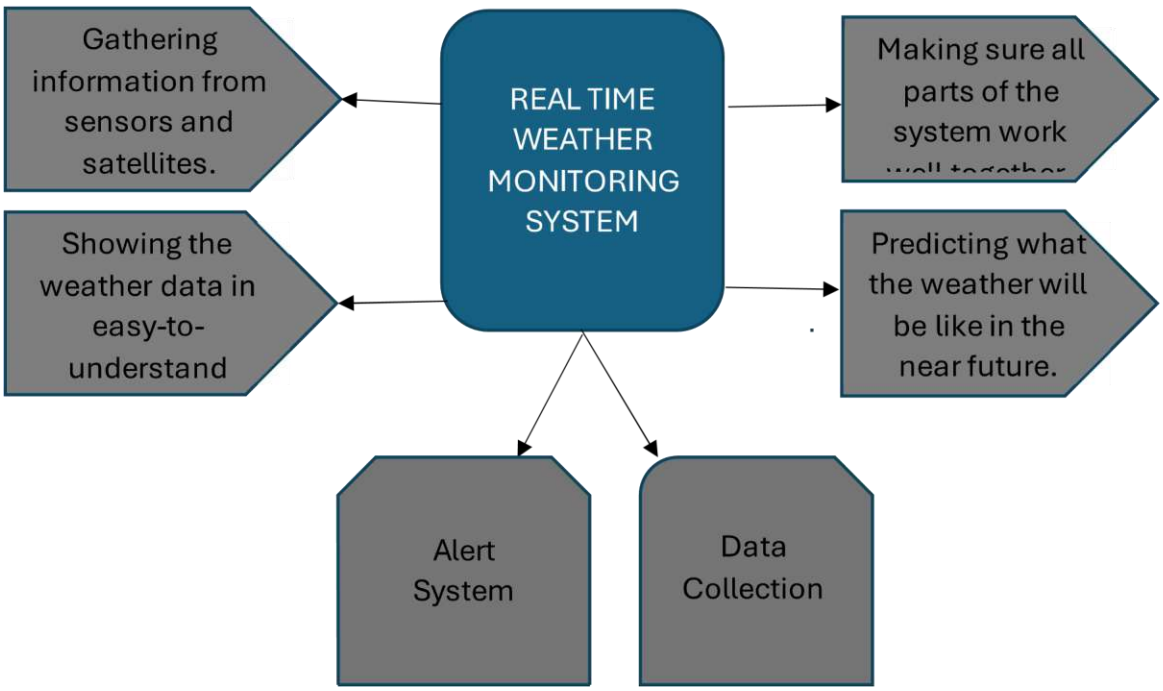
3. Display the current weather information, including temperature, weather conditions, humidity, and wind speed.

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: PSEUDOCODE

```
1. Import necessary libraries: requests, json

2. Define a function get_weather(api_key, city):
   2.1 Construct the API URL with the provided api_key and city name
   2.2 Send a GET request to the API URL using requests.get()
   2.3 If the response status code is 200 (OK):
       2.3.1 Parse the JSON response using response.json()
       2.3.2 Extract weather data from the JSON response:
           - weather_description
           - temperature
           - humidity
           - wind_speed
           - pressure
           - visibility (handle if it's not available)
           - cloudiness
       2.3.3 Print the weather data for the specified city
   2.4 If the response status code is not 200:
       2.4.1 Print an error message indicating the failure to fetch weather
data

3. Define main code execution:
   3.1 Replace 'your_api_key_here' with your actual OpenWeatherMap API key
   3.2 Replace 'New York' with the city name you want to monitor
   3.3 Call get_weather(api_key, city) function with your API key and city
```

3.IMPLEMENTATION

```
import requests
import json

def get_weather(api_key, city):
    url =
f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&unit
s=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']
        pressure = data['main']['pressure']
        visibility = data.get('visibility', 'N/A')
        cloudiness = data['clouds']['all']

        print(f"Weather in {city}:")
        print(f"Description: {weather_description}")
        print(f"Temperature: {temperature} °C")
        print(f"Humidity: {humidity}%")
        print(f"Wind Speed: {wind_speed} m/s")
        print(f"Pressure: {pressure} hPa")
        print(f"Visibility: {visibility} meters")
        print(f"Cloudiness: {cloudiness}%")
    else:
        print(f"Error fetching weather data. Status code:
{response.status_code}")

api_key = '09fc98b8358a08a6a4ce604e9443f368'
city = 'New York'

get_weather(api_key, city)
```

4.OUTPUT

Weather in New York:

Description: clear sky

Temperature: 29.26 °C

Humidity: 55%

Wind Speed: 2.57 m/s

Pressure: 1011 hPa

Visibility: 10000 meters

Cloudiness: 0%

5: DOCUMENTATION

Overview:

- The Real-Time Weather Monitoring System aims to provide up-to-date weather information for specified locations using the OpenWeatherMap API.

Functionality:

- Weather description, temperature, humidity, wind speed, pressure, visibility, and cloudiness.
- Retrieves weather data by constructing the API request URL, sending a GET request, parsing JSON response, and printing weather details.

Usage:

- View current weather information on the command line.

6.ASSUMPTION AND IMPROVEMENT

Localization Support:

- Enhance the application to support fetching weather data in multiple languages or formats based on user preferences or location.

Historical Weather Data:

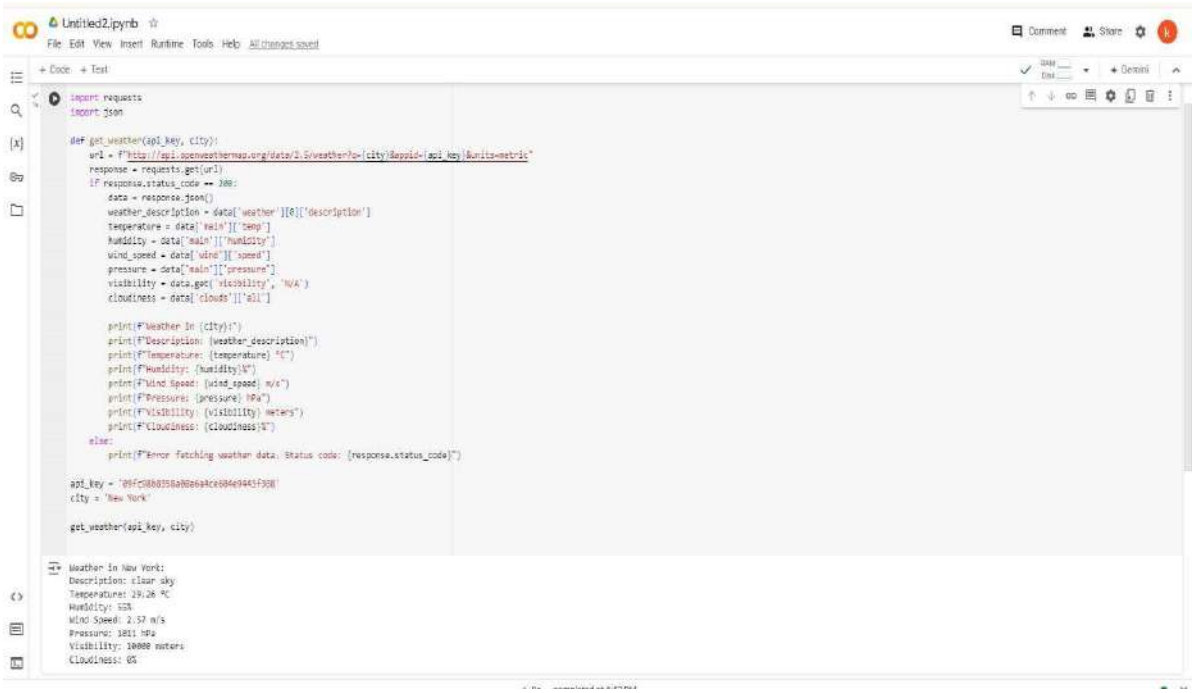
- Extend functionality to fetch historical weather data for analysis or comparison, beyond just current weather conditions.

Alert System:

- Integrate an alert system that notifies users of significant weather changes or warnings based on predefined thresholds.

Mobile Compatibility:

- Ensure the system is mobile-friendly or develop a dedicated mobile app for users to access weather information on the go.



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.

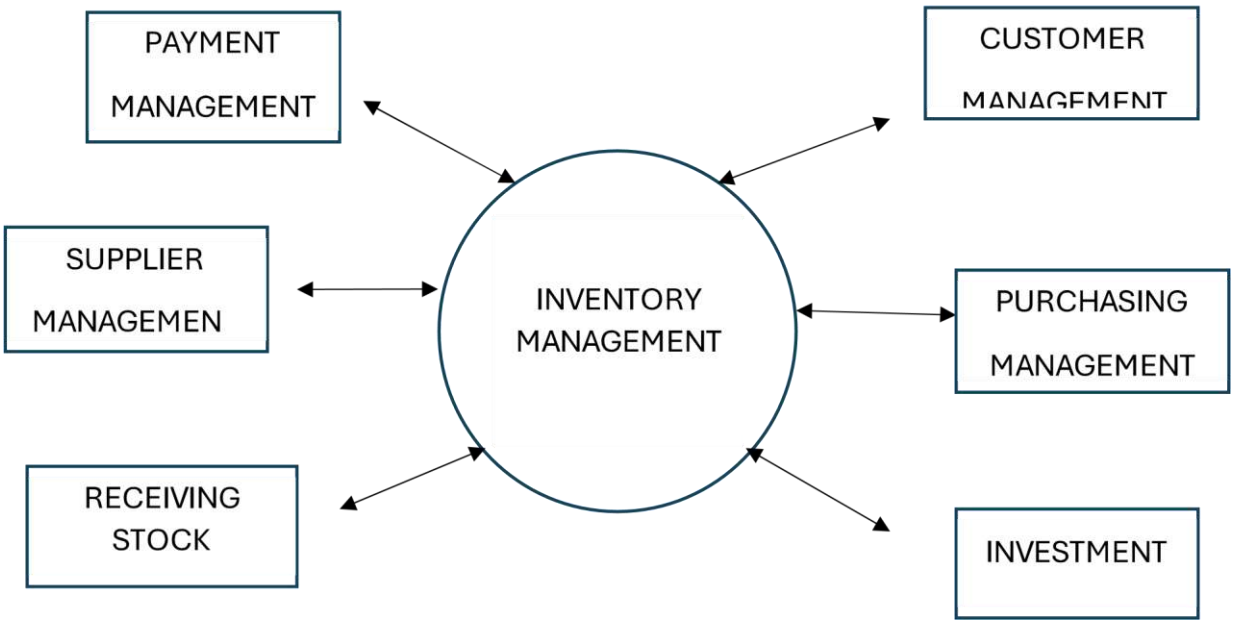
4. Generate reports: Provide reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.

5. User interaction: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

SOLUTION :

INVESTORY MANAGEMENT SYSTEM OPTIMIZATION

1: DATA CHART DIAGRAM



2: IMPLEMENTATION

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

    def add_item(self, item_name, quantity):
        if item_name in self.inventory:
            self.inventory[item_name] += quantity
        else:
            self.inventory[item_name] = quantity

    def remove_item(self, item_name, quantity):
        if item_name in self.inventory:
            if self.inventory[item_name] >= quantity:
                self.inventory[item_name] -= quantity
                if self.inventory[item_name] == 0:
                    del self.inventory[item_name]
                return True
```

```
            else:
                print(f"Not enough {item_name} in stock.")
                return False
        else:
            print(f"{item_name} not found in inventory.")
            return False

    def update_quantity(self, item_name, new_quantity):
        if item_name in self.inventory:
            self.inventory[item_name] = new_quantity
        else:
            print(f"{item_name} not found in inventory. Adding it now.")
            self.inventory[item_name] = new_quantity

    def display_inventory(self):
        if not self.inventory:
            print("Inventory is empty.")
        else:
            print("Current Inventory:")
            for item, quantity in self.inventory.items():
                print(f"{item}: {quantity}")

if __name__ == "__main__":
    manager = InventoryManager()
    manager.add_item("Apple", 50)
    manager.add_item("Banana", 30)
    manager.add_item("Orange", 40)
    manager.display_inventory()
    manager.update_quantity("Banana", 25)
    manager.remove_item("Apple", 20)
    manager.display_inventory()
```

_3.OUTPUT

Current Inventory:

Apple: 50

Banana: 30

Orange: 40

Current Inventory:

Apple: 30

Banana: 25

Orange: 40

4:DOCUMENTATION

Security:

- secure communication protocols to safeguard sensitive inventory information.

User Interface and Experience:

- Design a user-friendly interface with intuitive features for managing inventory items, updating quantities, and generating reports.

Scalability:

- Design the system to handle increasing numbers of inventory items, users, and transactions over time.

5:USER INTERFACE

Dashboard Overview:

- Display summary of key inventory metrics (total items, low stock alerts, etc.)

Reports and Analytics:

- View reports on transaction history, sales, and inventory movements and Predictive analytics for inventory planning based on historical data.

Usage:

- Optimize the user interface of the Inventory Management System for improved efficiency and usability.

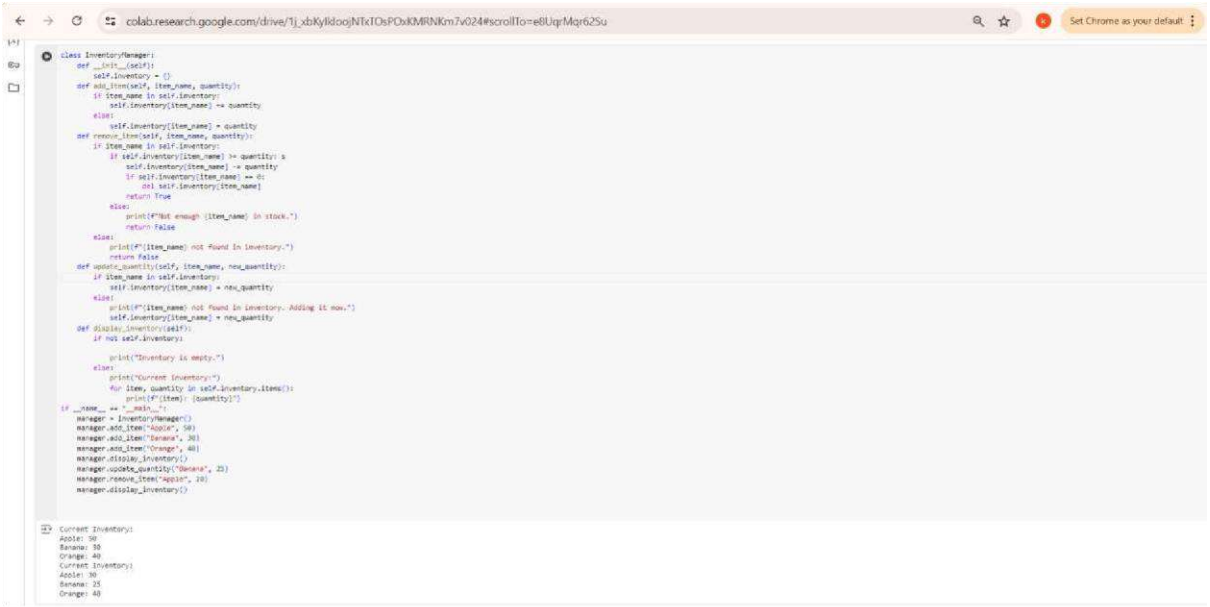
6.ASSUMPTIONS AND IMPROVEMENTS

Assumptions:

- Users have basic knowledge of the system and its functionalities.
- Current system performance issues may stem from inefficient processes.
- Inventory data accuracy is crucial for decision-making.

Improvements:

- Provide analytical tools for forecasting demand and optimizing stock levels.
- Ensure mobile compatibility for on-the-go access to inventory information.
- Conduct regular training sessions for users to maximize system utilization.



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.

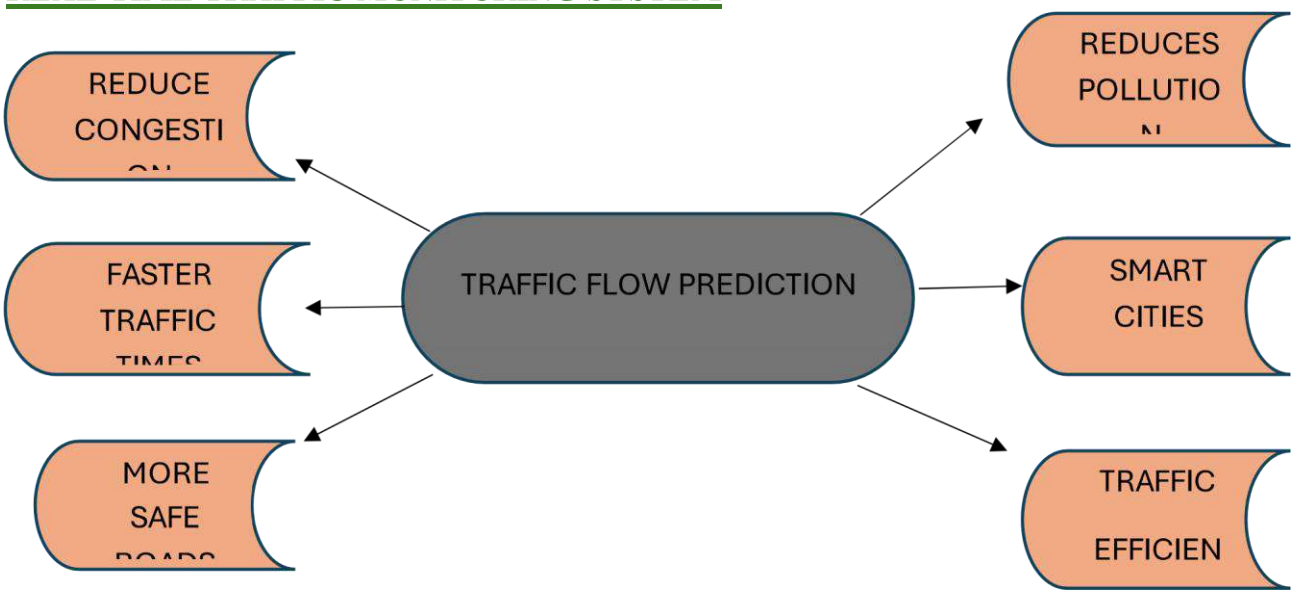
3. Display current traffic conditions, estimated travel time, and any incidents or delays.

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

1: DATA CHART DIAGRAM

SOLUTION :

REAL-TIME TRAFFIC MONITORING SYSTEM



2: PSEUDOCODE

```
function generate_traffic_data(num_points):
    timestamps = []
    traffic_flows = []
    congestion_levels = []

    for i from 1 to num_points:
        timestamps[i] = "Time i"
        traffic_flows[i] = random integer between 50 and 100
        congestion_levels[i] = random float between 0 and 10

    return timestamps, traffic_flows, congestion_levels

function main():
    num_points = 10
      timestamps, traffic_flows, congestion_levels =
generate_traffic_data(num_points)

    for i from 1 to num_points:
        print "Timestamp i: Traffic Flow=traffic_flows[i], Congestion
Level=congestion_levels[i]"

if script is executed directly:
    run the main function
```

3.IMPLEMENTATION

```
import random
def generate_traffic_data(num_points):
    timestamps = [f'Time {i+1}' for i in range(num_points)]
    traffic_flows = [random.randint(50, 100) for _ in range(num_points)]
    congestion_levels = [random.uniform(0, 10) for _ in range(num_points)]

    return timestamps, traffic_flows, congestion_levels

def main():
    num_points = 10

    timestamps, traffic_flows, congestion_levels =
generate_traffic_data(num_points)

    for i in range(num_points):
        print(f"{timestamps[i]}: Traffic Flow={traffic_flows[i]}, Congestion
Level={congestion_levels[i]}")

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

3.OUTPUT

Time 1: Traffic Flow=63, Congestion Level=2.057383624509238

Time 2: Traffic Flow=87, Congestion Level=4.407019028509783

Time 3: Traffic Flow=50, Congestion Level=5.581688379470457

Time 4: Traffic Flow=69, Congestion Level=7.781959523473124

Time 5: Traffic Flow=55, Congestion Level=9.738473627342529

Time 6: Traffic Flow=93, Congestion Level=8.098231662482638

Time 7: Traffic Flow=76, Congestion Level=7.040424264761346

Time 8: Traffic Flow=65, Congestion Level=3.8939014243349

Time 9: Traffic Flow=78, Congestion Level=9.71554277536545

Time 10: Traffic Flow=92, Congestion Level=5.946842677518548

4: DOCUMENTATION

Data Acquisition: Captures traffic data from sensors, cameras, or external APIs.

Real-Time Processing: Analysis incoming data to detect traffic patterns and anomalies.

Alerting: Notifies stakeholders about high traffic flow, slow speeds, or predefined conditions.

Visualization: Provides visual representations of traffic data for easier interpretation.

Customization: Configurable to adapt to different traffic monitoring needs.

5: ASSUMPTIONS AND IMPROVEMENTS

Real-Time Data Availability: Expectation that traffic data is available promptly and consistently to ensure timely monitoring and response.

Environmental Factors: Consideration of external factors such as weather conditions or road construction impacting traffic patterns.

Traffic Flow Optimization: integrate with traffic control systems to dynamically adjust signal timings based on real-time traffic data, optimizing traffic flow and reducing congestion.

User Interface Improvements: Improve user interfaces for easier navigation, customizable views, and intuitive controls, catering to both technical and non-technical users.

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.

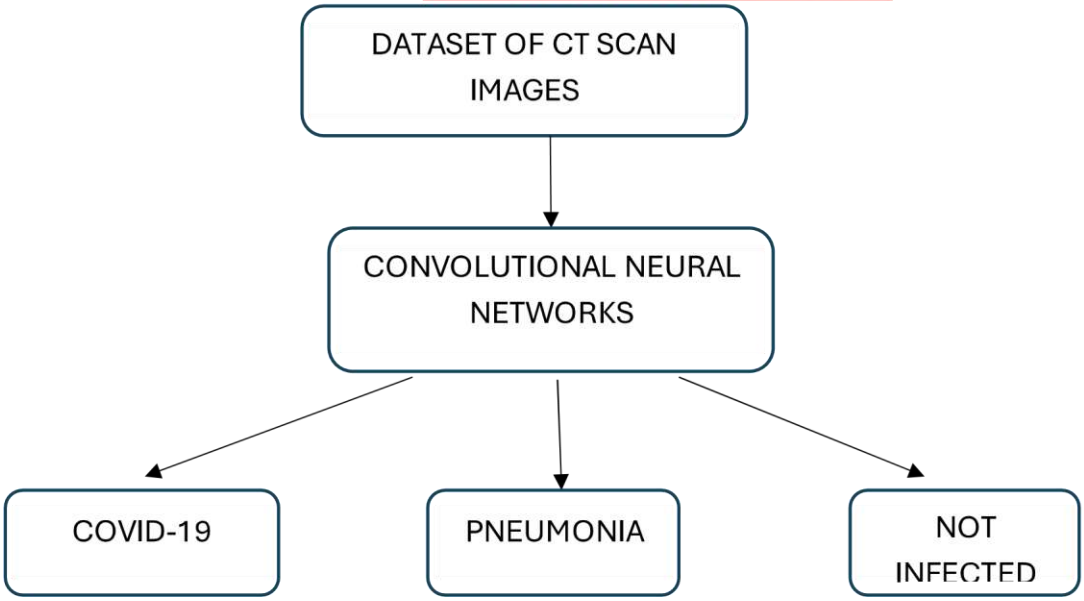
3. Display the current number of cases, recoveries, and deaths for a specified region.

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

SOLUTION :

REAL-TIME COVID-19 STATISTICS TRACKER

1: DATA CHART DIAGRA



2: PSEUDOCODE

```
1. Define a function get_covid_stats(country):
    a. Construct the API URL based on the provided country.
    b. Try to send a GET request to the constructed URL.
    c. If the request is successful (status code 200):
       i. Parse the response JSON data.
       ii. Return the parsed data.
    d. If the request fails (status code is not 200):
       i. Print an error message indicating the failure.
       ii. Return None.
    e. Handle exceptions (e.g., network errors) and print appropriate error
messages.

2. Define a function display_stats(data):
    a. Check if the data parameter is not None:
       i. Extract and print the country name from the data.
       ii. Extract and print the total number of cases, deaths, and recovered
cases.
    b. If the data parameter is None:
       i. Print a message indicating that no data is available for the
specified country.

3. Define the main function:
    a. Specify the country variable with the name of the country to fetch
COVID-19 statistics for.
    b. Call get_covid_stats function with the specified country and store the
returned data.
    c. Call display_stats function with the retrieved data to display the
statistics.

4. Execute the main function if this script is run as the main module.
```

3.IMPLEMENTATION

```
import requests

def get_covid_stats(country):
    url = f"https://disease.sh/v3/covid-19/countries/{country}"
    try:
        response = requests.get(url)
        if response.status_code == 200:
```

```
            data = response.json()
            return data
        else:
            print(f"Failed to fetch data: {response.status_code}")
            return None
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")
        return None

def display_stats(data):
    if data:
        country = data['country']
        cases = data['cases']
        deaths = data['deaths']
        recovered = data['recovered']

        print(f"COVID-19 Statistics for {country}:")
        print(f"Total Cases: {cases}")
        print(f"Total Deaths: {deaths}")
        print(f"Total Recovered: {recovered}")
    else:
        print("No data available for the specified country.")

def main():
    country = "INDIA"  # Replace with the country you want to track
    stats = get_covid_stats(country)
    display_stats(stats)

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

4:DOCUMENTATION

3.OUTPUT

COVID-19 Statistics for India:

Total Cases: 45035393

Total Deaths: 533570

Total Recovered: 0

Data Display

Statistics retrieved from the API are displayed in a formatted output, including:

- Total cases
- Total deaths
- Total recovered

Usage

1. **Setup**: Ensure Python and necessary libraries are installed.
2. **Execution**: Run the script, specifying the desired country.
3. **Output**: View real-time COVID-19 statistics for the specified country.

Future Enhancements

- **Graphical Interface**: Develop a graphical interface for easier interaction and visualization of statistics.
- **Historical Data**: Include historical data tracking and visualization.

  Global Statistics: Extend functionality to retrieve and compare statistics across multiple countries.

5:ASSUMPTIONS AND IMPROVEMENTS

User Interaction: Assumes users have basic knowledge of running Python scripts and interpreting COVID-19 statistics.

Enhanced Data Visualization:Implement graphical representations (charts, graphs) of COVID-19 statistics for better visual understanding.

Historical Data Tracking:Introduce functionality to track and display historical trends of COVID-19 cases, deaths, and recoveries over time.

Global Comparison:Extend the tracker to compare COVID-19 statistics across multiple countries simultaneously, providing a broader perspective.

Predictive Analysis:Incorporate machine learning models to predict future COVID-19 trends based on historical data and current state.