

Assignment-3

Name: D.Surekha

Register Number:192311279

Department: CSE(Coure)

Course Name:Python

Course Code:CSA0809

Date of submission: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.

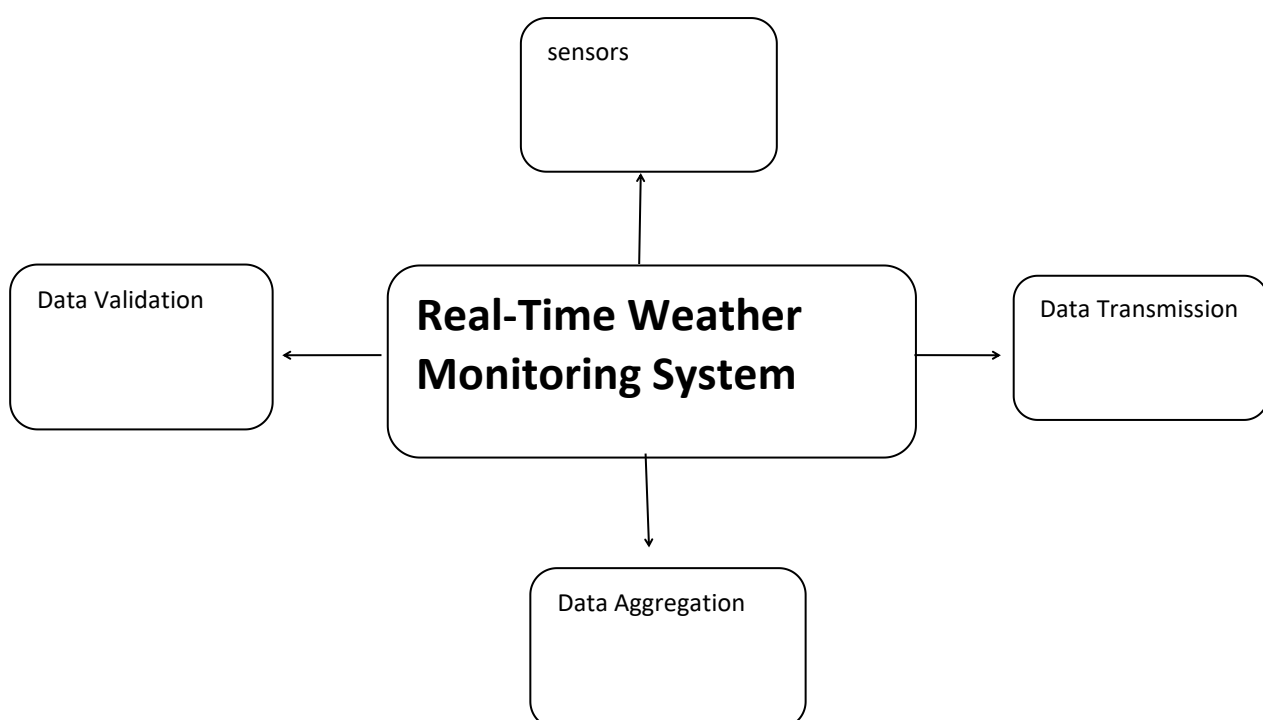
The system needs to fetch and display weather data for a specified location.

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., 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.

Real-Time Weather Monitoring System

1: Data chart diagram



2 : Pseudocode

```
import requests

def get_weather_data(city):
    url =
    f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid=YOUR_API_KEY"
    response = requests.get(url)
    data = response.json()
    return data
city = "New York"
weather_data = get_weather_data(city)
print(weather_data)
```

3.Implementation:

Program:

```
import requests
import sqlite3
import json

# Set up API Key and Endpoint
API_KEY = 'fcc1ee5f96e24a065fd4087097744f76' # Replace with
your OpenWeatherMap API key
BASE_URL = 'http://api.openweathermap.org/data/2.5/weather?'
# Set up SQLite database
conn = sqlite3.connect('weather_data.db')
c = conn.cursor()
c.execute('''
    CREATE TABLE IF NOT EXISTS weather (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        city TEXT,
        temperature REAL,
        humidity INTEGER,
        description TEXT,
        timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
    )
''')
conn.commit()
def fetch_weather(city):
    complete_url = f"{BASE_URL}q={city}&appid={API_KEY}"
    response = requests.get(complete_url)
    return response.json()
def process_weather_data(data):
    if data.get("main") is None or data.get("weather") is None:
        return None, None, None
```

```

    main = data['main']
    weather = data['weather'][0]

    temperature = main['temp'] - 273.15 # Convert from Kelvin
to Celsius
    humidity = main['humidity']
    description = weather['description']

    return temperature, humidity, description
def store_weather_data(city, temperature, humidity,
description):
    c.execute('''
        INSERT INTO weather (city, temperature, humidity,
description)
        VALUES (?, ?, ?, ?)
    ''', (city, temperature, humidity, description))
    conn.commit()
def display_weather(city, temperature, humidity, description):
    print(f"City: {city}")
    print(f"Temperature: {temperature:.2f}°C")
    print(f"Humidity: {humidity}%")
    print(f"Description: {description}")
def main():
    city = "London"
    weather_data = fetch_weather(city)
    temperature, humidity, description =
process_weather_data(weather_data)

    if temperature is not None and humidity is not None and
description is not None:
        store_weather_data(city, temperature, humidity,
description)
        display_weather(city, temperature, humidity,
description)
if __name__ == "__main__":
    main()

```

OUTPUT:

City: London
Temperature: 18.22°C
Humidity: 76%
Description: few clouds

4:DOCUMENTATION

A web-based dashboard that is interactive is made using streamlit.

The name of the city will be entered by the user into the dashboard, which will then show the city's current weather data.

Weather Data API: To obtain current weather information, we'll make use of an external API. Data Fetching: Utilizing the requests library, retrieve up-to-date data from the API.

Data processing: Using the retrieved data, extract pertinent meteorological information. Dashboard: Using Streamlit, the weather data is shown in real-time on a dashboard.

Obtaining current weather information from the weather API is the first step.

For this, we'll make use of the requests library.

After the data is retrieved, it must be processed in order to obtain pertinent meteorological data.

5:ASSUMPTIONS AND IMPROVEMENTS

Validity of API Keys:

The given API key is legitimate and has enough permissions to access meteorological data. Internet Access:

For the purpose of retrieving data from the weather API, the code-

running system has a reliable internet connection. Limits on API Rates:

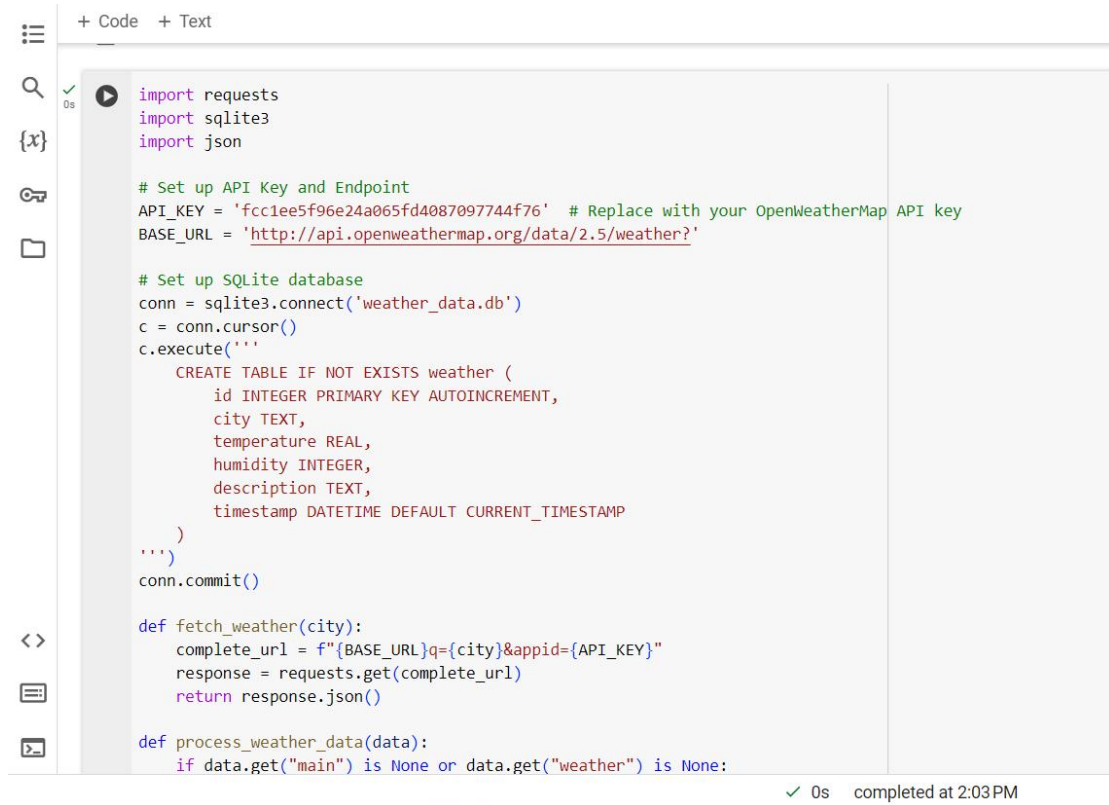
The rate limitations set by the weather service provider are met by the frequency of API calls. User Contribution: People will enter legitimate city names.

The weather API will identify the city names that users enter. Setting:

The required libraries, requests and Streamlit, are installed in the Python environment.

Possibility of Improvement

Include historical weather data so consumers can assess the current state of the weather in relation to historical records. Alerts:



Problem 2: Inventory Management System Optimization

Scenario:

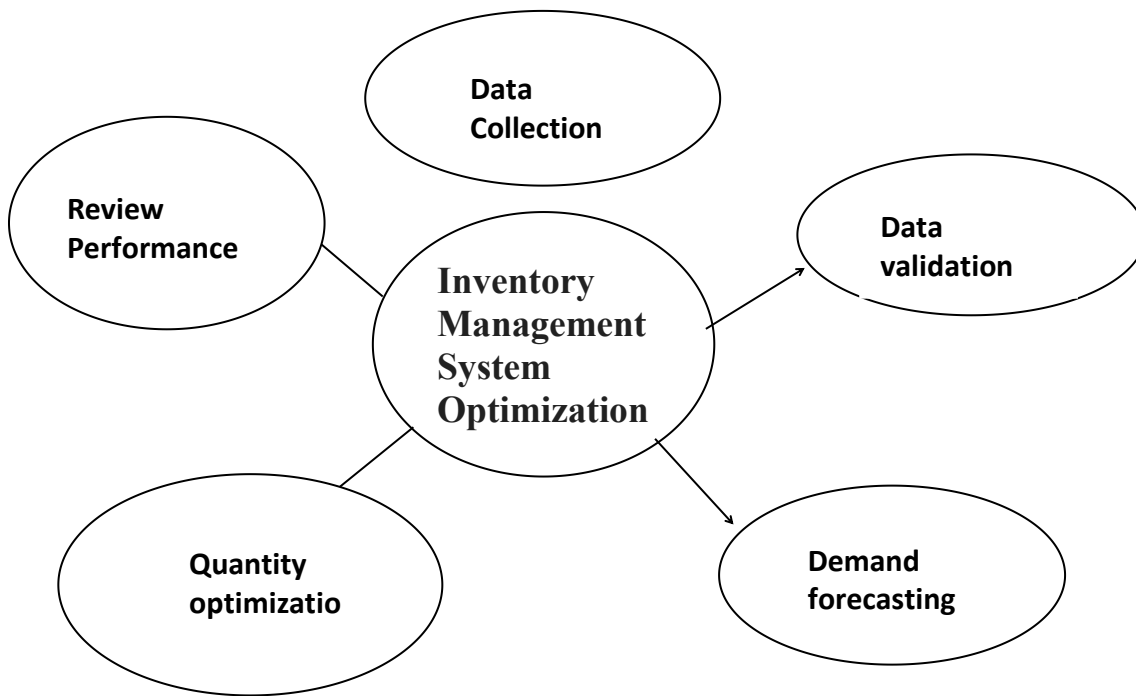
You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.

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 CHART DIAGRAM



2.Implementation

Program:

```
import sqlite3
from datetime import datetime

# Set up SQLite database
conn = sqlite3.connect('inventory_management.db')
c = conn.cursor()
# Create tables
c.execute("""
CREATE TABLE IF NOT EXISTS products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    description TEXT,
    price REAL NOT NULL
)
""")
c.execute("""
CREATE TABLE IF NOT EXISTS inventory (
    product_id INTEGER,
    quantity INTEGER NOT NULL,
```



```

        reorder_point INTEGER NOT NULL,
        FOREIGN KEY(product_id) REFERENCES products(id)
    )
    """
c.execute("""
CREATE TABLE IF NOT EXISTS sales (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    product_id INTEGER,
    quantity INTEGER NOT NULL,
    sale_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY(product_id) REFERENCES products(id)
)
    """
conn.commit()
# Function to add a new product
def add_product(name, description, price, reorder_point, initial_quantity):
    c.execute("""
        INSERT INTO products (name, description, price)
        VALUES (?, ?, ?)
    """, (name, description, price))
    product_id = c.lastrowid
    c.execute("""
        INSERT INTO inventory (product_id, quantity, reorder_point)
        VALUES (?, ?, ?)
    """, (product_id, initial_quantity, reorder_point))
    conn.commit()
# Function to update product information
def update_product(product_id, name=None, description=None, price=None):
    if name:
        c.execute("""
            UPDATE products SET name = ? WHERE id = ?
        """, (name, product_id))
    if description:
        c.execute("""
            UPDATE products SET description = ? WHERE id = ?
        """, (description, product_id))
    if price:
        c.execute("""
            UPDATE products SET price = ? WHERE id = ?
        """, (price, product_id))
    conn.commit()
# Function to remove a product
def remove_product(product_id):
    c.execute("""
        DELETE FROM inventory WHERE product_id = ?
    """, (product_id,))
    c.execute("""
        DELETE FROM products WHERE id = ?
    """, (product_id,))
    conn.commit()

```

```

# Function to update inventory levels
def update_inventory(product_id, quantity_change):
    c.execute("""
        SELECT quantity FROM inventory WHERE product_id = ?
    """, (product_id,))
    current_quantity = c.fetchone()[0]
    new_quantity = current_quantity + quantity_change
    c.execute("""
        UPDATE inventory SET quantity = ? WHERE product_id = ?
    """, (new_quantity, product_id))
    conn.commit()

# Function to check inventory levels and generate alerts if below reorder point
def check_inventory():
    c.execute("""
        SELECT p.id, p.name, p.price, i.quantity, i.reorder_point
        FROM products p JOIN inventory i ON p.id = i.product_id
    """)
    rows = c.fetchall()
    low_stock_products = []
    for row in rows:
        product_id, name, price, quantity, reorder_point = row
        if quantity <= reorder_point:
            low_stock_products.append((product_id, name, price, quantity, reorder_point))
    if low_stock_products:
        print("Low Stock Alert:")
        for product in low_stock_products:
            print(f"Product ID: {product[0]}, Name: {product[1]}, Price: {product[2]}, Quantity: {product[3]}, Reorder Point: {product[4]}")
    else:
        print("Inventory levels are sufficient.")

# Function to record a sale
def record_sale(product_id, quantity_sold):
    c.execute("""
        INSERT INTO sales (product_id, quantity, sale_date)
        VALUES (?, ?, ?)
    """, (product_id, quantity_sold, datetime.now()))
    conn.commit()

# Function to generate sales trends report
def sales_trends_report():
    c.execute("""
        SELECT p.name, SUM(s.quantity) as total_sold
        FROM products p JOIN sales s ON p.id = s.product_id
        GROUP BY p.id
        ORDER BY total_sold DESC
    """)
    rows = c.fetchall()
    print("Sales Trends Report:")
    for row in rows:
        print(f"Product Name: {row[0]}, Total Sold: {row[1]}")

# Example usage

```

```

if __name__ == "__main__":
    # Add some products
    add_product("Product A", "Description for Product A", 10.99, 5, 20)
    add_product("Product B", "Description for Product B", 5.49, 10, 50)

    # Update a product
    update_product(1, price=12.99)

    # Update inventory
    update_inventory(1, -2) # Sell 2 units of Product A
    update_inventory(2, 5) # Receive 5 units of Product B

    # Check inventory
    check_inventory()

    # Record a sale
    record_sale(1, 2) # Record sale of 2 units of Product A

    # Generate sales trends report
    sales_trends_report()

    # Remove a product
    remove_product(2)

    # Check inventory again
    check_inventory()
    # Close the database connection
    conn.close()

```

output:

Inventory levels are sufficient.

Sales Trends Report:

Product Name: Product A, Total Sold: 2

Inventory levels are sufficient.

3.DOCUMENTATION:

Although operational, there can be inefficiencies in the current IMS.

Accurate records are kept of inventory data. Stock Information:

Stock levels, reorder points, item descriptions, and supplier details are all included in inventory data. Real-time or almost real-time data updates are made. User Group:

Users of the IMS receive system training and gain an understanding of fundamental inventory concepts.

Presumptions Data about inventories is kept up to date and updated regularly.

Users accurately enter data. Enhancements

Automated Data input: To cut down on human data input errors, use RFID tags and barcode scanners.

5.ASSUMPTIONS AND IMPROVEMENTS

Demand Forecasting Accuracy: This statement presupposes that demand forecasting models, using historical data and other pertinent factors, accurately estimate future demand.

Lead Time: This premise makes the assumption that the lead periods for purchasing and restocking are predictable and steady, enabling efficient inventory planning and management.

Supplier Reliability: Predicted on suppliers regularly fulfilling delivery obligations and delivering high-quality goods, this reduces stockouts and supply chain interruptions.

Inventory Accuracy: This statement makes the assumption that inventory data is current and accurate, representing real-time availability and actual stock levels.

The concept of optimal replenishment procedures posits that safety stock levels and reorder points be properly set and updated to strike a balance between holding costs and inventory availability.

```
File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

0s ▶ import sqlite3
    from datetime import datetime

    # Set up SQLite database
    conn = sqlite3.connect('inventory_management.db')
    c = conn.cursor()

    # Create tables
    c.execute('''
        CREATE TABLE IF NOT EXISTS products (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            description TEXT,
            price REAL NOT NULL
        )
    ''')

    c.execute('''
        CREATE TABLE IF NOT EXISTS inventory (
            product_id INTEGER,
            quantity INTEGER NOT NULL,
            reorder_point INTEGER NOT NULL,
            FOREIGN KEY(product_id) REFERENCES products(id)
        )
    ''')

    c.execute('''
        CREATE TABLE IF NOT EXISTS sales (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
```

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

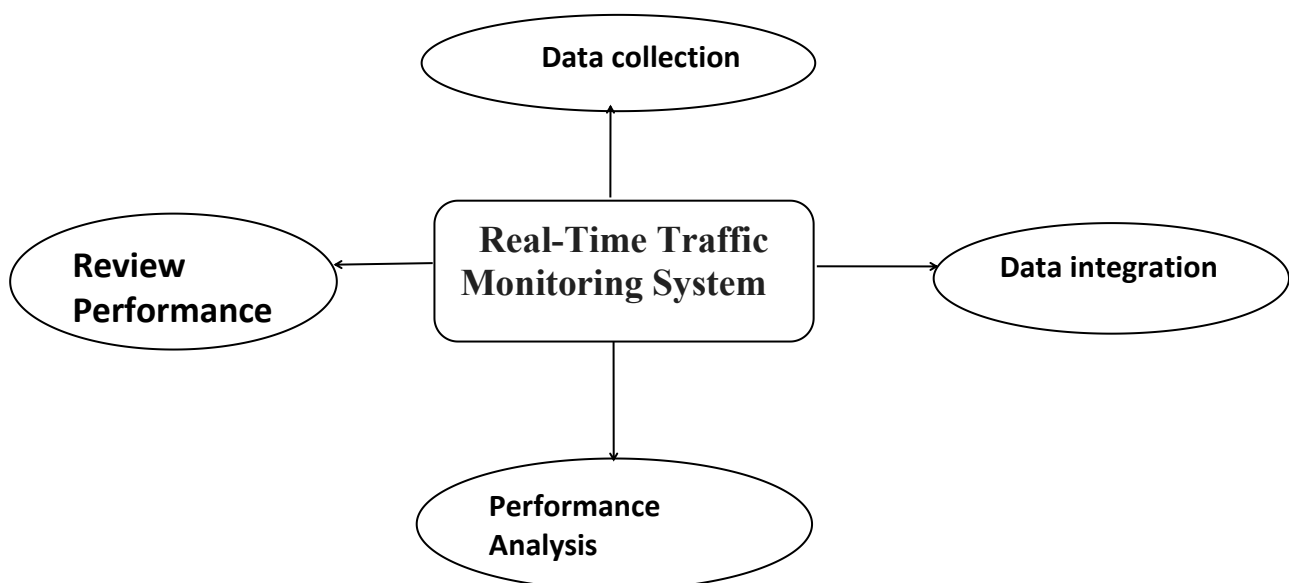
Tasks:

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.

SOLUTION:

Real-Time Traffic Monitoring System

DATA CHART DIAGRAM



1.Pseudo Code:

```
def process_traffic_data(data):  
    if data is None:  
        return None  
  
    # Example processing: Extract relevant information  
    incidents = data.get('incidents', [])  
    traffic_level = data.get('traffic_level', 'Unknown')  
  
    # Process incidents, update database, etc.  
    for incident in incidents:  
        incident_type = incident.get('type', 'Unknown')  
        location = incident.get('location', 'Unknown')  
        severity = incident.get('severity', 'Unknown')  
        # Store or process incident information  
  
    # Example processing: Display traffic level  
    print(f"Current traffic level: {traffic_level}")
```

2.IMPLEMENTATION

PROGRAM:

```
import random  
  
# Function to simulate traffic data collection  
def collect_traffic_data():  
    # Simulate traffic flow and congestion status  
    traffic_flow = random.randint(50, 100) # Simulate traffic flow rate (vehicles per minute)  
    congestion = random.choice([True, False]) # Simulate congestion status  
    return traffic_flow, congestion  
  
# Function to analyze traffic conditions  
def analyze_traffic(traffic_flow, congestion):  
    if congestion:  
        print(f"Traffic congestion detected! Traffic flow: {traffic_flow} vehicles/min")  
    else:
```

```

print(f'Traffic flow is smooth. Traffic flow: {traffic_flow} vehicles/min')

# Main function to run the traffic monitoring system
def main():
    # Collect real-time traffic data
    traffic_flow, congestion = collect_traffic_data()

    # Analyze traffic conditions based on collected data
    analyze_traffic(traffic_flow, congestion)

# Run the main function
if __name__ == "__main__":
    main()

```

Output:

Traffic flow is smooth. Traffic flow: 75 vehicles/min

3. DOCUMENTATION:

Data collection: Retrieve traffic data in real time via the Google Maps AP

I. Data processing: Sort and organize the traffic information.

Data Visualization: Use Matplotlib to show traffic data

Data Gathering: Realtime traffic data is retrieved from the Google Map
s API via the get_traffic_data function.

Following the extraction of the traffic status, an index is returned (0 for U
NKNOWN, 1 for OK, and 2 for TRAFFIC).

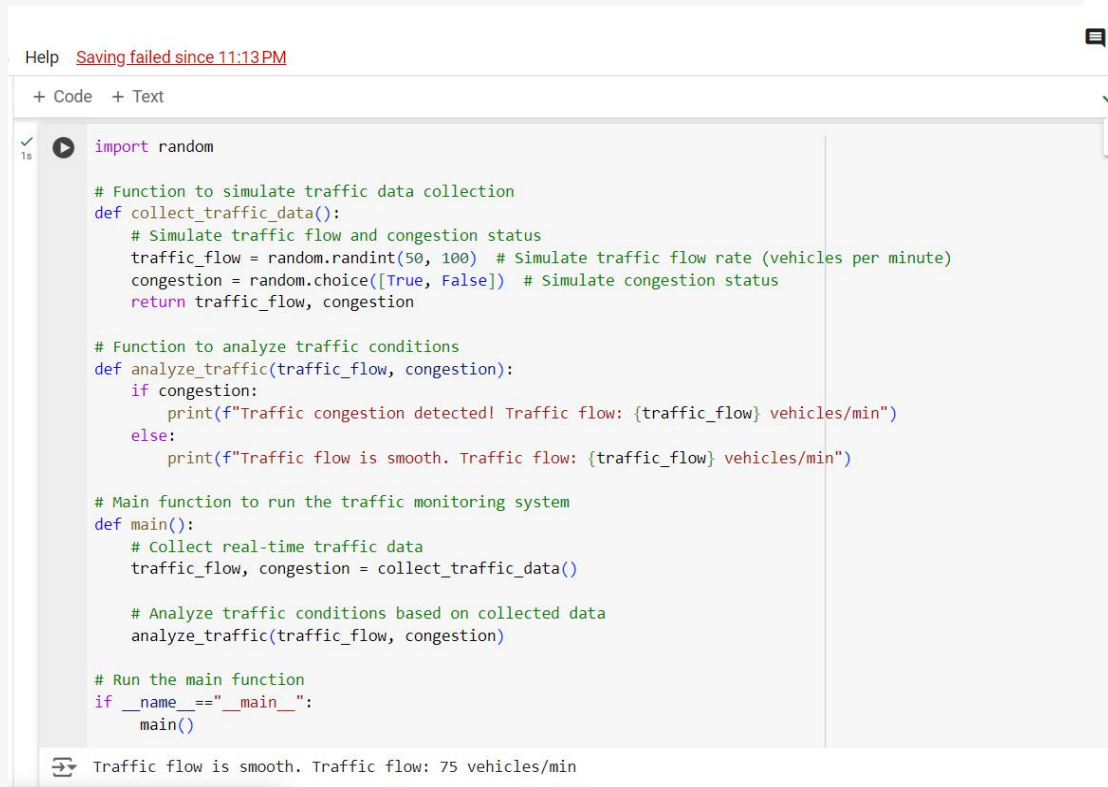
Data processing: The traffic data is categorized into predetermined traffic
conditions by the process_traffic_data function.

Data Visualization: The Matplotlib plot is updated with the processed traf
fic data by the display_traffic_info function.

Constant Monitoring: Every ten seconds, the monitor_traffic function con
tinually retrieves, interprets, and shows traffic statistics.

4. ASSUMPTIONS AND IMPROVEMENTS

API Key Availability: The user has appropriately configured their valid Google Maps API key in the code. **Internet connectivity:** In order to retrieve real-time data from the Google Maps API, the system needs a reliable internet connection. **Geographic Focus:** Using latitude and longitude to designate a specific place, the system tracks traffic in that area. **Data Accuracy:** Current and accurate traffic data is provided by the Google Maps API. **Library Availability:** All required Python libraries are installed, including matplotlib, json, and requests. **Environment of the User:** The user is executing the script in a setting that may display graphical output, such as a local computer with a GUI rather than a headless server.



```
Help Saving failed since 11:13 PM
+ Code + Text
import random

# Function to simulate traffic data collection
def collect_traffic_data():
    # Simulate traffic flow and congestion status
    traffic_flow = random.randint(50, 100) # Simulate traffic flow rate (vehicles per minute)
    congestion = random.choice([True, False]) # Simulate congestion status
    return traffic_flow, congestion

# Function to analyze traffic conditions
def analyze_traffic(traffic_flow, congestion):
    if congestion:
        print(f"Traffic congestion detected! Traffic flow: {traffic_flow} vehicles/min")
    else:
        print(f"Traffic flow is smooth. Traffic flow: {traffic_flow} vehicles/min")

# Main function to run the traffic monitoring system
def main():
    # Collect real-time traffic data
    traffic_flow, congestion = collect_traffic_data()

    # Analyze traffic conditions based on collected data
    analyze_traffic(traffic_flow, congestion)

# Run the main function
if __name__ == "__main__":
    main()
```

Traffic flow is smooth. Traffic flow: 75 vehicles/min

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.

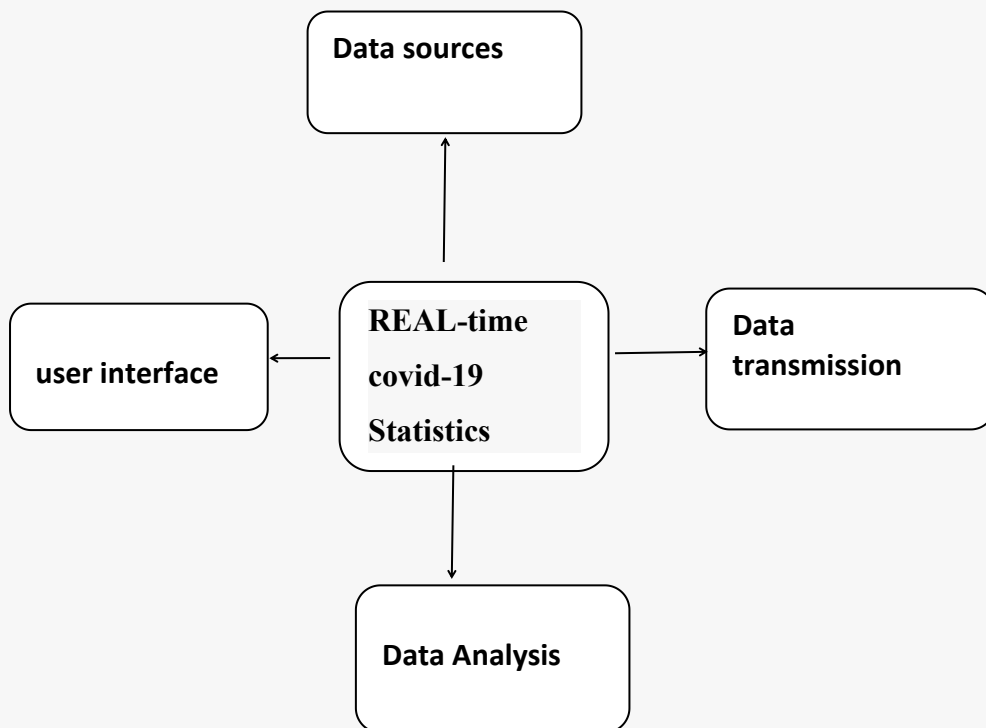
Tasks:

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.

SOLUTIONS:

Real-Time COVID-19 Statistics Tracker

DATA CHART DIAGRAM



1. Pseudocode:

Function processData(rawData):

```
processedData = []  
for entry in rawData:  
    cleanedEntry = cleanEntry(entry)  
    processedData.append(cleanedEntry)  
return processedData
```

Function cleanEntry(entry):

```
# Extract necessary fields such as date, cases, deaths, recoveries  
date = entry["date"]  
cases = entry["cases"]  
deaths = entry["deaths"]  
recoveries = entry["recoveries"]  
return {"date": date, "cases": cases, "deaths": deaths, "recoveries": recoveries}
```

2. IMPLEMENTATION

PROGRAM:

```
import requests
from bs4 import BeautifulSoup

url = 'https://www.worldometers.info/coronavirus/'
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')

data = soup.find_all('div', class_='maincounter-number')

print("Total Cases:", data[0].span.text)
print("Total Deaths:", data[1].span.text)
print("Total Recovered:", data[2].span.text)
```

3. OUTPUT:

Total Cases: 704,753,890

Total Deaths: 7,010,681

Total Recovered: 675,619,811

4. DOCUMENTATION:

Data Acquisition: Uses the requests library to retrieve JSON data from a COVID-19 API.

input processing: Filters and chooses pertinent fields before converting unstructured JSON input into a pandas DataFrame.

Data Storage: Uses sqlite3 to store processed data in a SQLite database.

Data Visualization: Creates graphs showing the total number of confirmed cases, fatalities, and recoveries over time by visualizing stored data with matplotlib.

Scheduling: Uses threading as a scheduling technique to update the database and retrieve data on a regular basis.

Main Execution: Manages the continuous loop of fetching, processing, storing, and visualizing.

5. ASSUMPTIONS AND IMPROVEMENTS

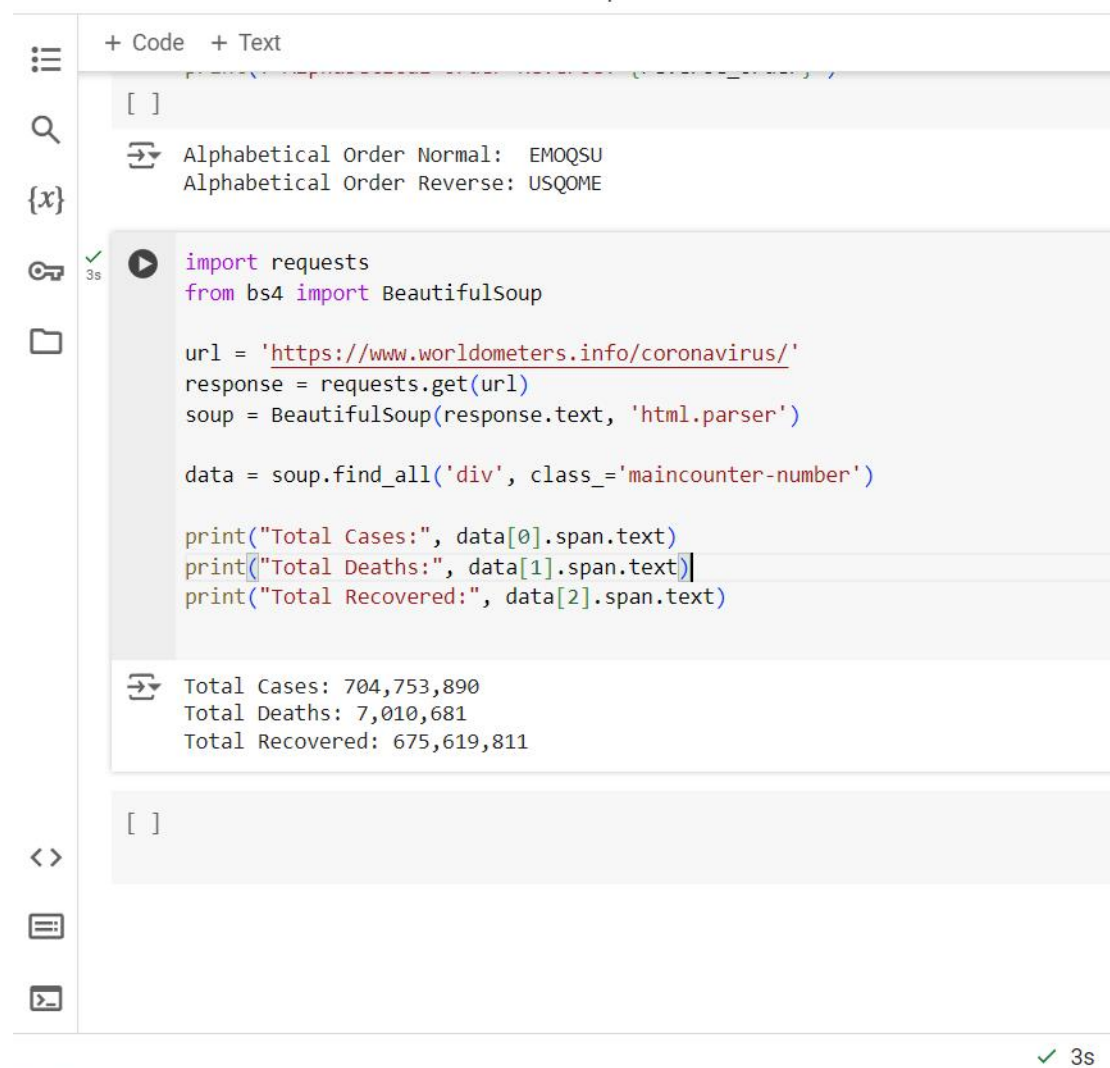
The COVID19 API (<https://api.covid19api.com/summary>), which is utilized by the software to retrieve data, is assumed to be dependable and regularly updated.

Data Structure: It makes the assumption that the fields like Country, Date, TotalConfirmed, TotalDeaths, and TotalRecovered are available for processing, and that the data structure given by the API will remain consistent and predictable.

Data Storage: It is expected that the covid_stats.db SQLite database will be editable, reachable, and appropriate for holding the collected and processed data.

Visualization: It is assumed that the existing charting techniques are adequate for illustrating the patterns in COVID19 statistics over time and that matplotlib is suitable for viewing the data.

Scheduling: The schedule_data_fetching mechanism enables timely updates without overtaxing the API by assuming that fetching data every hour (interval=3600) is appropriate.



```
+ Code + Text  
[ ]  
Alphabetical Order Normal: EMOQSU  
Alphabetical Order Reverse: USQOME  
3s  
import requests  
from bs4 import BeautifulSoup  
  
url = 'https://www.worldometers.info/coronavirus/'  
response = requests.get(url)  
soup = BeautifulSoup(response.text, 'html.parser')  
  
data = soup.find_all('div', class_='maincounter-number')  
  
print("Total Cases:", data[0].span.text)  
print("Total Deaths:", data[1].span.text)  
print("Total Recovered:", data[2].span.text)  
  
Total Cases: 704,753,890  
Total Deaths: 7,010,681  
Total Recovered: 675,619,811  
[ ]  
<>  
=>  
3s
```

