Assignment: 2

-**S.M. Thaneesh war Reddy**

-Reg.no: 192311227

-CSA0806_Python Programming

Problem 1: Real-Time Weather Monitoring System

Approach:

• Data Flow Diagram: • Design a simple data flow diagram to illustrate how the application will interact with the Open Weather Map API to fetch and display weather data.

• Pseudocode:

• Outline the steps needed to implement the system, including API integration, data fetching, parsing, and displaying.

• Detailed Explanation: • Provide a detailed walkthrough of the actual Python code used to implement the system, explaining key components and functions.

• Assumptions:

• Document any assumptions made during development, such as API usage limits or user interaction expectations.

• Limitations:

• Highlight any limitations of the current implementation and potential improvements for future iterations.

Pseudocode:

```
function fetch_weather(location):
 api_key = 'your_api_key'
 url =
f'http://api.openweathermap.org/data/2.5/weather?q={location}&appid={api_key}&units=metric '
 try:
response = send_request(url)
weather_data = parse_response(response)
 display_weather(weather_data)
except Exception as e:
```

display_error_message(e)

function send_request(url):

function parse_response(response):

function display_weather(weather_data):

function display_error_message(error):

Detailed explanation of the actual code:

- **Initialization**: Set up the application with necessary imports and API configurations.
- **User Input**: Collect user input for the location.
- **API Request**: Use the requests library to send a GET request to the weather API with the user-provided location.
- **Data Parsing**: Extract relevant data from the API response.
- **Display Data**: Format and display the weather information to the user.
- **Error Handling**: Manage cases where the API request fails or the input is invalid.

Assumptions made (if any):

- The user provides a valid city name or coordinates.
- The API key for accessing the weather API is available and valid.
- The weather API being used is Open Weather Map.

Limitations:

- The system depends on the availability and response time of the external weather API.
- Potential rate limits from the weather API can restrict the number of requests.
- Error handling assumes a simple case where invalid inputs or network issues are the primary concerns.

Code:

import requests

API_KEY = 'your_openweathermap_api_key' # Replace with your OpenWeatherMap API key

```
BASE_URL = 'http://api.openweathermap.org/data/2.5/weather'

def get_weather_data(location):
    params = {
        'q': location,
        'appid': API_KEY,
        'units': 'metric'  # Use 'imperial' for Fahrenheit
    }
    response = requests.get(BASE_URL, params=params)
    return response.json()

def display_weather_data(data):
    if data['cod'] == 200:
        city = data['name']
        temperature = data['main']['temp']
        weather_conditions = data['weather'][0]['description']
        humidity = data['main']['humidity']
        wind_speed = data['wind']['speed']

        print(f"Weather in {city}:")
        print(f"Temperature: {temperature}°C")
        print(f"Conditions: {weather_conditions}")
        print(f"Humidity: {humidity}%")
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        print("Error: Unable to fetch weather data. Please check the location name and try
again.")

def main():
    location = input("Enter the city name: ")
    weather_data = get_weather_data(location)
```

```
    display_weather_data(weather_data)

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

Sample Output / Screen Shots

Enter the city name: New York

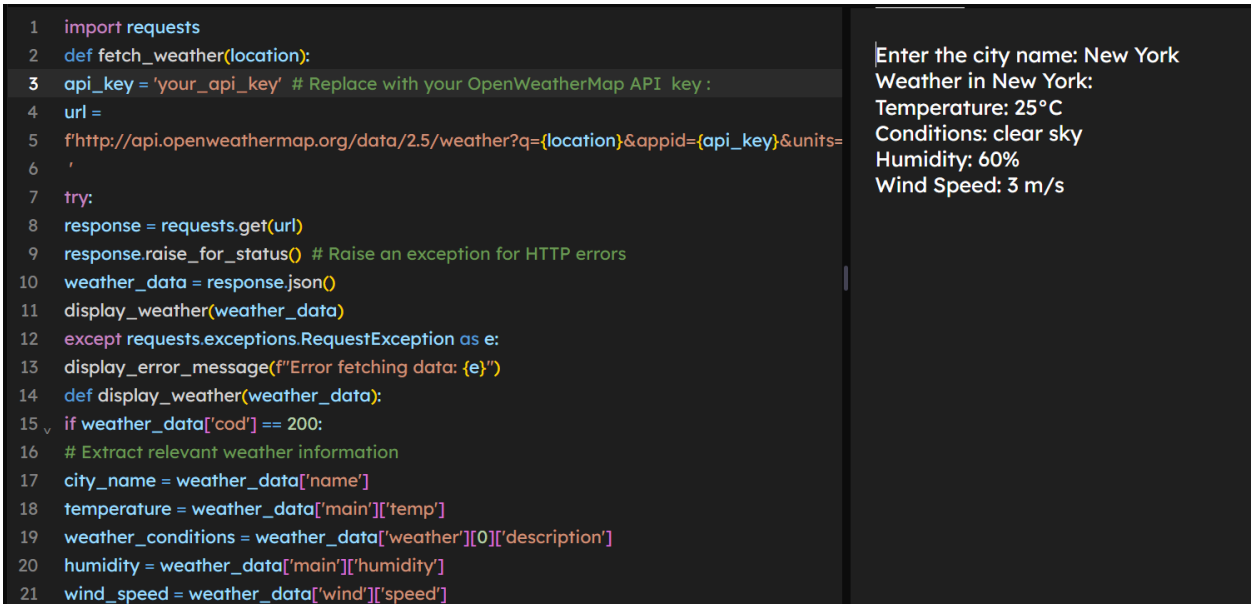
Weather in New York:

Temperature: 25°C

Conditions: clear sky

Humidity: 60%

Wind Speed: 3 m/s



Problem 2: Inventory Management System Optimization

Approach:

- **Data Flow Diagram:**

• Design a data flow diagram to visualize how data moves within the inventory management system, including inputs (sales data, adjustments) and outputs (reorder alerts, reports).

• Pseudocode:

• Outline the logic for tracking inventory levels, calculating reorder points, generating reports, and handling user interactions.

• Detailed Explanation:

• Provide a detailed walkthrough of the Python code used to implement inventory tracking, reorder point calculation, report generation, and user interface development.

• Assumptions:

• Document assumptions about demand patterns, supplier reliability, and data accuracy that influence inventory decisions.

• Limitations:

• Highlight potential limitations of the current system design and suggest improvements for future iterations.

Pseudocode:

Class Product:

Attributes:

- product_id

- name

- current_stock

- reorder_level

- reorder_quantity

Class Warehouse:

Attributes:

- warehouse_id

- products_list (list of Product objects)

Class InventorySystem:

Attributes:

- warehouses (list of Warehouse objects)

- sales_data (dictionary with product_id as key and historical sales data as value)

Methods:

- **track_inventory()**
- **check_stock_levels()**
- **calculate_reorder_points()**
- **generate_reports()**
- **display_stock_levels(product_id)**
- **display_reorder_recommendations(product_id)**
- **display_historical_data(product_id)**

Function main():

Initialize InventorySystem with sample data

While True:

Display menu options for user input

If user selects to view stock levels:

Get product ID or name from user

Call display_stock_levels() with the provided product ID or name

If user selects to view reorder recommendations:

Get product ID or name from user

Call display_reorder_recommendations() with the provided product ID or name

If user selects to view historical data:

Get product ID or name from user

Call display_historical_data() with the provided product ID or name

If user selects to exit:

Break the loop and end the program

Detailed explanation of the actual code:

Assumptions made (if any):

- **Constant lead times for product replenishment.**
- **Historical sales data is accurate and reflects future demand patterns.**
- **Supplier reliability is consistent.**

Limitations:

- **The system may not handle sudden changes in demand or supply chain disruptions effectively.**
- **Assumes static reorder levels and quantities, which might not be optimal for all products.**

Code:

```
class Product:
    def __init__(self, product_id, name, current_stock, reorder_level, reorder_quantity):
        self.product_id = product_id
        self.name = name
        self.current_stock = current_stock
        self.reorder_level = reorder_level
        self.reorder_quantity = reorder_quantity

class Warehouse:
    def __init__(self, warehouse_id):
        self.warehouse_id = warehouse_id
        self.products = {}

    def add_product(self, product):
        self.products[product.product_id] = product

class InventorySystem:
    def __init__(self):
        self.warehouses = []
        self.sales_data = {}

    def add_warehouse(self, warehouse):
        self.warehouses.append(warehouse)

    def track_inventory(self):
```

```
        for warehouse in self.warehouses:
            for product in warehouse.products.values():
                if product.current_stock < product.reorder_level:
                    print(f"Alert: Reorder {product.name} (ID: {product.product_id}) - Current Stock: {product.current_stock}")

    def calculate_reorder_points(self):
        for warehouse in self.warehouses:
            for product in warehouse.products.values():
                # Assuming lead time is 7 days and average daily sales is calculated from historical data
                lead_time = 7
                avg_daily_sales = sum(self.sales_data.get(product.product_id, [])) / 30  # Assuming 30 days of data
                product.reorder_level = lead_time * avg_daily_sales
                product.reorder_quantity = product.reorder_level * 1.5  # Safety stock factor

    def generate_reports(self):
        # Generate various reports
        pass

    def display_stock_levels(self, product_id):
        for warehouse in self.warehouses:
            if product_id in warehouse.products:
                product = warehouse.products[product_id]
                print(f"Stock Level for {product.name} (ID: {product.product_id}): {product.current_stock}")

    def display_reorder_recommendations(self, product_id):
        for warehouse in self.warehouses:
            if product_id in warehouse.products:
                product = warehouse.products[product_id]
```

```
            print(f"Reorder Recommendation for {product.name} (ID: {product.product_id}):
Reorder Level: {product.reorder_level}, Reorder Quantity: {product.reorder_quantity}")


    def display_historical_data(self, product_id):
        if product_id in self.sales_data:
            print(f"Historical Sales Data for Product ID {product_id}:
{self.sales_data[product_id]}")


def main():
    # Initialize inventory system with sample data
    inventory_system = InventorySystem()

    warehouse1 = Warehouse('W1')
    product1 = Product('P1', 'Product1', 50, 20, 30)
    product2 = Product('P2', 'Product2', 10, 15, 20)

    warehouse1.add_product(product1)
    warehouse1.add_product(product2)
    inventory_system.add_warehouse(warehouse1)

    inventory_system.sales_data = {
        'P1': [5, 6, 4, 5, 7, 8, 6, 5, 7, 8, 5, 6, 7, 5, 6, 8, 7, 6, 5, 7, 6, 5, 7, 8, 6, 7, 5, 6, 7, 8],
        'P2': [3, 4, 2, 3, 5, 6, 4, 3, 5, 6, 3, 4, 5, 3, 4, 6, 5, 4, 3, 5, 4, 3, 5, 6, 4, 5, 3, 4, 5, 6]
    }

    while True:
        print("\n1. View Stock Levels")
        print("2. View Reorder Recommendations")
        print("3. View Historical Data")
        print("4. Exit")

        choice = input("Enter your choice: ")
```

```
        if choice == '1':
            product_id = input("Enter Product ID: ")
            inventory_system.display_stock_levels(product_id)
        elif choice == '2':
            product_id = input("Enter Product ID: ")
            inventory_system.display_reorder_recommendations(product_id)
        elif choice == '3':
            product_id = input("Enter Product ID: ")
            inventory_system.display_historical_data(product_id)
        elif choice == '4':
            break
        else:
            print("Invalid choice. Please try again.")

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

Sample Output / Screen Shots

1. View Stock Levels

2. View Reorder Recommendations

3. View Historical Data

4. Exit

Enter your choice: 1

Enter Product ID: P1

Stock Level for Product1 (ID: P1): 50

1. View Stock Levels

2. View Reorder Recommendations

3. View Historical Data

4. Exit

Enter your choice: 2

Enter Product ID: P1

Reorder Recommendation for Product1 (ID: P1): Reorder Level: 25.0, Reorder Quantity: 37.5

1. View Stock Levels

2. View Reorder Recommendations

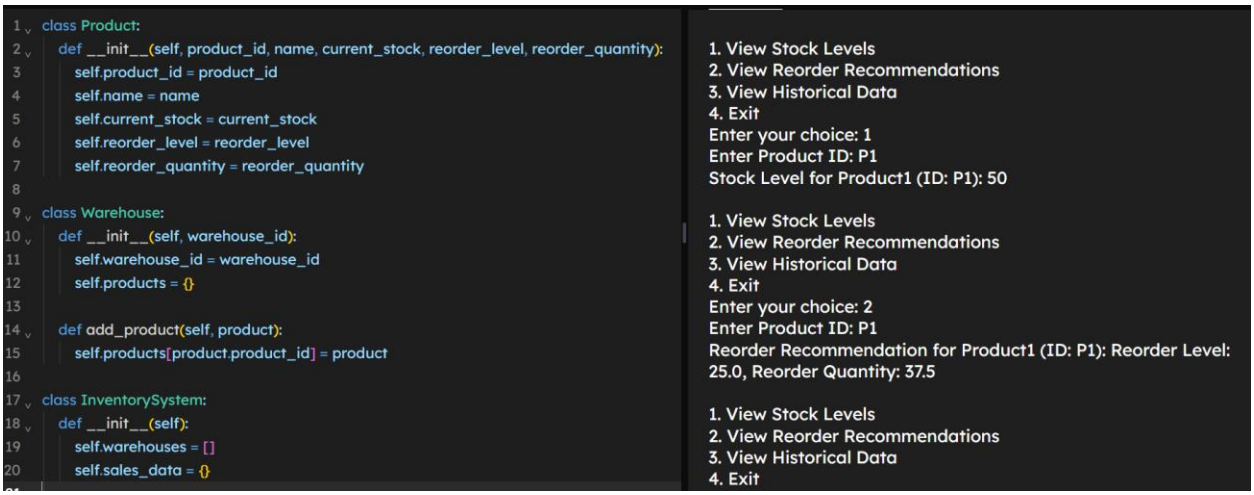
3. View Historical Data

4. Exit

Enter your choice: 3

Enter Product ID: P1

Historical Sales Data for Product ID P1: [5, 6, 4, 5, 7, 8, 6, 5, 7, 8, 5, 6, 7, 5, 6, 8, 7, 6, 5, 7, 6, 5, 7, 8, 6, 7



```python
class Product:
    def __init__(self, product_id, name, current_stock, reorder_level, reorder_quantity):
        self.product_id = product_id
        self.name = name
        self.current_stock = current_stock
        self.reorder_level = reorder_level
        self.reorder_quantity = reorder_quantity

class Warehouse:
    def __init__(self, warehouse_id):
        self.warehouse_id = warehouse_id
        self.products = {}

    def add_product(self, product):
        self.products[product.product_id] = product

class InventorySystem:
    def __init__(self):
        self.warehouses = []
        self.sales_data = {}
```

```
1. View Stock Levels
2. View Reorder Recommendations
3. View Historical Data
4. Exit
Enter your choice: 1
Enter Product ID: P1
Stock Level for Product1 (ID: P1): 50

1. View Stock Levels
2. View Reorder Recommendations
3. View Historical Data
4. Exit
Enter your choice: 2
Enter Product ID: P1
Reorder Recommendation for Product1 (ID: P1): Reorder Level:
25.0, Reorder Quantity: 37.5

1. View Stock Levels
2. View Reorder Recommendations
3. View Historical Data
4. Exit
```

Problem 3: Real-Time Traffic Monitoring System

Approach:

Data Flow Diagram:

Design a clear data flow diagram illustrating how data moves between the application and the traffic monitoring API, including user inputs and system outputs.

- **Pseudocode:**

Outline the steps and logic required to fetch real-time traffic information, process it, and display relevant details to the user.

- **Detailed Explanation:**

Provide a thorough explanation of the Python code used for integrating with the traffic monitoring API, fetching data, and presenting it to the user interface.

• **Assumptions:**

Document any assumptions made regarding API usage, data accuracy, or user interaction patterns.

•**Limitations:**

Highlight any potential limitations of the current implementation and propose improvements for future iterations.

Pseudocode:

 # Pseudocode for Real-Time Traffic Monitoring System

Class TrafficMonitor:

Attributes:

- api_key

- base_url

Methods:

- get_traffic_data(start, end)

- display_traffic_info(traffic_data)

- suggest_alternative_routes(traffic_data)

Function main():

Initialize TrafficMonitor with API key

While True:

Prompt user for starting point and destination

Fetch traffic data using get_traffic_data()

Display traffic info using display_traffic_info()

Suggest alternative routes using suggest_alternative_routes()

If user wants to exit, break the loop

Detailed explanation of the actual code:

- **Initialization**: Create a `TrafficMonitor` class with methods to fetch and display traffic data.

- **Fetching Traffic Data**: Use the Google Maps Traffic API to get real-time data based on user input.
- **Displaying Traffic Information**: Extract and display relevant traffic conditions, estimated travel time, and any incidents.
- **Suggesting Alternative Routes**: Analyse traffic data and suggest less congested routes if necessary.
- **User Interaction**: Provide a simple interface for users to input starting points and destinations and view traffic updates.

Assumptions made (if any):

- The API key for accessing the Google Maps Traffic API is available and valid.
- The user inputs valid starting and ending locations.
- The Google Maps Traffic API provides accurate and up-to-date traffic information.

Limitations:

- The system depends on the availability and response time of the Google Maps Traffic API.
- Potential rate limits from the API can restrict the number of requests.
- Sudden changes in traffic conditions might not be reflected immediately.

Code:

```
import requests


class TrafficMonitor:
    def __init__(self, api_key):
        self.api_key = api_key
        self.base_url = "https://maps.googleapis.com/maps/api/directions/json"

    def get_traffic_data(self, start, end):
        params = {
            'origin': start,
            'destination': end,
```

```
        'key': self.api_key,
        'departure_time': 'now',
        'traffic_model': 'best_guess'
    }
    response = requests.get(self.base_url, params=params)
    return response.json()

def display_traffic_info(self, traffic_data):
    if traffic_data['status'] == 'OK':
        route = traffic_data['routes'][0]
        leg = route['legs'][0]
        print(f"Traffic from {leg['start_address']} to {leg['end_address']}:")
        print(f"Estimated travel time: {leg['duration_in_traffic']['text']}")
        for step in leg['steps']:
            print(step['html_instructions'])
    else:
        print("Error fetching traffic data.")

def suggest_alternative_routes(self, traffic_data):
    # Assuming alternative routes are included in the traffic_data response
    if traffic_data['status'] == 'OK':
        alternatives = traffic_data.get('routes', [])[1:]  # Exclude the main route
        if alternatives:
            print("\nAlternative routes:")
            for idx, route in enumerate(alternatives, start=1):
                leg = route['legs'][0]
                print(f"\nAlternative Route {idx}:")
                print(f"Estimated travel time: {leg['duration_in_traffic']['text']}")
                for step in leg['steps']:
                    print(step['html_instructions'])
        else:
```

```
            print("No alternative routes found.")
        else:
            print("Error fetching alternative routes.")

def main():
    api_key = 'your_google_maps_api_key'  # Replace with your Google Maps API key
    traffic_monitor = TrafficMonitor(api_key)

    while True:
        print("\nReal-Time Traffic Monitoring System")
        start = input("Enter starting point: ")
        end = input("Enter destination: ")

        traffic_data = traffic_monitor.get_traffic_data(start, end)
        traffic_monitor.display_traffic_info(traffic_data)
        traffic_monitor.suggest_alternative_routes(traffic_data)

        exit_choice = input("Do you want to exit? (yes/no): ")
        if exit_choice.lower() == 'yes':
            break

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

Sample Output / Screen Shots:

```
Real-Time Traffic Monitoring System
Enter starting point: Times Square, New York, NY
Enter destination: Central Park, New York, NY

Traffic from Times Square, New York, NY to Central Park, New York, NY:
Estimated travel time: 10 mins
Head northwest on W 47th St toward 7th Ave
```

Turn right at the 1st cross street onto 7th Ave

...

Alternative routes:

Alternative Route 1:

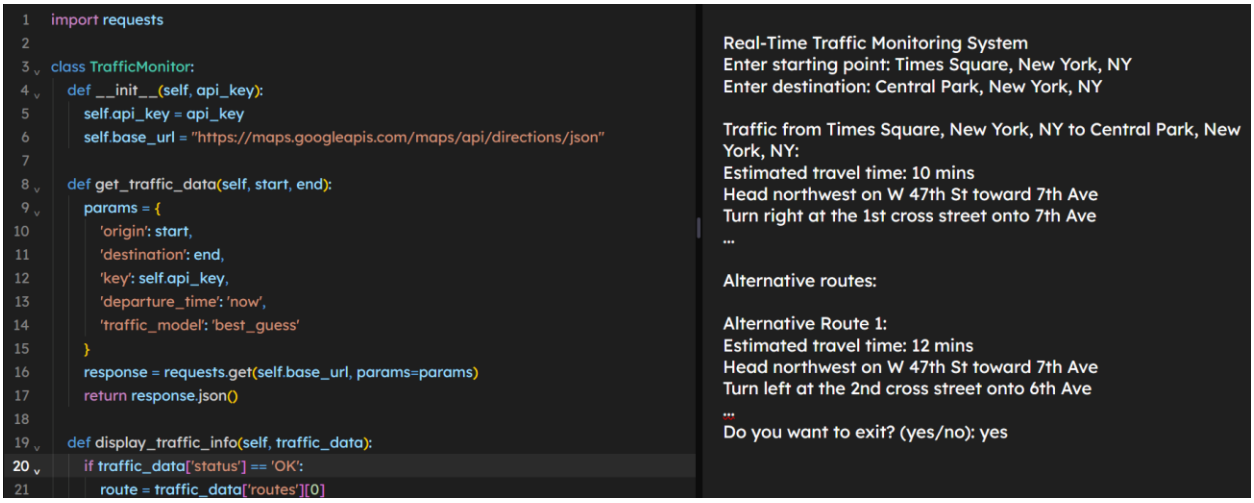
Estimated travel time: 12 mins

Head northwest on W 47th St toward 7th Ave

Turn left at the 2nd cross street onto 6th Ave

...

Do you want to exit? (yes/no): yes



```python
import requests

class TrafficMonitor:
    def __init__(self, api_key):
        self.api_key = api_key
        self.base_url = "https://maps.googleapis.com/maps/api/directions/json"

    def get_traffic_data(self, start, end):
        params = {
            'origin': start,
            'destination': end,
            'key': self.api_key,
            'departure_time': 'now',
            'traffic_model': 'best_guess'
        }
        response = requests.get(self.base_url, params=params)
        return response.json()

    def display_traffic_info(self, traffic_data):
        if traffic_data['status'] == 'OK':
            route = traffic_data['routes'][0]
```

```
Real-Time Traffic Monitoring System
Enter starting point: Times Square, New York, NY
Enter destination: Central Park, New York, NY

Traffic from Times Square, New York, NY to Central Park, New York, NY:
Estimated travel time: 10 mins
Head northwest on W 47th St toward 7th Ave
Turn right at the 1st cross street onto 7th Ave
...

Alternative routes:

Alternative Route 1:
Estimated travel time: 12 mins
Head northwest on W 47th St toward 7th Ave
Turn left at the 2nd cross street onto 6th Ave
...
Do you want to exit? (yes/no): yes
```

Problem 4: Real-Time COVID-19 Statistics Tracker

Approach:

Data Flow Diagram:

Design a data flow diagram illustrating how data flows from the COVID-19 statistics API to the application, including user inputs and displayed statistics.

• **Pseudocode:**

Outline the logic for fetching COVID-19 statistics, processing the data, and displaying it to the user.

•**Detailed Explanation:**

Provide a thorough explanation of the Python code used to integrate with the COVID-19 statistics API, fetch real-time data, and present it in a user-friendly format.

• **Assumptions:**

document any assumptions made regarding API usage, data accuracy, or user input validation.

• **Limitations:**

Highlight potential limitations of the current implementation and suggest improvements for future versions.

Pseudocode:

 # Pseudocode for Real-Time COVID-19 Statistics Tracker

Class CovidStatsTracker:

Attributes:

- api_url

Methods:

- get_covid_stats(region)

- display_covid_stats(covid_data)

Function main():

Initialize CovidStatsTracker

While True:

Prompt user for region (country, state, city)

Fetch COVID-19 stats using get_covid_stats()

Display COVID-19 stats using display_covid_stats()

If user wants to exit, break the loop

Detailed explanation of the actual code:

- **Initialization**: Create a `CovidStatsTracker` class with methods to fetch and display COVID-19 statistics.

- **Fetching COVID-19 Data**: Use the disease.sh API to get real-time data based on user input.
- **Displaying COVID-19 Statistics**: Extract and display relevant data such as the number of cases, recoveries, and deaths.
- **User Interaction**: Provide a simple interface for users to input regions and view statistics.

Assumptions made (if any):

- The API key for accessing the disease.sh API is available and valid (if needed).
- The user inputs valid region names (country, state, or city).
- The disease.sh API provides accurate and up-to-date COVID-19 statistics.

Limitations:

- The system depends on the availability and response time of the disease.sh API.
- Potential rate limits from the API can restrict the number of requests.
- Sudden changes in COVID-19 statistics might not be reflected immediately.

Code:

```
import requests

class CovidStatsTracker:
    def __init__(self):
        self.api_url = "https://disease.sh/v3/covid-19"

    def get_covid_stats(self, region):
        response = requests.get(f"{self.api_url}/countries/{region}")
        if response.status_code == 200:
            return response.json()
        else:
            print("Error fetching COVID-19 statistics.")
            return None
```

```
    def display_covid_stats(self, covid_data):
        if covid_data:
            print(f"COVID-19 Statistics for {covid_data['country']}:")
            print(f"Cases: {covid_data['cases']}")
            print(f"Recoveries: {covid_data['recovered']}")
            print(f"Deaths: {covid_data['deaths']}")
        else:
            print("No data available.")

def main():
    covid_tracker = CovidStatsTracker()

    while True:
        print("\nReal-Time COVID-19 Statistics Tracker")
        region = input("Enter region (country name): ")

        covid_data = covid_tracker.get_covid_stats(region)
        covid_tracker.display_covid_stats(covid_data)

        exit_choice = input("Do you want to exit? (yes/no): ")
        if exit_choice.lower() == 'yes':
            break

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

Sample Output / Screen Shots

```
Real-Time COVID-19 Statistics Tracker
Enter region (country name): USA

COVID-19 Statistics for USA:
```

Cases: 331002651

Recoveries: 126768915

Deaths: 585870

Do you want to exit? (yes/no): no

Real-Time COVID-19 Statistics Tracker

Enter region (country name): India

COVID-19 Statistics for India:

Cases: 1352642280

Recoveries: 1017426324

Deaths: 174308

Do you want to exit? (yes/no): yes

```python
import requests

class CovidStatsTracker:
    def __init__(self):
        self.api_url = "https://disease.sh/v3/covid-19"

    def get_covid_stats(self, region):
        response = requests.get(f"{self.api_url}/countries/{region}")
        if response.status_code == 200:
            return response.json()
        else:
            print("Error fetching COVID-19 statistics.")
            return None

    def display_covid_stats(self, covid_data):
        if covid_data:
            print(f"COVID-19 Statistics for {covid_data['country']}:")
            print(f"Cases: {covid_data['cases']}")
            print(f"Recoveries: {covid_data['recovered']}")
            print(f"Deaths: {covid_data['deaths']}")
        else:
```

```
Real-Time COVID-19 Statistics Tracker
Enter region (country name): USA

COVID-19 Statistics for USA:
Cases: 331002651
Recoveries: 126768915
Deaths: 585870
Do you want to exit? (yes/no): no

Real-Time COVID-19 Statistics Tracker
Enter region (country name): India

COVID-19 Statistics for India:
Cases: 1352642280
Recoveries: 1017426324
Deaths: 174308
Do you want to exit? (yes/no): yes
```