**Assignment: 2**

-**P. Jawed khan**

**-Reg.no: 192311263**

**-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**

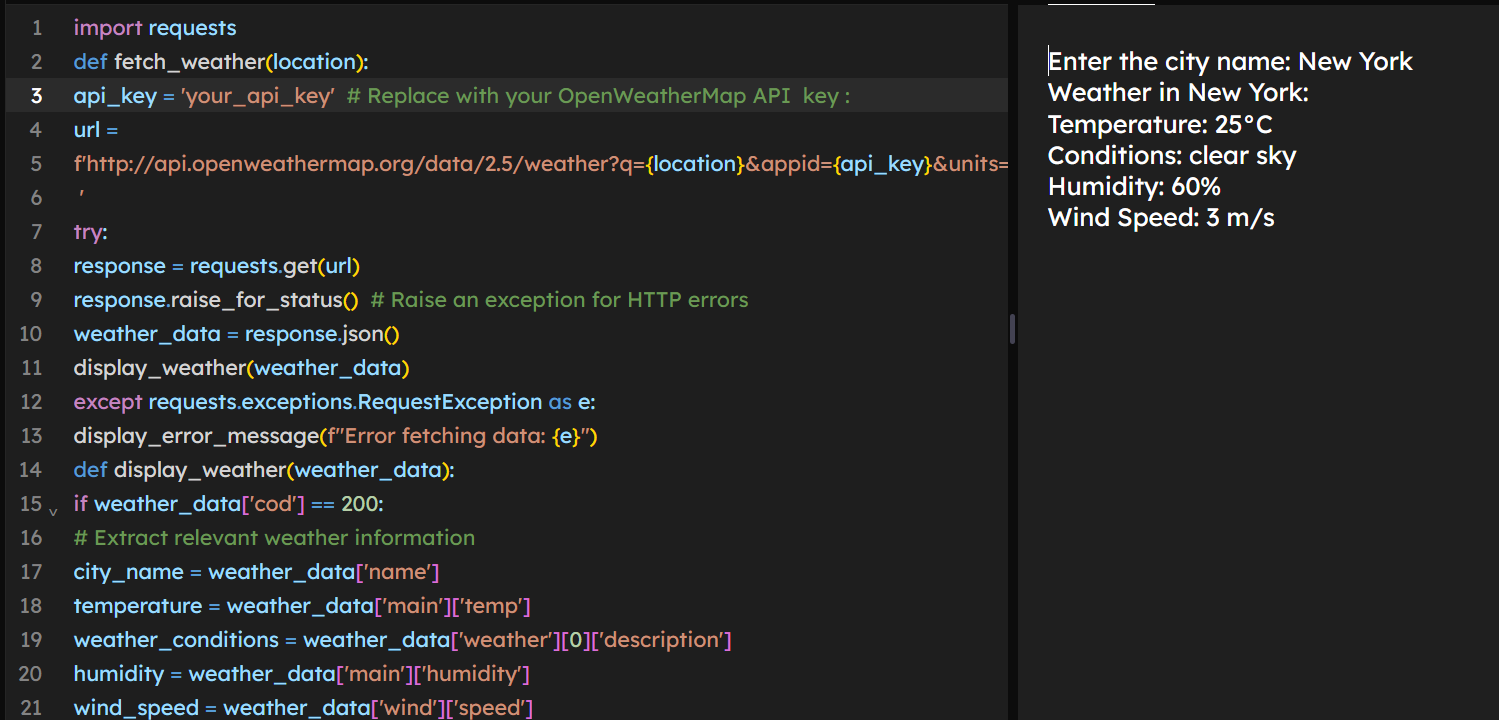
**Weather in New York:**

**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**

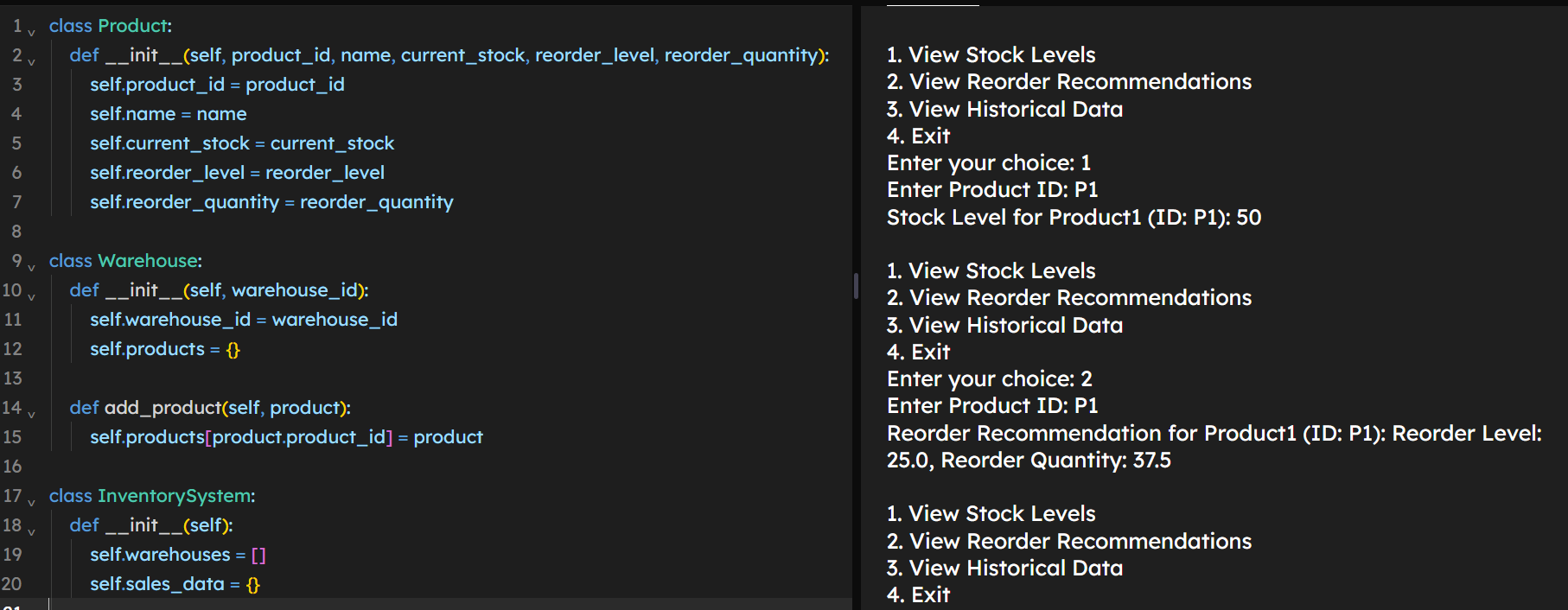
**3. View Historical Data**

**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**



**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:**

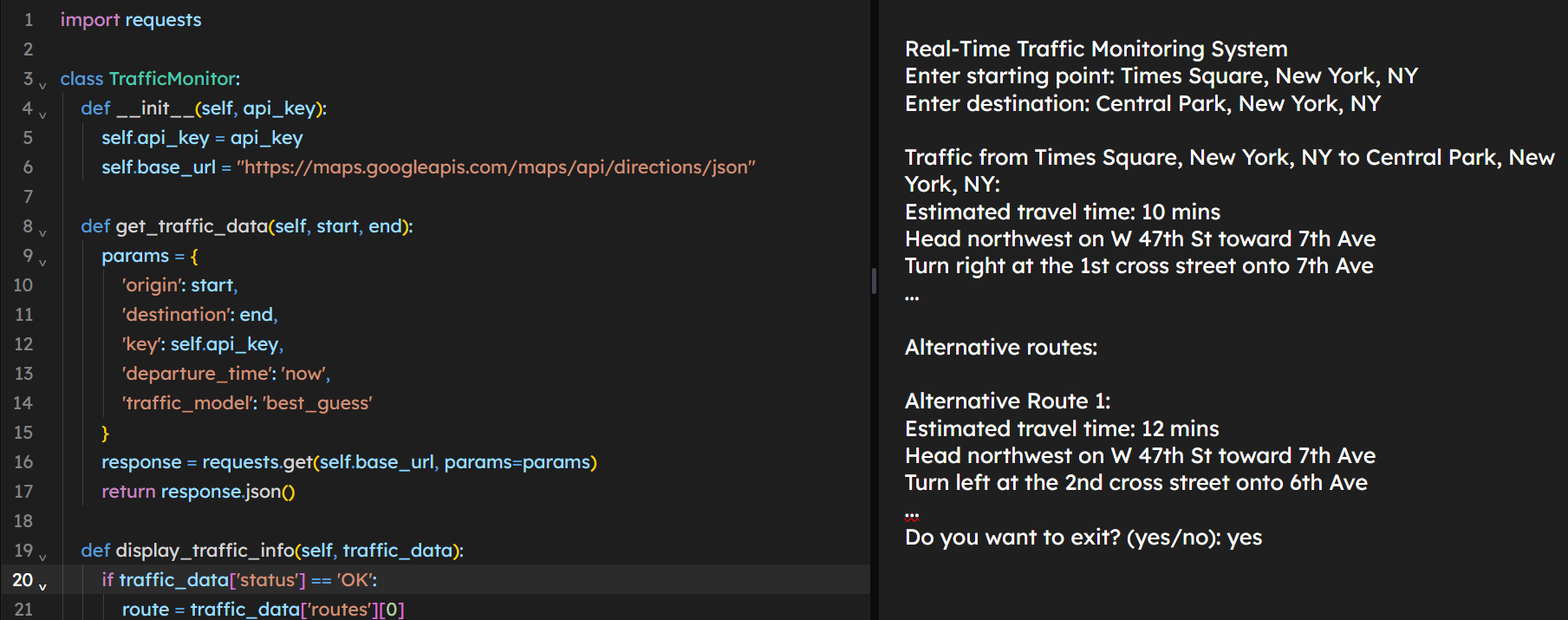
**Estimated travel time: 12 mins**

**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**



**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**

