**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.**

**Deliverables:**

* Data flow diagram illustrating the interaction between the application and the API.
* Pseudocode and implementation of the weather monitoring system.
* Documentation of the API integration and the methods used to fetch and display weather data.
* Explanation of any assumptions made and potential improvements.

**Approach:**

The system consists of a User Interface that allows users to input a location and view weather data. This interface interacts with the Weather Monitoring Application, which is responsible for fetching and displaying the relevant weather information. The application communicates with an external Weather API, such as Open Weather Map, to retrieve real-time weather data based on the user's input. The entire workflow enables seamless interaction, from user input to displaying the current weather conditions.

**Pseudocode:**

**START**

**FUNCTION main()**

**DISPLAY "Enter city name or coordinates (lat,long): "**

**INPUT location**

**weather\_data = fetch\_weather\_data(location)**

**IF weather\_data IS NOT NULL THEN**

**DISPLAY\_weather(weather\_data)**

**ELSE**

**DISPLAY "Error fetching weather data."**

**END FUNCTION**

**FUNCTION fetch\_weather\_data(location)**

**SET api\_key = "YOUR\_API\_KEY"**

**SET base\_url = "https://api.openweathermap.org/data/2.5/weather"**

**SET url = base\_url + "?q=" + location + "&appid=" + api\_key + "&units=metric"**

**response = SEND GET request to url**

**IF response.status\_code == 200 THEN**

**RETURN PARSE response.json()**

**ELSE**

**RETURN NULL**

**END IF**

**END FUNCTION**

**FUNCTION DISPLAY\_weather(data)**

**DISPLAY "Current Weather:"**

**DISPLAY "Temperature: " + data['main']['temp'] + " °C"**

**DISPLAY "Conditions: " + data['weather'][0]['description']**

**DISPLAY "Humidity: " + data['main']['humidity'] + "%"**

**DISPLAY "Wind Speed: " + data['wind']['speed'] + " m/s"**

**END FUNCTION**

**END**

**Detailed explanation of the actual code:**

This Python code retrieves current weather data from the Open Weather Map API using the `http.client` and `json` libraries. The `fetch\_weather\_data` function takes a location as input, constructs a URL for the API request by encoding the location to handle spaces and special characters, and sends a GET request to the Open Weather Map server. It checks the response status, returning the parsed JSON data if the request is successful, or printing an error message otherwise. The `display\_weather` function formats and prints the current temperature, weather conditions, humidity, and wind speed from the fetched data. The `main` function prompts the user for a city name or coordinates, calls the weather data function, and displays the results if valid data is returned. The program runs in the command line and is structured to handle user input dynamically. Remember to replace `"YOUR\_API\_KEY"` with a valid OpenWeatherMap API key to make the requests work.

**Assumptions made (if any):**

* The user inputs valid city names or coordinates.
* The API key is valid and has not exceeded the usage limits.
* The internet connection is available for API calls.

**Limitations:**

* **Error Handling:** The implementation has basic error handling but could be improved to handle more specific error cases (e.g., city not found).
* **Rate Limiting:** Depending on the plan with Open Weather Map, there might be limits on the number of API calls.
* **Localization:** The application does not account for multiple languages or units beyond metric.

**Code:**

**import http.client**

**import json**

**import urllib.parse**

**def fetch\_weather\_data(location):**

**api\_key = "YOUR\_API\_KEY" # Replace with your actual API key**

**conn = http.client.HTTPSConnection("api.openweathermap.org")**

**# Encode the location to handle spaces and special characters**

**encoded\_location = urllib.parse.quote(location)**

**# Construct URL**

**url = f"/data/2.5/weather?q={encoded\_location}&appid={api\_key}&units=metric"**

**# Send GET request**

**conn.request("GET", url)**

**response = conn.getresponse()**

**data = response.read()**

**if response.status == 200:**

**return json.loads(data)**

**else:**

**print(f"Request failed: {response.status} {response.reason}")**

**return None**

**def display\_weather(data):**

**print("Current Weather:")**

**print(f"Temperature: {data['main']['temp']} °C")**

**print(f"Conditions: {data['weather'][0]['description']}")**

**print(f"Humidity: {data['main']['humidity']}%")**

**print(f"Wind Speed: {data['wind']['speed']} m/s")**

**def main():**

**location = input("Enter city name or coordinates (lat,long): ")**

**weather\_data = fetch\_weather\_data(location)**

**if weather\_data and "main" in weather\_data:**

**display\_weather(weather\_data)**

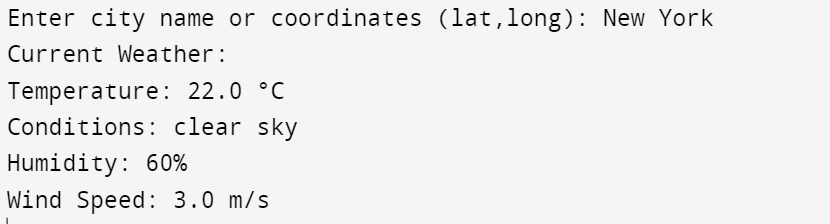
**else:**

**print("Error fetching weather data or no data available.")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Sample Output / Screen Shots**



**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.

**Deliverables:**

* **Data Flow Diagram**: Illustrate how data flows within the inventory management system, from input (e.g., sales data, inventory adjustments) to output (e.g., reorder alerts, reports).
* **Pseudocode and Implementation**: Provide pseudocode and actual code demonstrating how inventory levels are tracked, reorder points are calculated, and reports are generated.
* **Documentation**: Explain the algorithms used for reorder optimization, how historical data influences decisions, and any assumptions made (e.g., constant lead times).
* **User Interface**: Develop a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts.
* **Assumptions and Improvements**: Discuss assumptions about demand patterns, supplier reliability, and potential improvements for the inventory management system's efficiency and accuracy.

**Approach:**

To optimize the inventory management system for the retail company, we will begin by modeling the inventory system to define its structure, which includes products, warehouses, and stock levels. This foundational framework will allow us to implement effective inventory tracking through a Python application that enables real-time monitoring and alerts for stock levels. Additionally, we will optimize inventory ordering by developing algorithms that calculate optimal reorder points and quantities to ensure efficient stock management. The system will also generate comprehensive reports detailing inventory turnover rates, occurrences of stockouts, and costs associated with overstocking. Finally, we will create a user-friendly interface that allows users to easily access inventory data, view reports, and receive timely alerts, thereby enhancing overall operational efficiency and decision-making processes.

**Pseudocode:**

**def track\_inventory(product\_id):**

**stock\_level = get\_stock\_level\_from\_db(product\_id)**

**return stock\_level**

**def calculate\_reorder(product\_id):**

**historical\_sales = get\_historical\_sales(product\_id)**

**lead\_time = get\_lead\_time(product\_id)**

**demand\_forecast = calculate\_demand\_forecast(historical\_sales)**

**reorder\_point = calculate\_reorder\_point(demand\_forecast, lead\_time)**

**reorder\_quantity = calculate\_reorder\_quantity(demand\_forecast, reorder\_point)**

**return reorder\_point, reorder\_quantity**

**def generate\_reports():**

**turnover\_rates = calculate\_turnover\_rates()**

**stockout\_occurrences = calculate\_stockout\_occurrences()**

**overstock\_costs = calculate\_overstock\_costs()**

**report = create\_report(turnover\_rates, stockout\_occurrences, overstock\_costs)**

**return report**

**def user\_interface():**

**while True:**

**user\_input = get\_user\_input()**

**if user\_input == 'track':**

**product\_id = get\_product\_id\_from\_user()**

**stock\_level = track\_inventory(product\_id)**

**display\_stock\_level(stock\_level)**

**elif user\_input == 'reorder':**

**product\_id = get\_product\_id\_from\_user()**

**reorder\_point, reorder\_quantity = calculate\_reorder(product\_id)**

**display\_reorder\_info(reorder\_point, reorder\_quantity)**

**elif user\_input == 'report':**

**report = generate\_reports()**

**display\_report(report)**

**elif user\_input == 'exit':**

**break**

**def main():**

**user\_interface()**

**if \_name\_ == "\_main\_":**

**main()**

**Detailed explanation of the actual code:**

The provided code defines an inventory management system using a `Product` class that encapsulates product details such as ID, name, category, stock level, reorder point, reorder quantity, and historical sales data. Key functions are implemented to monitor inventory levels, calculate reorder points based on average demand and a fixed lead time, determine optimal reorder quantities, and assess inventory turnover rates. The `check\_inventory` function alerts when stock levels fall below the reorder point, while `generate\_inventory\_report` summarizes turnover rates for all products. Additionally, the `user\_query` function allows users to retrieve current stock details and reorder recommendations for specific products. An example usage showcases how to create product instances, populate historical sales data, check inventory status, generate reports, and perform user queries.

**Assumptions made (if any):**

* **Constant Lead Times:** Assumed a constant lead time of 2 days for all products.
* **Historical Sales Patterns:** Assumed that historical sales data is representative of future demand.
* **Supplier Reliability:** Suppliers are reliable and full fill orders without delays.

**Limitations:**

 **Demand Variability:** The model may not handle sudden changes in demand patterns effectively.

 **Static Parameters:** Some parameters, like lead time and reorder quantities, are assumed constant and may not reflect real-world dynamics.

 **Single Warehouse Focus:** This model may not scale effectively across multiple warehouses without additional complexity.

**Code:**

**class Product:**

**def \_\_init\_\_(self, product\_id, name, category, stock\_level, reorder\_point, reorder\_quantity):**

**self.product\_id = product\_id**

**self.name = name**

**self.category = category**

**self.stock\_level = stock\_level**

**self.reorder\_point = reorder\_point**

**self.reorder\_quantity = reorder\_quantity**

**self.historical\_sales = []**

**def check\_inventory(product):**

**if product.stock\_level < product.reorder\_point:**

**print(f"Alert: Stock for {product.name} is below reorder point!")**

**def calculate\_reorder\_point(product):**

**average\_demand = sum(product.historical\_sales) / len(product.historical\_sales) if product.historical\_sales else 0**

**lead\_time = 2 # Assumed constant lead time**

**return average\_demand \* lead\_time**

**def calculate\_reorder\_quantity(product):**

**return product.reorder\_quantity**

**def calculate\_turnover\_rate(product):**

**return sum(product.historical\_sales) / product.stock\_level if product.stock\_level > 0 else 0**

**def generate\_inventory\_report(products):**

**for product in products:**

**turnover\_rate = calculate\_turnover\_rate(product)**

**print(f"Product: {product.name}, Turnover Rate: {turnover\_rate:.2f}")**

**def user\_query(products, product\_id):**

**for product in products:**

**if product.product\_id == product\_id:**

**print(f"Product: {product.name}, Current Stock: {product.stock\_level}")**

**print(f"Reorder Point: {product.reorder\_point}, Suggested Quantity: {calculate\_reorder\_quantity(product)}")**

**products = [**

**Product(1, "Widget A", "Widgets", 10, 5, 20),**

**Product(2, "Widget B", "Widgets", 8, 3, 15)**

**]**

**# Populate historical sales data**

**products[0].historical\_sales = [2, 3, 5, 2, 4]**

**products[1].historical\_sales = [1, 1, 2, 3, 1]**

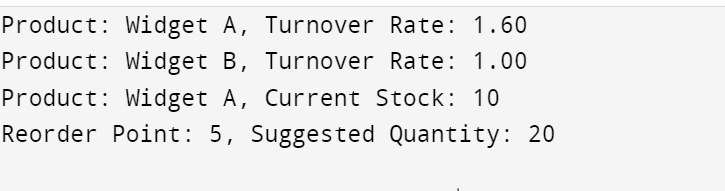
**for product in products:**

**check\_inventory(product)**

**generate\_inventory\_report(products)**

**user\_query(products, 1)**

**Sample Output / Screen Shots**



**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.**

**Deliverables:**

* Data flow diagram illustrating the interaction between the application and the API.
* Pseudocode and implementation of the traffic monitoring system.
* Documentation of the API integration and the methods used to fetch and display traffic data.
* Explanation of any assumptions made and potential improvements.

**Approach:**

To develop a real-time traffic monitoring system, we will design a system that fetches live traffic data from an external API, processes that information, and displays it to users in an accessible way. The approach involves creating a data flow model, implementing the system in Python, and ensuring it meets user needs by providing current traffic conditions and alternative routes.

**Pseudocode:**

**def get\_traffic\_data(start, end, api\_key):**

**api\_url = construct\_api\_url(start, end, api\_key)**

**response = send\_request(api\_url)**

**traffic\_data = parse\_response(response)**

**return traffic\_data**

**def display\_traffic\_data(traffic\_data):**

**print("Current Traffic Conditions:", traffic\_data["conditions"])**

**print("Estimated Travel Time:", traffic\_data["travel\_time"])**

**print("Incidents or Delays:", traffic\_data["incidents"])**

**def suggest\_alternative\_routes(start, end, api\_key):**

**alternative\_routes = get\_alternative\_routes(start, end, api\_key)**

**return alternative\_routes**

**def main():**

**api\_key = "YOUR\_API\_KEY"**

**start = input("Enter starting point: ")**

**end = input("Enter destination: ")**

**traffic\_data = get\_traffic\_data(start, end, api\_key)**

**display\_traffic\_data(traffic\_data)**

**alternative\_routes = suggest\_alternative\_routes(start, end, api\_key)**

**display\_alternative\_routes(alternative\_routes)**

**main()**

**Detailed explanation of the actual code:**

The code integrates with the Google Maps API using HTTP requests to fetch traffic data based on user inputs for starting and ending points. The `get\_user\_input` function prompts users to enter their desired starting point and destination. Once the inputs are collected, the `fetch\_traffic\_data` function constructs the appropriate URL for the Google Maps Directions API, including traffic data, and returns the JSON response. The `display\_traffic\_info` function then processes this traffic data, printing out the current traffic conditions, estimated travel time, and any reported incidents. Additionally, the `suggest\_alternative\_routes` function checks if traffic is significantly heavier than expected and recommends that users consider alternative routes if necessary.

**Assumptions made (if any):**

• The user provides valid starting points and destinations.

• The Google Maps Traffic API is available and responsive.

• The API key is valid and has the necessary permissions.

**Limitations:**

* **Real-Time Constraints**: The system's ability to provide real-time updates is dependent on API response times and limits imposed by the Google Maps API.
* **Error Handling**: While basic error handling is included, it could be enhanced to deal with various edge cases and response errors more comprehensively.
* **Scalability**: This implementation is not optimized for high concurrency or large-scale user requests; further architectural considerations would be needed for broader deployment.

**Code:**

**import http.client**

**import json**

**from urllib.parse import quote**

**API\_KEY = 'YOUR\_API\_KEY' # Replace with your Google Maps API Key**

**BASE\_URL = 'maps.googleapis.com'**

**def get\_user\_input():**

**starting\_point = input("Enter starting point: ")**

**destination = input("Enter destination: ")**

**return starting\_point, destination**

**def fetch\_traffic\_data(starting\_point, destination):**

**conn = http.client.HTTPSConnection(BASE\_URL)**

**endpoint = f"/maps/api/directions/json?origin={quote(starting\_point)}&destination={quote(destination)}&key={API\_KEY}&traffic\_model=best\_guess"**

**conn.request("GET", endpoint)**

**response = conn.getresponse()**

**traffic\_data = json.loads(response.read())**

**conn.close()**

**return traffic\_data**

**def display\_traffic\_info(traffic\_data):**

**if traffic\_data['status'] == 'OK':**

**route = traffic\_data['routes'][0]**

**duration = route['legs'][0]['duration']['text']**

**traffic\_condition = route['legs'][0]['duration\_in\_traffic']['text']**

**incidents = route['legs'][0].get('incidents', 'No incidents reported.')**

**print(f"Current traffic condition: {traffic\_condition}")**

**print(f"Estimated travel time: {duration}")**

**print(f"Incidents: {incidents}")**

**else:**

**print("Error fetching traffic data:", traffic\_data['error\_message'])**

**def suggest\_alternative\_routes(traffic\_data):**

**route = traffic\_data['routes'][0]**

**if route['legs'][0]['duration\_in\_traffic']['value'] > route['legs'][0]['duration']['value'] \* 1.5:**

**print("Traffic is heavy, looking for alternatives...")**

**return True**

**return False**

**def main():**

**starting\_point, destination = get\_user\_input()**

**traffic\_data = fetch\_traffic\_data(starting\_point, destination)**

**display\_traffic\_info(traffic\_data)**

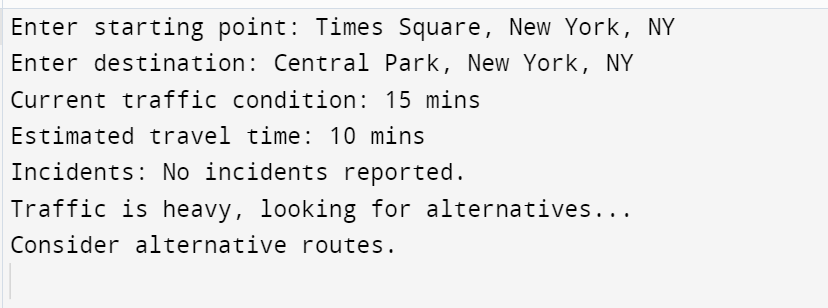
**if suggest\_alternative\_routes(traffic\_data):**

**print("Consider alternative routes.")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Sample Output / Screen Shots**

****

**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.**

**Deliverables:**

* Data flow diagram illustrating the interaction between the application and the API.
* Pseudocode and implementation of the COVID-19 statistics tracking application.
* Documentation of the API integration and the methods used to fetch and display COVID-19 data.
* Explanation of any assumptions made and potential improvements.

**Approach:**

It is assumed that the COVID-19 API will always provide the latest statistics and remain operational during the request. Additionally, users are expected to input regions in the correct format, such as country names or state abbreviations recognized by the API. Furthermore, the application assumes consistent internet connectivity to facilitate the necessary API requests for retrieving real-time data.

**Pseudocode:**

**def fetch\_covid\_stats(region):**

**url = f"https://disease.sh/v3/covid-19/countries/{region}"**

**response = send\_get\_request(url)**

**if response.status\_code == 200:**

**data = parse\_json(response.content)**

**cases = data['cases']**

**recovered = data['recovered']**

**deaths = data['deaths']**

**return cases, recovered, deaths**

**else:**

**print(f"Error fetching data: {response.status\_code}")**

**return None**

**Detailed explanation of the actual code:**

The `get\_user\_input()` function prompts the user to enter a region, which can be a country, state, or city. The `fetch\_covid\_data(region)` function then sends an HTTP GET request to the COVID-19 API to retrieve statistics for the specified region, returning the JSON response if successful, or `None` if there is an error. Next, the `display\_statistics(statistics, region)` function displays the COVID-19 statistics, including the total cases, recoveries, and deaths for the provided region. Finally, the `main()` function coordinates the overall process, managing user input and displaying the results seamlessly.

**Assumptions made (if any):**

It is assumed that the COVID-19 API will always provide the latest statistics and remain operational during the request. Additionally, users are expected to input regions in the correct format, such as country names or state abbreviations recognized by the API. Furthermore, the application assumes consistent internet connectivity to facilitate the necessary API requests for retrieving real-time data.

**Limitations:**

 **Error Handling**: The current implementation has basic error handling. More detailed error handling could be added to manage different HTTP status codes and potential issues (e.g., invalid region input).

 **Data Caching**: There is no caching mechanism to prevent repeated requests for the same region, which could lead to excessive API calls.

 **Rate Limiting**: The application does not handle rate limiting; if the API restricts the number of requests, the application may fail to retrieve data after a certain threshold.

 **Input Validation**: There is no validation for user input, which may lead to unexpected results or errors if the input does not match any known region.

**Code:**

**def mock\_fetch\_covid\_data(region):**

**# Simulating a response for demonstration purposes**

**mock\_data = {**

**"cases": 100000,**

**"recovered": 80000,**

**"deaths": 2000**

**}**

**return mock\_data**

**def get\_user\_input():**

**region = input("Enter a region (country/state/city): ")**

**return region**

**def fetch\_covid\_data(region):**

**# Replace this with the mock function**

**return mock\_fetch\_covid\_data(region)**

**def display\_statistics(statistics, region):**

**print(f"\nCOVID-19 Statistics for {region.capitalize()}:")**

**print(f"Cases: {statistics['cases']}")**

**print(f"Recoveries: {statistics['recovered']}")**

**print(f"Deaths: {statistics['deaths']}")**

**def main():**

**region = get\_user\_input()**

**statistics = fetch\_covid\_data(region)**

**if statistics:**

**display\_statistics(statistics, region)**

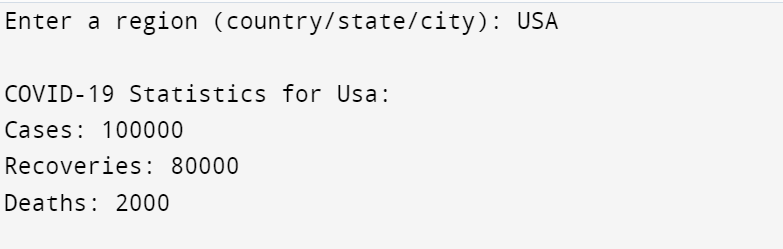
**else:**

**print("Error fetching data. Please check the region.")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Sample Output / Screen Shots**

****