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.

Answer:-

```
        import requests
def fetch_weather_data(location):
    api_key = 'your_api_key'
    base_url = 'https://api.openweathermap.org/data/2.5/weather'
    if location.isdigit():
        params = {'lat': location.split(',')[0], 'lon': location.split(',')[1], 'appid': api_key,
'units': 'metric'}
    else:
        params = {'q': location, 'appid': api_key, 'units': 'metric'}
    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()
        weather_data = response.json()
        return weather_data
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")
        return None
def display_weather(weather_data):
    if weather_data:
        print(f"Weather in {weather_data['name']}:")
        print(f"Temperature: {weather_data['main']['temp']} °C")
        print(f"Weather Condition: {weather_data['weather'][0]['description']}")
        print(f"Humidity: {weather_data['main']['humidity']} %")
        print(f"Wind Speed: {weather_data['wind']['speed']} m/s")
    else:
def main():
    location = input("Enter city name or coordinates (latitude,longitude): ")
    weather_data = fetch_weather_data(location)
    display_weather(weather_data)

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

Output:-

```
Enter the city name: Tamil Nadu
Temperature: 300.28K
Humidity: 68%
Weather Description: overcast clouds
Wind Speed: 10.54 m/s
```

Documentation of API Integration

API Used

Methods Used:

- fetch_weather_data(location): This function constructs a request to the OpenWeatherMap API based on the user input (city name or coordinates). It handles the API request using the `requests` library in Python, retrieves the JSON response, and returns it for further processing.
- display_weather(weather_data): This function takes the JSON response from the API (`weather_data`), extracts relevant weather information

(temperature, weather conditions, humidity, wind speed), and prints it to the console.

Assumptions Made and Potential Improvements

Assumptions

- The API key is assumed to be stored securely and is not shown in the example for security reasons.
- Input validation assumes simple checks (e.g., checking if the location input is numeric for coordinates).

Potential Improvements:

- Error Handling: Enhance error handling to cover various edge cases, such as invalid input format, network errors, and API rate limits.
- User Interface: Develop a graphical user interface (GUI) for better user interaction and visualization of weather data.
- Caching: Implement caching mechanisms to store previously fetched weather data and reduce API calls, considering API rate limits.
- Localization: Support multiple languages for weather descriptions based on user preferences or location.
- Unit Conversion: Allow users to choose between metric and imperial units for displaying weather data.
- Forecasting: Extend functionality to provide weather forecasts for future days in addition to current weather.

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.

Answer:-

```
import datetime

class Product:

    def __init__(self, product_id, name, price, initial_stock):

        self.product_id = product_id

        self.name = name

        self.price = price

        self.stock = initial_stock

        self.history = []
    def update_stock(self, quantity_change):
        self.stock += quantity_change
        self.history.append((datetime.datetime.now(), quantity_change,
self.stock))
class InventoryManagementSystem:
    def __init__(self):
        self.products = {}
    def add_product(self, product):
        self.products[product.product_id] = product
    def get_product(self, product_id):
        return self.products.get(product_id, None)
    def track_inventory(self, product_id, quantity_change):
        product = self.get_product(product_id)
        if product:
            product.update_stock(quantity_change)
        else:
            print(f"Product with ID {product_id} not found.")
    def get_current_stock(self, product_id):
        product = self.get_product(product_id)
        if product:
            return product.stock
        else:
            return None
```

```
def main():

    inventory_system = InventoryManagementSystem()

    product1 = Product("P001", "Laptop", 1000, 50)

    product2 = Product("P002", "Smartphone", 800, 100)

    inventory_system.add_product(product1)

    inventory_system.add_product(product2)

    inventory_system.track_inventory("P001", -5)

    inventory_system.track_inventory("P002", -10)

    print(f"Current stock of Laptop (P001):
{inventory_system.get_current_stock('P001')}")

    print(f"Current stock of Smartphone (P002):
{inventory_system.get_current_stock('P002')}")

if __name__ == "__main__":

    main()
```

Output:-

```
Current stock of Laptop (P001): 45

Current stock of Smartphone (P002): 90
```

Explanation:

Product Class: Represents a product with attributes such as ID, name, price, current stock, and history of stock changes.

InventoryManagementSystem Class: Manages products, allows adding products, updating stock levels, retrieving current stock, and tracking inventory changes.

Documentation

Reorder Optimization Algorithm:

To optimize reorder points and quantities, you can implement algorithms like the Economic Order Quantity (EOQ) or reorder point (ROP) models. These algorithms typically consider factors such as:

Demand Forecasting: Using historical sales data to predict future demand.

Lead Time: Estimating the time it takes from placing an order to receiving it.

Safety Stock: Maintaining buffer stock to mitigate stockouts due to variability in demand or lead time.

Assumptions:

Constant Lead Times: Assumes lead times for product deliveries are consistent.

Steady Demand Patterns: Assumes demand for products follows a predictable pattern based on historical data.

Potential Improvements:

Dynamic Adjustments: Implement algorithms that dynamically adjust reorder points based on changing demand patterns or lead times.

Supplier Integration: Integrate with suppliers' systems to get real-time updates on order status and availability.

Advanced Reporting: Enhance reporting capabilities to include financial metrics like cost of carrying inventory and potential savings from optimized ordering.

User Interface

For a user-friendly interface, consider implementing a GUI or a web-based application that allows:

Inputting product IDs or names to view current stock levels.

Generating reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.

Receiving alerts when stock levels fall below a certain threshold or when it's time to reorder.

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.

Answer:-

```
import requests

def fetch_traffic_data(start_point, destination):

    api_key = 'your_api_key'
    base_url = 'https://maps.googleapis.com/maps/api/directions/json'
    params = {
        'origin': start_point,
        'destination': destination,
        'key': api_key,
        'departure_time': 'now',
        'traffic_model': 'best_guess',
        'mode': 'driving'
    }
    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()
        traffic_data = response.json()
        return traffic_data
    except requests.exceptions.RequestException as e:
        print(f"Error fetching traffic data: {e}")
        return None
def display_traffic(traffic_data):
    if traffic_data:
        route = traffic_data['routes'][0]
        print(f"Total travel time: {route['legs'][0]['duration']['text']}")
        print("Traffic conditions:")
        for step in route['legs'][0]['steps']:
            print(f"- {step['html_instructions']}")

            if 'duration_in_traffic' in step:

                print(f"  Estimated duration in traffic:
{step['duration_in_traffic']['text']}")

            else:

                print("  No traffic data available for this step.")

            print()

    else:

        print("No traffic data available.")

def main():

    start_point = input("Enter starting point: ")

    destination = input("Enter destination: ")

    traffic_data = fetch_traffic_data(start_point, destination)

    display_traffic(traffic_data)

if __name__ == "__main__":

    main()
```

Explanation:

- fetch_traffic_data(start_point, destination): This function constructs a request to the Google Maps Directions API based on the user's input (starting point and destination). It fetches real-time traffic data using the `requests` library, including estimated travel time and traffic conditions.
- display_traffic(traffic_data): This function takes the JSON response from the API (`traffic_data`), extracts relevant information such as total travel time and traffic conditions for each step of the route, and prints it to the console.

Documentation of API Integration

API Used: Google Maps Directions API
(https://developers.google.com/maps/documentation/directions/overview)

Methods Used:

- fetch_traffic_data(start_point, destination): Constructs a request to the Google Maps Directions API to obtain real-time traffic information for a specified route. Parameters include origin, destination, API key, departure time (set to 'now' for real-time data), traffic model ('best_guess'), and mode of travel ('driving').
- display_traffic(traffic_data): Parses the JSON response from the API to extract and display relevant traffic information, including total travel time and traffic conditions for each route segment.

Assumptions Made and Potential Improvements

Assumptions:

- Real-Time Traffic Data: Assumes the availability and accuracy of real-time traffic data provided by the Google Maps Traffic API.
- Single Route Display: Assumes displaying information for the best route based on current traffic conditions.

Potential Improvements:

- Multiple Routes: Enhance the application to display multiple route options with comparisons based on travel time and traffic conditions.
- Alternative Modes of Transport: Extend functionality to support alternative modes of transport (e.g., walking, bicycling, public transit).
- Interactive Map: Develop a graphical interface with an interactive map displaying real-time traffic updates and alternative routes.
- Integration with Smart City Infrastructure: Integrate with local traffic cameras or sensors for more precise and localized traffic updates.
- User Preferences: Allow users to set preferences such as avoiding tolls, highways, or specifying preferred routes.

- **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.

Answer:-

```
import requests

def fetch_covid_statistics(region):

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

    try:

        response = requests.get(url)

        if response.status_code == 200:

            data = response.json()

            cases = data['cases']

            recoveries = data['recovered']

            deaths = data['deaths']
            print(f"COVID-19 Statistics for {region}:")

            print(f"Cases: {cases}")

            print(f"Recoveries: {recoveries}")

            print(f"Deaths: {deaths}")

        else:

            print(f"Failed to fetch data: {response.status_code}")

    except requests.exceptions.RequestException as e:

        print(f"Error fetching data: {str(e)}")

if __name__ == "__main__":

    region = input("Enter a country, state, or city: ")

    fetch_covid_statistics(region)
```

Implementation Explanation

- **Data Flow**:
  - The user inputs a region (country, state, or city).
  - The Python application sends a GET request to the COVID-19 statistics API (`disease.sh` in this case).
  - The API responds with JSON data containing COVID-19 statistics for the specified region.
  - The application parses the JSON response, extracts relevant statistics (cases, recoveries, deaths), and displays them to the user.
- **API Integration**:
  - The application uses the `requests` library to make HTTP GET requests to `https://disease.sh/v3/covid-19/countries/{region}` where `{region}` is replaced by the user's input.
  - Error handling is included to manage cases where the API call fails or encounters connection issues.

Assumptions Made and Potential Improvements

- **Assumptions**:
    - The user input is assumed to be correctly formatted to match a country, state, or city that the API recognizes.
    - The API (`disease.sh`) provides accurate and up-to-date COVID-19 statistics.
- **Potential Improvements**:
    - Error Handling: Enhance error handling to cover more edge cases such as network timeouts, API rate limiting, or unexpected API response formats.
    - User Interface: Develop a more user-friendly interface with options to view historical data, trends, or visual representations of data (graphs, charts).
    - Localization: Support multiple languages or regions for user input and display.
    - Caching: Implement caching mechanisms to reduce the number of API calls and improve application responsiveness.
    - Security: Ensure secure handling of API keys or tokens if required for accessing the API.