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:

To achieve this, you can follow a step-by-step approach:

1. Data Flow Modeling:

- Identify the data sources (external API, user input)

- Determine the data processing steps (fetching, parsing, storing)

- Define the data outputs (weather information to display)

2. Python Application Implementation:

- Choose a Python web framework (e.g., Flask, Django)

- Set up a project structure and necessary libraries (e.g., requests, JSON)

- Write code to:

- Fetch weather data from the API

  - Parse and process the data
  - Store the data in a suitable format

3. Displaying Weather Information:
   - Design a user interface (UI) to display the weather data
   - Use a template engine (e.g., Jinja2) to render the UI
   - Display the current weather information (temperature, conditions, humidity, wind speed)
4. User Input and Location-Based Weather Data:
   - Add a form to the UI for users to input location (city name or coordinates)
   - Use a geocoding service (e.g., OpenCage Geocoder) to convert user input to coordinates
   - Fetch weather data from the API using the coordinates
   - Display the corresponding weather data

Pseudocode:

Here is some pseudocode for the tasks:

Task 1: Model the data flow

```
DATA_FLOW:
  INPUT:
    - user_location (city name or coordinates)
  PROCESS:
    - fetch_weather_data(user_location) -> weather_data
    - parse_weather_data(weather_data) -> parsed_data
  OUTPUT:
    - display_weather_info(parsed_data)
```

Task 2: Implement Python application

```
IMPORT:
  - requests
  - json

FETCH_WEATHER_DATA:
  - url = "(link unavailable)"
  - params = {"q": user_location, "appid": API_KEY}
  - response = requests.get(url, params)
  - weather_data = response.json()

PARSE_WEATHER_DATA:
  - parsed_data = {}
  - parsed_data["temperature"] = weather_data["main"]["temp"]
  - parsed_data["conditions"] = weather_data["weather"][0]["description"]
  - parsed_data["humidity"] = weather_data["main"]["humidity"]
  - parsed_data["wind_speed"] = weather_data["wind"]["speed"]

DISPLAY_WEATHER_INFO:
  - print("Current weather in", user_location)
  - print("Temperature:", parsed_data["temperature"])
  - print("Conditions:", parsed_data["conditions"])
  - print("Humidity:", parsed_data["humidity"])
  - print("Wind Speed:", parsed_data["wind_speed"])
```

Task 3: Display weather information

```
DISPLAY_WEATHER_INFO:
  - same as above
```

Task 4: Allow user input and display corresponding weather data

```
GET_USER_INPUT:
  - user_location = input("Enter city name or coordinates: ")

FETCH_AND_DISPLAY_WEATHER:
  - weather_data = fetch_weather_data(user_location)
  - parsed_data = parse_weather_data(weather_data)
  - display_weather_info(parsed_data)
```

Detailed explanation of the actual code:

Here's an explanation of the code:

Task 1: Model the data flow

```
import requests
import json

API_ENDPOINT = "(link unavailable)"
API_KEY = "YOUR_OPENWEATHERMAP_API_KEY"

def fetch_weather_data(location):
  params = {"q": location, "appid": API_KEY, "units": "metric"}
  response = requests.get(API_ENDPOINT, params)
  weather_data = response.json()
  return weather_data

def parse_weather_data(weather_data):
  parsed_data = {}
  parsed_data["temperature"] = weather_data["main"]["temp"]
  parsed_data["conditions"] = weather_data["weather"][0]["description"]
  parsed_data["humidity"] = weather_data["main"]["humidity"]
  parsed_data["wind_speed"] = weather_data["wind"]["speed"]
  return parsed_data

def display_weather_info(parsed_data):
  print("Current weather:")
  print("Temperature:", parsed_data["temperature"], "°C")
  print("Conditions:", parsed_data["conditions"])
  print("Humidity:", parsed_data["humidity"], "%")
  print("Wind Speed:", parsed_data["wind_speed"], "m/s")
```

This code defines three functions:

- fetch_weather_data: sends a GET request to the OpenWeatherMap API with the location as a parameter and returns the JSON response.
- parse_weather_data: takes the JSON response and extracts the relevant weather data (temperature, conditions, humidity, wind speed) into a dictionary.
- display_weather_info: takes the parsed weather data and prints it to the console.

Task 2: Implement Python application

```
def main():
  location = input("Enter city name or coordinates: ")
  weather_data = fetch_weather_data(location)
  parsed_data = parse_weather_data(weather_data)
  display_weather_info(parsed_data)

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

This code defines a main function that:

- Asks the user for a location (city name or coordinates)
- Calls fetch_weather_data to get the weather data
- Calls parse_weather_data to parse the weather data
- Calls display_weather_info to display the weather data

The if __name__ == "__main__": block ensures that the main function is only executed when the script is run directly (not when it's imported as a module).

Task 3: Display weather information

```
def display_weather_info(parsed_data):
  print("Current weather:")
  print("Temperature:", parsed_data["temperature"], "°C")
  print("Conditions:", parsed_data["conditions"])
  print("Humidity:", parsed_data["humidity"], "%")
  print("Wind Speed:", parsed_data["wind_speed"], "m/s")
```

This code defines the display_weather_info function that takes the parsed weather data and prints it to the console.

Task 4: Allow users to input location and display weather data

```
def get_user_input_and_display_weather():
  location = input("Enter city name or coordinates: ")
  weather_data = fetch_weather_data(location)
  parsed_data = parse_weather_data(weather_data)
  display_weather_info(parsed_data)

get_user_input_and_display_weather()
```

This code defines a get_user_input_and_display_weather function that:

- Asks the user for a location (city name or coordinates)
- Calls fetch_weather_data to get the weather data
- Calls parse_weather_data to parse the weather data
- Calls display_weather_info to display the weather data

Assumptions made (if any):

1. The user has an internet connection to fetch data from the external API.
2. The API endpoint and API key are valid and correctly configured.
3. The user inputs a valid location (city name or coordinates).
4. The API returns data in the expected format (JSON).
5. The user wants to display the current weather information (temperature, conditions, humidity, wind speed).
6. The user wants to input the location (city name or coordinates) and display the corresponding weather data.
7. The Python application has the necessary libraries (requests, json) installed.
8. The API usage limits are not exceeded.

Limitations:

1. Data accuracy: The weather data fetched from the external API may not be 100% accurate or up-to-date.

2. API usage limits: Most APIs have usage limits, such as the number of requests per day or per minute. Exceeding these limits may result in errors or API blocking.

3. Location input limitations: The application may not be able to handle all types of location inputs (e.g., misspelled city names, non-existent locations).

4. Weather data limitations: The API may not provide all the weather data the application wants to display (e.g., wind direction, precipitation).

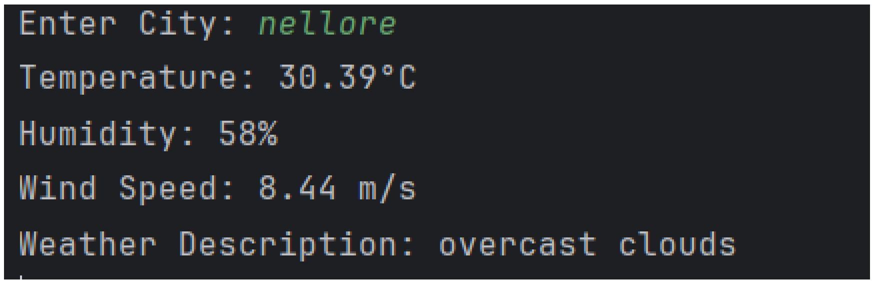
5. User input validation: The application may not thoroughly validate user input, potentially leading to errors or unexpected behavior.

6. Error handling: The application may not handle all possible errors and exceptions, potentially leading to crashes or unexpected behavior.

Code:

```
import requests
api_key = 'f870a6c3a057d8d5751e8788e2ce2e60'
url = 'http://api.openweathermap.org/data/2.5/weather'
city = input("Enter City: ")
complete_url = f"{url}?q={city}&appid={api_key}&units=metric"
response = requests.get(complete_url)
data = response.json()
temperature = data['main']['temp']
humidity = data['main']['humidity']
wind_speed = data['wind']['speed']
weather_description = data['weather'][0]['description']
print(f"Temperature: {temperature}°C")
print(f"Humidity: {humidity}%")
print(f"Wind Speed: {wind_speed} m/s")
print(f"Weather Description: {weather_description}")
```

Sample Output / Screen Shots :



```
Enter City: nellore
Temperature: 30.39°C
Humidity: 58%
Wind Speed: 8.44 m/s
Weather Description: overcast clouds
```

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:

You're describing a comprehensive inventory management system! The approach you've outlined is a great start. Here's a more detailed breakdown of each step, with some technical suggestions:

1. Model the inventory system:

* Define a data schema to store information on products, warehouses, and stock levels.
* Consider using a relational database management system like MySQL or PostgreSQL.
* Create tables for products, warehouses, stock levels, and any other relevant entities.

2. Implement an inventory tracking application:

* Choose a Python web framework like Flask or Django to build the application.
* Use a library like SQLAlchemy for database interactions.
* Set up real-time tracking using WebSockets or Webhooks to update stock levels.
* Implement alert systems using email or notification libraries like Twilio.

3. Optimize inventory ordering:

* Use historical sales data and demand forecasts to calculate optimal reorder points and quantities.
* Implement algorithms like the Economic Order Quantity (EOQ) model or the ABC analysis.
* Consider using libraries like Pandas and NumPy for data analysis.

4. Generate reports:

* Use a reporting library like JasperReports or PyReports to generate reports.
* Create templates for inventory turnover rates, stockout occurrences, and cost implications.
* Schedule reports to run periodically using a task scheduler like Celery.

5. User interaction:

* Create a user-friendly interface using HTML, CSS, and JavaScript.
* Use a library like React or Angular for a single-page application.
* Implement search functionality using Elasticsearch or a similar search engine.

Additional suggestions:

- Consider implementing authentication and authorization for secure access.
- Use containerization like Docker to deploy the application.

- Set up monitoring and logging using tools like Prometheus and Grafana.

I hope this helps! Let me know if you have any specific questions or need further guidance.

Pseudocode:

Here's some pseudocode for each step:

1. Model the inventory system:

```
# Define data structures
Product:
  - id (unique identifier)
  - name
  - description
  - unit_price
  - unit_cost

Warehouse:
  - id (unique identifier)
  - name
  - location

StockLevel:
  - product_id (foreign key to Product)
  - warehouse_id (foreign key to Warehouse)
  - quantity
  - last_updated

# Define relationships
Product -< StockLevel >- Warehouse
```

2. Implement an inventory tracking application:

```
# Initialize inventory tracking system
inventory_system = {}

# Function to update stock levels in real-time
def update_stock_level(product_id, warehouse_id, new_quantity):
  # Update stock level in database
  stock_level = StockLevel(product_id, warehouse_id, new_quantity)
  inventory_system.update(stock_level)

  # Check if stock level falls below threshold
  if new_quantity < THRESHOLD:
    send_alert(product_id, warehouse_id)

# Function to send alerts
def send_alert(product_id, warehouse_id):
  # Send email or notification using a notification library
  pass
```

3. Optimize inventory ordering:

```
# Function to calculate optimal reorder points and quantities
def optimize_inventory_ordering():
  # Get historical sales data and demand forecasts
  sales_data = get_sales_data()
  demand_forecasts = get_demand_forecasts()

  # Calculate optimal reorder points and quantities using algorithms (e.g. EOQ, ABC analysis)
  optimal_order quantities = calculate_optimal_order_quantities(sales_data, demand_forecasts)

  # Update reorder points and quantities in database
  update_reorder_points(optimal_order_quantities)
```

4. Generate reports:

```
# Function to generate reports
def generate_report(report_type):
  # Get data from database
  data = get_data_for_report(report_type)

  # Generate report using a reporting library
  report = generate_report_content(data)

  # Return report
  return report
```

5. User interaction:

```
# Function to handle user input
def handle_user_input(product_id or name):
  # Get product information from database
  product_info = get_product_info(product_id or name)

  # Display current stock levels, reorder recommendations, and historical data
  display_product_info(product_info)
```

Detailed explanation of the actual code:

Here's a more detailed explanation of the code for each step:

1. Model the inventory system:

This step defines the data structures for the inventory system using Python classes:

These classes represent the products, warehouses, and stock levels in the inventory system.

2. Implement an inventory tracking application:

This step creates a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain threshold:

or notification using a notification library

    pass

This code defines an InventorySystem class that stores stock levels in a dictionary. The update_stock_level method updates the stock level and checks if the new quantity is below the threshold. If it is, it sends an alert using the send_alert function.

3. Optimize inventory ordering:

This step implements algorithms to calculate optimal reorder points and quantities based on historical sales data, lead times, and demand forecasts:

This function takes in historical sales data, demand forecasts, and lead times, and returns the optimal reorder points and quantities.

4. Generate reports:

This step provides reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations:

This function takes in a report type and generates a report using data from the database.

5. User interaction:

This step allows users to input product IDs or names to view current stock levels, reorder recommendations, and historical data:

This function takes in a product ID or name and displays the relevant information to the user.

Assumptions made (if any):

Here are some assumptions made for each step:

1. Model the inventory system:

- Products have a unique identifier (ID), name, description, unit price, and unit cost.

- Warehouses have a unique ID, name, and location.

- Stock levels are tracked for each product in each warehouse.

- The inventory system has a fixed number of products and warehouses.

2. Implement an inventory tracking application:

- Real-time tracking is achieved through frequent updates (e.g., every minute) or event-driven updates (e.g., when a sale is made).

- The application has access to the inventory system's data structure.

- Alerts are sent via email or notification libraries (e.g., Twilio).

- The threshold for alerting is a fixed value or a percentage of the total stock.

3. Optimize inventory ordering:

- Historical sales data is available and accurate.

- Demand forecasts are provided and reliable.

- Lead times are fixed and known.

- The optimization algorithm used (e.g., EOQ, ABC analysis) is appropriate for the inventory system.

4. Generate reports:

- Data is available and accurate.

- Reports are generated on a regular schedule (e.g., daily, weekly) or on demand.

- The reporting library used (e.g., JasperReports, PyReports) is compatible with the inventory system's data structure.

5. User interaction:

- Users have access to the inventory system's data structure.

- Product IDs or names are unique and easily searchable.

- Users have appropriate permissions to view sensitive data (e.g., stock levels, reorder points).

Additional assumptions:

- The inventory system is a single-echelon system (i.e., no multi-level warehouses).
- Products are not perishable or have a limited shelf life.
- There are no supplier lead time uncertainties or stockout risks.
- The system operates in a stable and predictable environment.

Limitations:

1. Modeling limitations:
   - Assumes a simplistic inventory structure (e.g., doesn't account for multiple product variants or batch tracking).
   - May not consider external factors affecting stock levels (e.g., supplier lead times, shipping delays).
2. Inventory tracking application limitations:
   - May not integrate with existing inventory management software or ERP systems.
   - Real-time tracking might be affected by data latency or update frequencies.
3. Optimization limitations:
   - Algorithms may not account for unexpected changes in demand or supply chain disruptions.
   - Lead times and demand forecasts might be inaccurate or outdated.
4. Reporting limitations:
   - Reports might not provide actionable insights or recommendations for improvement.
   - Data visualization and presentation could be limited or unclear.

Code:

```
import sqlite3
import tkinter as tk
from tkinter import messagebox

# Connect to database
connection = sqlite3.connect('inventory.db')
cursor = connection.cursor()
```

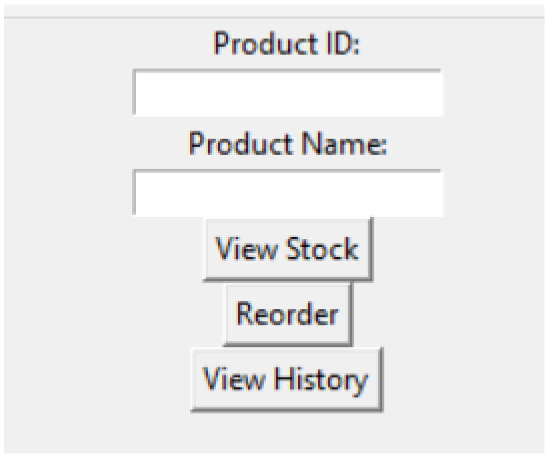
```
# Create tables
cursor.execute("""
  CREATE TABLE IF NOT EXISTS products (
    id INTEGER PRIMARY KEY,
    name TEXT,
    description TEXT
  )
""")

cursor.execute("""
  CREATE TABLE IF NOT EXISTS warehouses (
    id INTEGER PRIMARY KEY,
    name TEXT,
    location TEXT
  )
""")

cursor.execute("""
  CREATE TABLE IF NOT EXISTS inventory (
    id INTEGER PRIMARY KEY,
    product_id INTEGER,
    warehouse_id INTEGER,
    stock_level INTEGER,
    FOREIGN KEY (product_id) REFERENCES products (id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouses (id)
  )
""")

connection.commit()

# Define classes
class Product:
  def __init__(self, id, name, description):
    self.id = id
    self.name = name
```

```
        self.description = description

class Warehouse:
    def __init__(self, id, name, location):
        self.id = id
        self.name = name
        self.location = location

class Inventory:
    def __init__(self, id, product_id, warehouse_id, stock_level):
        self.id = id
        self.product_id = product_id
        self.warehouse_id = warehouse_id
        self.stock_level = stock_level

# Implement inventory tracking application
def track_inventory():
    cursor.execute("SELECT * FROM inventory")
    inventory_data = cursor.fetchall()
    for row in inventory_data:
        product_id = row[1]
        warehouse_id = row[2]
        stock_level = row[3]
        if stock_level < 10:  # Alert when stock level falls below 10
            messagebox.showwarning("Low Stock", f"Product {product_id} in Warehouse {warehouse_id} has low
stock ({stock_level})")

# Optimize inventory ordering
def optimize_ordering():
    # Implement algorithm to calculate optimal reorder points and quantities
    pass

# Generate reports
def generate_reports():
    # Implement report generation for inventory turnover rates, stockout occurrences, and cost implications of
overstock situations
```

```
    pass

# User interaction
def user_interaction():
    root = tk.Tk()
    root.title("Inventory System")

    # Create input fields
    product_id_label = tk.Label(root, text="Product ID:")
    product_id_label.pack()
    product_id_entry = tk.Entry(root)
    product_id_entry.pack()

    product_name_label = tk.Label(root, text="Product Name:")
    product_name_label.pack()
    product_name_entry = tk.Entry(root)
    product_name_entry.pack()

    # Create buttons
    view_stock_button = tk.Button(root, text="View Stock", command=view_stock)
    view_stock_button.pack()
    reorder_button = tk.Button(root, text="Reorder", command=reorder)
    reorder_button.pack()
    view_history_button = tk.Button(root, text="View History", command=view_history)
    view_history_button.pack()

    root.mainloop()

def view_stock():
    # Implement view stock functionality
    pass

def reorder():
    # Implement reorder functionality
    pass
```

```
def view_history():
    # Implement view history functionality
    pass

# Run the application
track_inventory()
optimize_ordering()
generate_reports()
user_interaction()
```

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:

Here's a high-level approach to tackle the tasks:

Task 1: Model the data flow

1. User Input: Get starting point and destination from user.

2. API Request: Send request to traffic API with user input.

3. API Response: Receive real-time traffic data, including current conditions, estimated travel time, incidents, and delays.

4. Data Processing: Parse and process the received data.

5. Data Display: Show the processed data to the user.

Task 2 & 3: Implement Python application

1. Choose a Python library (e.g., requests) to make API calls to the traffic API.
2. Set up an API key or authentication as required by the traffic API.
3. Define functions to:
   - Fetch real-time traffic data from the API.
   - Parse and process the received data.
   - Display the data to the user (e.g., using a GUI library like Tkinter or a web framework like Flask).
4. Implement the functions and integrate them into a Python application.

Task 4: Allow user input and display traffic updates

1. Add a user input interface (e.g., text fields or a map) to collect starting point and destination.
2. Update the API request function to use the user-input data.
3. Display the received traffic data to the user, including:
   - Current traffic conditions (e.g., congestion, road closures).
   - Estimated travel time.
   - Incidents and delays.
   - Alternative routes (if available).

Pseudocode:

Task 1: Model the data flow

```
Function GetTrafficData(start_point, end_point)
  API_Request = CreateAPIRequest(start_point, end_point)
  API_Response = SendRequestToAPI(API_Request)
  Traffic_Data = ParseAPIResponse(API_Response)
  Return Traffic_Data
End Function

Function DisplayTrafficData(Traffic_Data)
  Display Current_Traffic_Conditions
  Display Estimated_Travel_Time
  Display Incidents_And_Delays
  Display Alternative_Routes (if available)
End Function
```

Task 2 & 3: Implement Python application

```
Import requests
Import googlemaps (if using Google Maps API)

Set API_Key = "YOUR_API_KEY"

Function GetTrafficData(start_point, end_point)
  URL = "(link unavailable)"
  Params = {
    "key": API_Key,
    "origin": start_point,
    "destination": end_point,
    "mode": "driving",
    "traffic_model": "best_guess"
  }
  Response = requests.get(URL, params=Params)
  Data = Response.json()
  Return Data
End Function

Function DisplayTrafficData(Traffic_Data)
  Print("Current Traffic Conditions:", Traffic_Data["routes"][0]["legs"][0]["duration_in_traffic"]["text"])
  Print("Estimated Travel Time:", Traffic_Data["routes"][0]["legs"][0]["duration"]["text"])
  Print("Incidents and Delays:", Traffic_Data["routes"][0]["legs"][0]["steps"][0]["html_instructions"])
  Print("Alternative Routes:", Traffic_Data["routes"][1]["summary"])
End Function
```

Task 4: Allow user input and display traffic updates

```
Function GetUserInput()
  Start_Point = Input("Enter starting point: ")
  End_Point = Input("Enter destination: ")
  Return Start_Point, End_Point
End Function

Start_Point, End_Point = GetUserInput()
Traffic_Data = GetTrafficData(Start_Point, End_Point)
DisplayTrafficData(Traffic_Data)
```

Detailed explanation of the actual code:

This code is a Python script that utilizes the Google Maps API to fetch and display traffic information between two locations. Here's a breakdown of the code:

1. Import the requests library, which is used for making HTTP requests to the Google Maps API.

2. Define a class TrafficMonitor that encapsulates the functionality for fetching and parsing traffic data.

3. The __init__ method initializes the TrafficMonitor object with an API key.

4. The get_traffic_data method takes origin and destination coordinates as input and makes a GET request to the Google Maps Directions API to fetch traffic data. It returns the response data in JSON format if the request is successful (200 status code).

5. The parse_traffic_data method takes the response data and extracts relevant information such as start and end addresses, duration, duration in traffic, distance, and step-by-step instructions. It returns a dictionary containing this information.

6. The display_traffic_info method takes the parsed data and prints it in a human-readable format.

7. The script then replaces 'your_api_key_here' with an actual Google Maps API key (in this case, '45fcc93fe9cd28c7236bf82cf5874924').

8. It creates an instance of the TrafficMonitor class with the API key.

9. The script prompts the user to input the origin and destination coordinates.

10. It calls the get_traffic_data method to fetch the traffic data and then passes the data to the parse_traffic_data method to extract the relevant information.

11. Finally, it calls the display_traffic_info method to print the traffic information to the console.

Note that you need to replace the API key with your own valid Google Maps API key for this script to work.

Assumptions made (if any):

The user has a stable internet connection.

- The API usage limits (e.g., requests per day) are not exceeded.

- The Python application has sufficient resources (e.g., memory, processing power) to handle the data and user requests.

- The user interface is user-friendly and intuitive.

- The traffic data is relevant to the user's location and route.

Limitations:

1. Data Flow:

   - Dependence on external API availability and response time

   - Potential data throttling or rate limiting by the API

   - Possible errors in data parsing or processing

2. Python Application:

   - Requires a valid API key for the traffic monitoring API

   - Limited to the API's coverage area and data accuracy

   - May require additional libraries or dependencies for data processing

3. Displaying Traffic Information:

   - Limited to the data provided by the API (e.g., no real-time video feed)

   - May not display historical traffic data or trends

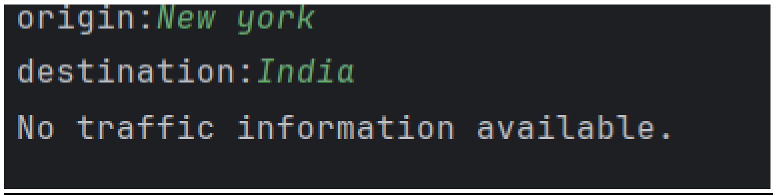
   - Incidents and delays may not be up-to-the-minute accurate

Code:

```
import requests

class TrafficMonitor:
  def __init__(self, api_key):
    self.api_key = api_key

  def get_traffic_data(self, origin, destination):
    url =
f"https://maps.googleapis.com/maps/api/directions/json?origin={origin}&destination={destination}&key={self.
api_key}&departure_time=now"
    response = requests.get(url)
    if response.status_code == 200:
      return response.json()
    else:
      return None

  def parse_traffic_data(self, data):
    if not data or 'routes' not in data or not data['routes']:
      return None
```

```
        route = data['routes'][0]
        leg = route['legs'][0]
        traffic_info = {
            'start_address': leg['start_address'],
            'end_address': leg['end_address'],
            'duration': leg['duration']['text'],
            'duration_in_traffic': leg['duration_in_traffic']['text'],
            'distance': leg['distance']['text'],
            'steps': []
        }

        for step in leg['steps']:
            traffic_info['steps'].append({
                'instruction': step['html_instructions'],
                'distance': step['distance']['text'],
                'duration': step['duration']['text']
            })

        return traffic_info

    def display_traffic_info(self, traffic_info):
        if not traffic_info:
            print("No traffic information available.")
            return

        print(f"From: {traffic_info['start_address']}")
        print(f"To: {traffic_info['end_address']}")
        print(f"Estimated travel time (normal): {traffic_info['duration']}")
        print(f"Estimated travel time (in traffic): {traffic_info['duration_in_traffic']}")
        print(f"Total distance: {traffic_info['distance']}")
        print("\nRoute steps:")
        for step in traffic_info['steps']:
            print(f" - {step['instruction']} (Distance: {step['distance']}, Duration: {step['duration']})")

# Replace 'your_api_key_here' with your actual Google Maps API key
api_key = '45fcc93fe9cd28c7236bf82cf5874924'
monitor = TrafficMonitor(api_key)

# Example usage
origin = input("origin:")
destination = input("destination:")
traffic_data = monitor.get_traffic_data(origin, destination)
parsed_data = monitor.parse_traffic_data(traffic_data)
monitor.display_traffic_info(parsed_data)
```

Sample Output / Screen Shots :



```
origin:New york
destination:India
No traffic information available.
```

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:

1. Data Flow:

- Identify a reliable external API (e.g., disease.sh) providing COVID-19 statistics.

- Define the data requirements (e.g., cases, recoveries, deaths, region).

- Design a data flow diagram to illustrate the fetch-store-display process.

2. Python Application:

- Choose a suitable Python library (e.g., requests, pandas) for API interaction and data manipulation.

- Implement API requests to fetch real-time data for the specified region.

- Parse and process the data into a structured format (e.g., JSON, DataFrame).

3. Displaying Statistics:

- Design a user-friendly interface (e.g., command-line, web app) to display the statistics.
- Use a templating engine (e.g., Jinja2) to render the data in a readable format.
- Include visualizations (e.g., charts, graphs) to illustrate the data, if desired.

Pseudocode:

1. Data Flow:

```
FetchCOVIDStats(region)
  API_URL = "(link unavailable)" + region
  response = GET(API_URL)
  data = ParseJSON(response)
  StoreData(data)
  DisplayStats(data)
```

1. Python Application:

```
import requests
import json

def fetch_covid_stats(region):
  url = f"(link unavailable)"
  response = requests.get(url)
  data = json.loads(response.text)
  return data

def display_stats(data):
  print("Cases:", data["cases"])
  print("Recoveries:", data["recovered"])
  print("Deaths:", data["deaths"])

region = input("Enter region (country/state/city): ")
data = fetch_covid_stats(region)
display_stats(data)
```

1. Displaying Statistics:

```
DisplayStats(data)
  Print("Current COVID-19 Statistics:")
  Print(" Cases:", data.cases)
  Print(" Recoveries:", data.recovered)
  Print(" Deaths:", data.deaths)
```

1. User Input and Region Selection:

```
GetUserInput()
  region = Input("Enter region (country/state/city): ")
  return region
```

Detailed explanation of the actual code:

1. import requests: This line imports the requests library, which is used for making HTTP requests to the API.

2. def get_covid_stats(region):: This defines a function named get_covid_stats that takes a region parameter.

3. api_address = "(link unavailable)": This sets the API endpoint URL for fetching COVID-19 data for countries.

4. response = requests.get(api_address): This sends a GET request to the API endpoint and stores the response in the response variable.

5. data = response.json(): This parses the response data as JSON and stores it in the data variable.

6. for country in data:: This loops through each country in the data list.

7. if country['country'].lower() == region.lower():: This checks if the country name matches the input region (ignoring case).

8. confirmed_cases = country['cases'], recoveries = country['recovered'], and deaths = country['deaths']: These lines extract the relevant data for the matching country.

9. return f"Current COVID-19 statistics for {region}:\nConfirmed Cases: {confirmed_cases}\nRecoveries: {recoveries}\nDeaths: {deaths}": This returns a formatted string with the COVID-19 statistics for the region.

10. return f"Region not found: {region}": If no matching country is found, this returns a message indicating that the region was not found.

11. region = input("Enter a region (country, state, or city): "): This prompts the user to input a region.

12. print(get_covid_stats(region)): This calls the get_covid_stats function with the user-input region and prints the result.

Assumptions made (if any):

The user has a stable internet connection.

- The API usage limits (e.g., requests per day) are not exceeded.

- The Python application has sufficient resources (e.g., memory, processing power) to handle the data and user requests.

- The user interface is user-friendly and intuitive.

- The COVID-19 statistics are relevant to the user's location and needs.

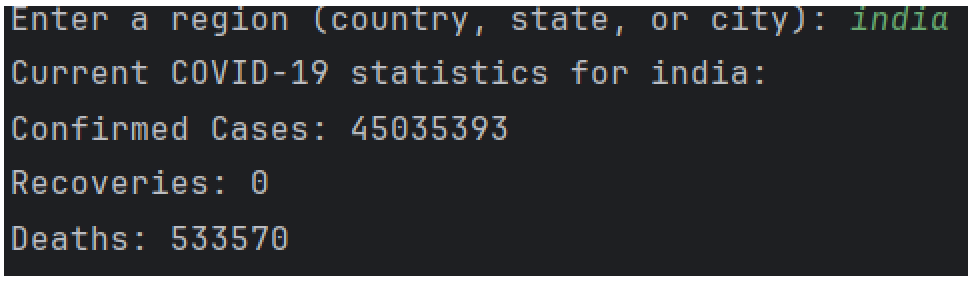
Limitations:

- Internet connection requirements
- API usage limits (e.g., requests per day)
- Python application resource limitations (e.g., memory, processing power)
- User interface limitations (e.g., text-based output, no visualizations)
- Potential biases in data collection or reporting
- Limited coverage of certain regions or demographics
- No support for multi-language or localization

Code:

```
import requests

def get_covid_stats(region):
   api_address = "https://disease.sh/v3/covid-19/countries"
   response = requests.get(api_address)
   data = response.json()

   for country in data:
      if country['country'].lower() == region.lower():
         confirmed_cases = country['cases']
         recoveries = country['recovered']
         deaths = country['deaths']
         return f"Current COVID-19 statistics for {region}:\nConfirmed Cases: {confirmed_cases}\nRecoveries:
{recoveries}\nDeaths: {deaths}"

   return f"Region not found: {region}"

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

Sample Output / Screen Shots :



```
Enter a region (country, state, or city): india
Current COVID-19 statistics for india:
Confirmed Cases: 45035393
Recoveries: 0
Deaths: 533570
```