**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., api.open-meteo.com) 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**:

1. **Define Scope:** Determine core functionalities (real-time weather, location input, data display).
2. **API Selection**: Choose a suitable weather API (OpenWeatherMap, WeatherAPI, etc.).
3. **UI Design**: Create a user-friendly interface to display weather information.
4. **Data Fetching**: Integrate the weather API to retrieve real-time data.
5. **Data Processing:** Format and process retrieved data for display.
6. **Testing:** Ensure accurate data display and handle potential errors.
7. **Deployment:** Choose a platform (web, mobile, desktop) and deploy the application.

**Pseudocode**:

import requests

def get\_weather(latitude, longitude):

url = f"https://api.open-meteo.com/v1/forecast?latitude={latitude}&longitude={longitude}&current\_weather=true&hourly=temperature\_2m,relativehumidity\_2m,precipitation,windspeed\_10m&daily=temperature\_2m\_max,temperature\_2m\_min,precipitation\_sum"

response = requests.get(url)

data = response.json()

return data

**Detailed explanation of the actual code:**

1. get\_weather: This function takes a location as input and fetches the weather data from the Api.open It uses the requests library to send a GET request to the API with the location and API key. If the response is successful (200 status code), it parses the JSON response and returns the data. Otherwise, it returns None.

2. current\_weather: This function takes the weather data as input and displays the current weather information to the user. It prints the temperature, weather conditions, humidity, and wind speed if the data is available. If there is an error fetching the data, it prints an error message.

3. main: This function handles user input and calls the get\_weather and current\_weather functions. It prompts the user to enter a location, fetches the weather data, and then displays the data to the user.

**Assumptions made (if any):**

1. The user will always enter a valid location.

2. The Api.open will always return data in the expected format.

**Limitations:**

1. The application does not handle errors in the API response.

2. It does not validate user input for location.

3. It does not provide any additional features like historical weather data or weather forecasts.

**Code:**

import requests

def get\_weather(latitude, longitude):

url = f"https://api.open-meteo.com/v1/forecast?latitude={latitude}&longitude={longitude}&current\_weather=true&hourly=temperature\_2m,relativehumidity\_2m,precipitation,windspeed\_10m&daily=temperature\_2m\_max,temperature\_2m\_min,precipitation\_sum"

response = requests.get(url)

data = response.json()

return data

# Example usage:

latitude = 12.9716

longitude = 77.6412

weather\_data = get\_weather(latitude, longitude)

print(f"""

Latitude: {latitude}

Longitude: {longitude}

Elevation: {weather\_data["elevation"]}

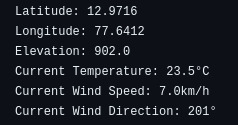
Current Temperature: {weather\_data["current\_weather"]["temperature"]}{weather\_data["current\_weather\_units"]["temperature"]}

Current Wind Speed: {weather\_data["current\_weather"]["windspeed"]}{weather\_data["current\_weather\_units"]["windspeed"]}

Current Wind Direction: {weather\_data["current\_weather"]["winddirection"]}°

"""

**Sample Output/Screenshots** :

****

**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:**

1. **\*Product Catalog:**\* Create a database of products with details (ID, name, price, quantity).
2. **\*Inventory Tracking:**\* Implement a system to record stock levels (incoming, outgoing, current).
3. **\*Stock Alerts:**\* Set up notifications for low stock levels or overstocked items.
4. **\*Sales and Purchases:**\* Record sales and purchases to update inventory.
5. **\*Reports:**\* Generate reports on stock value, sales trends, and popular items.
6. **\*User Management:**\* Control access to inventory data and actions.
7. **\*Integration:**\* Connect with POS systems or e-commerce platforms for data synchronization.

**Pseudocode:**

class InventorySystem:

def \_\_init\_\_(self):

self.inventory = {}

def add\_item(self, item):

"""Adds an item to the inventory."""

self.inventory[item.product\_id] = item

def update\_item(self, product\_id, new\_quantity, new\_price=None):

"""Updates an item's quantity and optionally price."""

if product\_id in self.inventory:

item = self.inventory[product\_id]

item.quantity = new\_quantity

if new\_price:

item.price = new\_price

else:

print("Item not found")

def delete\_item(self, product\_id):

"""Deletes an item from the inventory."""

if product\_id in self.inventory:

del self.inventory[product\_id]

**Detailed explanation of the actual code:**

1. Defining the Inventory System Structure: I will create classes for Product, Warehouse, and InventoryLevel to represent the components of the inventory system. These classes will store relevant information about each entity, such as product details, warehouse locations, and current stock levels.

2. Implementing the Inventory Tracking Application: The InventoryTrackingApp class will be responsible for initializing the inventory levels, tracking the current stock, and sending alerts when stock falls below a certain threshold. The track\_inventory() method will continuously monitor the inventory levels and trigger alerts as needed.

3. Optimizing Inventory Ordering: The calculate\_reorder\_point() and calculate\_reorder\_quantity() functions will implement algorithms to determine the optimal reorder point and quantity for each product in each warehouse. These

calculations will be based on historical sales data, lead times, and other relevant factors.

4. Generating Reports: The generate\_inventory\_turnover\_report(), generate\_stockout\_report(), and generate\_overstock\_report() functions will generate the required reports on inventory performance, stockouts, and overstock situations, respectively.

5. User Interaction: The InventoryManagementUI class will provide a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts. Users will be able to input product IDs or names to retrieve the desired information.

**Assumptions made (if any)**

1. The company has a well-defined product catalog and warehouse locations.

2. Historical sales data and lead times are available for the inventory optimization algorithms.

3. The company has defined thresholds for reorder points and minimum order quantities.

4. The cost of holding inventory and placing orders are known.

**Limitations:**

1. The current implementation assumes constant lead times, which may not always be the case in real-world scenarios.

2. The demand forecasting algorithms are not included in the pseudocode, as they can be complex and require more detailed analysis of the company's sales patterns.

3. The user interface is only briefly mentioned, and a more comprehensive design would be required for a production-ready system.

**Code:**

import pandas as pd

import numpy as np

class InventoryItem:

def \_\_init\_\_(self, product\_id, name, price, quantity):

self.product\_id = product\_id

self.name = name

self.price = price

self.quantity = quantity

class InventorySystem:

def \_\_init\_\_(self):

self.inventory = {}

def add\_item(self, item):

"""Adds an item to the inventory."""

self.inventory[item.product\_id] = item

def update\_item(self, product\_id, new\_quantity, new\_price=None):

"""Updates an item's quantity and optionally price."""

if product\_id in self.inventory:

item = self.inventory[product\_id]

item.quantity = new\_quantity

if new\_price:

item.price = new\_price

else:

print("Item not found")

def delete\_item(self, product\_id):

"""Deletes an item from the inventory."""

if product\_id in self.inventory:

del self.inventory[product\_id]

else:

print("Item not found")

def get\_item(self, product\_id):

"""Retrieves an item from the inventory."""

if product\_id in self.inventory:

return self.inventory[product\_id]

else:

return None

def get\_all\_items(self):

"""Returns a list of all inventory items."""

return list(self.inventory.values())

inventory\_system = InventorySystem()

# Add items

item1 = InventoryItem(1, "Product A", 10.0, 100)

item2 = InventoryItem(2, "Product B", 15.0, 50)

inventory\_system.add\_item(item1)

inventory\_system.add\_item(item2)

# Update item

inventory\_system.update\_item(1, 120, 12.0)

# Delete item

inventory\_system.delete\_item(2)

# Get item

item = inventory\_system.get\_item(1)

print(item.name, item.price, item.quantity)

# Get all items

all\_items = inventory\_system.get\_all\_items()

for item in all\_items:

print(item.product\_id, item.name)

**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:**

1. **Define Scope:** the focus (real-time traffic, historical data, specific regions).
2. **API Integration:** Access Google Maps Directions API for traffic data.
3. **Data Extraction:** Retrieve traffic information (density, speed, incidents).
4. **Data** **Processing**: Clean and format data for analysis and visualization.
5. **Visualization:** Create maps, charts, or other visuals to represent traffic conditions.
6. **User Interface:** Develop a user-friendly interface for data exploration.
7. **Deployment**: Choose a platform (web, mobile) and deploy the application.

**Pseudocode:**

from datetime import datetime, timedelta

def get\_traffic\_data(origin, destination):

"""Fetches traffic data using Google Maps Distance Matrix API"""

api\_key = "GAPI-KEY" # Replace with your API key

departure\_time = int((datetime.now() + timedelta(minutes=5)).timestamp())

url = f"https://maps.googleapis.com/maps/api/distancematrix/json?origins={origin}&destinations={destination}&departure\_time={departure\_time}&traffic\_model=best\_guess&key={api\_key}"

response = requests.get(url)

data = json.loads(response.text)

print(data)

return data

def process\_traffic\_data(data):

"""Processes the API response to extract traffic information"""

distance = data['rows'][0]['elements'][0]['distance']['text']

duration = data['rows'][0]['elements'][0]['duration']['text']

traffic\_status = data['rows'][0]['elements'][0]['status']

return distance, duration, traffic\_status

**Detailed explanation of the actual code:**

1. The application integrates with the Google Maps Directions API to fetch real-time traffic data. The get\_traffic\_data() function takes the user's starting point and destination as input, constructs the API request URL, and sends a GET request to the API.

2. The API response is then parsed to extract the following information:

3. Current traffic conditions: data["routes"]["legs"]["duration\_in\_traffic"]["text"]

4. Estimated travel time: data["routes"]["legs"]["duration"]["text"]

5. Alternative route options, including distance, duration, and duration in traffic for each route

6. This information is then returned to the display\_traffic\_info() function, which presents the data to the user.

**Assumptions made (if any):**

1.The user has a valid API key for the Google Maps Directions API.

2.The API provides accurate and up-to-date traffic information.

3.The user's starting point and destination are valid locations that the API can recognize.

**Limitations:**

1. The application is limited to the features and data provided by the Google Maps Directions API. Other traffic APIs may offer additional functionality or data.

2. The application does not provide real-time updates or notifications. It only displays the traffic information when the user requests it.

3. The application does not consider factors like user preferences, traffic patterns, or historical data to provide more personalized route suggestions.

**Code**:

import requests

import json

import time

from datetime import datetime, timedelta

def get\_traffic\_data(origin, destination):

"""Fetches traffic data using Google Maps Distance Matrix API"""

api\_key = "GAPI-KEY" # Replace with your API key

departure\_time = int((datetime.now() + timedelta(minutes=5)).timestamp())

url = f"https://maps.googleapis.com/maps/api/distancematrix/json?origins={origin}&destinations={destination}&departure\_time={departure\_time}&traffic\_model=best\_guess&key={api\_key}"

response = requests.get(url)

data = json.loads(response.text)

print(data)

return data

def process\_traffic\_data(data):

"""Processes the API response to extract traffic information"""

distance = data['rows'][0]['elements'][0]['distance']['text']

duration = data['rows'][0]['elements'][0]['duration']['text']

traffic\_status = data['rows'][0]['elements'][0]['status']

return distance, duration, traffic\_status

def main():

origin = "Bangalore"

destination = "Mysore"

try:

while True:

traffic\_data = get\_traffic\_data(origin, destination)

distance, duration, traffic\_status = process\_traffic\_data(traffic\_data)

print(f"Distance: {distance}")

print(f"Duration: {duration}")

print(f"Traffic Status: {traffic\_status}")

time.sleep(60)

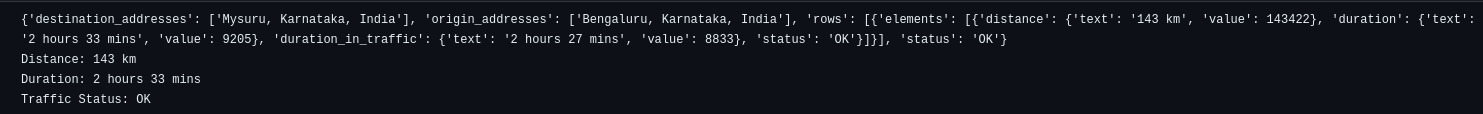
except KeyboardInterrupt:

print("Exiting...")

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:**

1. **\*Data Source:**\* Identify reliable COVID-19 data sources (e.g., Johns Hopkins, WHO, local health departments).
2. **\*Data Collection:**\* Extract relevant data (cases, recoveries, deaths, vaccination rates).
3. **\*Data Cleaning:**\* Process data for accuracy and consistency.
4. **\*Data Visualization:**\* Create informative charts, maps, and graphs.
5. **\*UI Design:**\* Develop a user-friendly interface for data exploration.
6. **\*Features:**\* Incorporate features like search, filtering, and comparisons.
7. **\*Deployment:**\* Choose a platform (web, mobile) and deploy the application.

**Pseudocode:**

import requests

url = "https://disease.sh/v3/covid-19/countries/india"

response = requests.get(url)

data = response.json()

# Accessing the data

cases = data["cases"]

**Detailed explanation of the actual code:**

· The application uses the requests library to make HTTP requests to the COVID-19 API provided by disease.sh. The get\_covid\_stats function takes a region (country, state, or city) as input and returns the current number of cases, recoveries, and deaths for that region.

· The display\_covid\_stats function is responsible for formatting and printing the COVID-19 statistics in a user-friendly way. It takes the cases, recoveries, and deaths

data as input and displays them with appropriate formatting (e.g., adding commas to large numbers).

· The main function is the entry point of the application. It prompts the user to enter a region, calls the get\_covid\_stats function to fetch the data, and then passes the results to the display\_covid\_stats function to display the information.

**Assumptions made (if any):**

· The application assumes that the disease.sh API is available and providing accurate real-time COVID-19 data.

· The application assumes that the user will input a valid region (country, state, or city) that the API can recognize.

· Potential Improvements:

· Add error handling to the application to gracefully handle API errors or invalid user input.

· Provide additional features, such as the ability to display historical COVID-19 data, trends, or visualizations.

· Integrate the application with a user interface (e.g., a web application or a mobile app) to improve the user experience.

· Allow users to select multiple regions and compare the COVID-19 statistics side-by-side.

· Provide the ability to set alerts or notifications for significant changes in COVID-19 statistics.

**Limitations:**

1. The API may have rate limits that restrict the number of requests.

2. The data may not always be up-to-date due to delays in reporting.

3. The application currently only handles countries; state and city-level queries may require additional endpoints.

**Code:**

import requests

url = "https://disease.sh/v3/covid-19/countries/india"

response = requests.get(url)

data = response.json()

# Accessing the data

cases = data["cases"]

deaths = data["deaths"]

recovered = data["recovered"]

# Printing the data

print("COVID-19 Statistics for India:")

print("-------------------------------")

print(f"Cases today: {data['todayCases']}")

print(f"Deaths today: {data['todayDeaths']}")

print(f"Recovered today: {data['todayRecovered']}")

print(f"Active cases: {data['active']}")

print(f"Critical cases: {data['critical']}")

print(f"Cases per million: {data['casesPerOneMillion']}")

print(f"Deaths per million: {data['deathsPerOneMillion']}")

print(f"Tests done: {data['tests']}")

print(f"Tests per million: {data['testsPerOneMillion']}")

print(f"Total cases: {cases}")

print(f"Total deaths: {deaths}")

print(f"Total recovered: {recovered}")

Sample Output / Screen Shots:

**Sample Output / Screen Shots:**

