**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 data flow diagram illustrates the interaction between the application and the OpenWeatherMap API. The application receives user input for the location, fetches the weather data from the API, and displays the current weather information to the user.

+-------------------+ +-------------------+

| User Interface | | Weather API |

| (Input Location) | | (OpenWeatherMap) |

+-------------------+ +-------------------+

| |

| Fetch Weather Data |

+----------------------------------->|

| |

| |

| Display Weather Data |

|<-----------------------------------+

| |

+-------------------+ +-------------------+

| Display Weather | | Weather Data |

| Information | | (JSON Response) |

+-------------------+ +-------------------+

**Pseudocode:**

BEGIN

FUNCTION fetch weather(location):

CALL Weather API with location

IF API returns valid response:

RETURN weather data

ELSE:

RETURN error message

FUNCTION display\_weather(weather\_data):

PRINT temperature

PRINT weather conditions

PRINT humidity

PRINT wind speed

FUNCTION main():

PROMPT user for location

weather\_data = fetch\_weather(location)

IF weather\_data is not empty:

display\_weather(weather\_data)

ELSE:

PRINT error message

END

**Detailed explanation of the actual code:**

1. fetch\_weather\_data: This function takes a location as input and fetches the weather data from the OpenWeatherMap API. 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. display\_weather\_data: 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 fetch\_weather\_data and display\_weather\_data 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 OpenWeatherMap API 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 http.client

import json

def get\_weather(api\_key, location):

conn = http.client.HTTPConnection("api.openweathermap.org")

conn.request("GET", f"/data/2.5/weather?q={location}&appid={api\_key}&units=metric")

response = conn.getresponse()

if response.status == 200:

data = json.loads(response.read().decode("utf-8"))

return {

"temperature": data["main"]["temp"],

"weather\_condition": data["weather"][0]["description"],

"humidity": data["main"]["humidity"],

"wind\_speed": data["wind"]["speed"]

}

else:

return {"error": "Failed to retrieve weather data"}

def display\_weather(weather\_data):

print("Current Weather Data:")

print(f"Temperature: {weather\_data['temperature']}°C")

print(f"Weather Condition: {weather\_data['weather\_condition']}")

print(f"Humidity: {weather\_data['humidity']}%")

print(f"Wind Speed: {weather\_data['wind\_speed']} m/s")

def main():

api\_key = input("Enter your OpenWeatherMap API key: ")

location = input("Enter the location (city name): ")

weather\_data = get\_weather(api\_key, location)

if "error" in weather\_data:

print(weather\_data["error"])

else:

display\_weather(weather\_data)

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

main()

**Sample Output in IDLE**



**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. Model the Inventory System**

Structure of the Inventory System

Entities:

* **Products**
  + **Product ID**: Unique identifier
  + **Name**: Product name
  + **Category**: Category of the product
  + **Current Stock Level**: Current quantity in stock
  + **Reorder Level**: Minimum stock level before reorder
  + **Reorder Quantity**: Amount to order when stock is low
  + **Historical Sales Data**: Past sales data for demand forecasting
  + **Lead Time**: Time taken to receive new stock
* **Warehouses**
  + **Warehouse ID**: Unique identifier
  + **Location**: Warehouse location
  + **Stock Levels**: Current stock levels of products at this warehouse

**Relationships:**

* Each product can be stored in multiple warehouses.
* Each warehouse can contain multiple products.

**Pseudocode:**

BEGIN

DEFINE Product:

ID

Name

Category

Current Stock Level

Reorder Level

Reorder Quantity

Historical Sales Data

Lead Time

DEFINE Warehouse:

ID

Location

Stock Levels (list of Products)

FUNCTION fetch\_inventory():

RETURN list of all products and their stock levels

FUNCTION check\_reorder(product):

IF product.Current Stock Level < product.Reorder Level:

RETURN True

ELSE:

RETURN False

FUNCTION calculate\_reorder\_quantity(product):

RETURN product.Reorder Quantity

FUNCTION generate\_report():

FOR each product:

CALCULATE turnover rate

CALCULATE stockout occurrences

CALCULATE cost implications of overstock

PRINT report data

FUNCTION main():

PROMPT user for action

IF user selects 'View Inventory':

DISPLAY fetch\_inventory()

IF user selects 'Check Reorder':

FOR each product:

IF check\_reorder(product):

DISPLAY product.ID and calculate\_reorder\_quantity(product)

IF user selects 'Generate Report':

CALL generate\_report()

END

**explanation of the actual code:**

**Product and Warehouse Structure:** Defines Product attributes (ID, name, stock level) and Warehouse attributes (ID, location, stock levels).

**Inventory Functions:** Key functions fetch inventory data, check reorder needs, calculate reorder quantities, and generate reports on inventory metrics.

**User Interaction:** The main() function prompts users to perform actions like viewing inventory or generating reports, facilitating easy interaction.

**Reorder Logic:** The system checks if stock levels fall below reorder points to trigger restocking, ensuring inventory availability.

**Reporting**: Generates reports that provide insights into turnover rates, stockouts, and overstock costs for informed decision-making

**Assumptions made (:if any)**

Stable Demand Patterns: The model assumes demand for products remains consistent over time, ensuring reliable forecasting.

Constant Lead Times: It presumes that lead times for receiving stock are predictable and do not fluctuate significantly.

Accurate Sales Data: Historical sales data is assumed to be accurate and sufficient for effective demand forecasting.

**Limitations:**

**Static Demand Assumptions**: The system assumes stable demand patterns, which may not reflect real-world fluctuations, leading to inaccurate forecasts.

**Simplistic Lead Time:** It relies on constant lead times, which can vary due to supplier issues or logistical challenges, affecting reorder accuracy.

**Basic Sales Data Utilization**: The system uses historical sales data in a straightforward manner, potentially overlooking seasonal trends or market changes.

**Limited User Interface**: The console-based interface restricts usability and accessibility, making it less user-friendly compared to graphical applications.

**Lack of Integration**: The system operates in isolation and does not integrate with other business systems (e.g., sales or accounting), limiting data accuracy and real-time insights

**Code:**

class Product:

def \_init\_(self, product\_id, name, current\_stock, reorder\_point, reorder\_quantity):

self.product\_id = product\_id

self.name = name

self.current\_stock = current\_stock

self.reorder\_point = reorder\_point

self.reorder\_quantity = reorder\_quantity

class Warehouse:

def \_init\_(self, warehouse\_id, location):

self.warehouse\_id = warehouse\_id

self.location = location

self.products = []

def track\_inventory(products):

for product in products:

if product.current\_stock < product.reorder\_point:

print(f"Alert: {product.name} is below the reorder point. Current stock: {product.current\_stock}")

recommend\_reorder(product)

def recommend\_reorder(product):

new\_stock = product.current\_stock + product.reorder\_quantity

print(f"Recommended reorder for {product.name}: {product.reorder\_quantity} units. New stock level: {new\_stock}")

def calculate\_reorder\_point(historical\_sales, lead\_time, desired\_service\_level):

# Implement algorithms to calculate the optimal reorder point

# based on historical sales data, lead time, and desired service level

pass

def calculate\_reorder\_quantity(historical\_sales, lead\_time, holding\_cost, ordering\_cost):

# Implement algorithms to calculate the optimal reorder quantity

# based on historical sales data, lead time, holding cost, and ordering cost

pass

def generate\_inventory\_report(products):

# Generate reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations

pass

def user\_interface():

# Define sample products and warehouses

product1 = Product(1, "Product A", 50, 20, 30)

product2 = Product(2, "Product B", 15, 10, 25)

warehouse1 = Warehouse(1, "Warehouse A")

warehouse1.products = [product1, product2]

while True:

user\_input = input("Enter a product ID or name (or 'exit' to quit): ")

if user\_input.lower() == "exit":

break

# Look up the product and display current stock, reorder recommendations, and historical data

for product in warehouse1.products:

if str(product.product\_id) == user\_input or product.name.lower() == user\_input.lower():

print(f"Product: {product.name}")

print(f"Current Stock: {product.current\_stock}")

recommend\_reorder(product)

# Display historical data

break

else:

print("Product not found.")

# Test the application

user\_interface()

**Sample Output in IDLE**



**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. Data Flow Diagram

Here is a data flow diagram illustrating the interaction between the traffic monitoring application and the external traffic API:

text

+-------------------+ +-------------------------+

| User Interface | | Traffic API |

| (Input Start/End) | | (Google Maps Traffic API) |

+-------------------+ +-------------------------+

| |

| Fetch Traffic Data |

+----------------------------------->|

| |

| |

| Display Traffic Data |

|<-----------------------------------+

| |

+-------------------+ +-------------------------+

| Display Traffic | | Traffic Data |

| Information | | (JSON Response) |

+-------------------+ +-------------------------+

**The key steps in the data flow are:**

* The user inputs their starting point and destination into the traffic monitoring application.
* The application sends a request to the external traffic API to fetch real-time traffic data and route options.
* The application processes the traffic data and displays the current conditions, estimated travel time, and alternative route suggestions to the user.
* The application may also send notifications to the user about any significant traffic incidents or delays.

**Pseudocode:**

BEGIN

DEFINE Route:

Start Point

End Point

Current Traffic Conditions

Estimated Travel Time

Incidents/Delays

FUNCTION fetch\_traffic\_data(start, end):

CALL Traffic API with start and end points

IF API returns valid response:

RETURN traffic data

ELSE:

RETURN error message

FUNCTION display\_traffic\_data(traffic\_data):

PRINT current traffic conditions

PRINT estimated travel time

PRINT incidents or delays

FUNCTION main():

PROMPT user for starting point and destination

traffic\_data = fetch\_traffic\_data(start, end)

IF traffic\_data is not empty:

display\_traffic\_data(traffic\_data)

ELSE:

PRINT error message

END

**Detailed explanation of the actual code:**

**fetch\_traffic\_data(start, end):** This function retrieves real-time traffic data from the Google Maps Traffic API by making an HTTP GET request with the user-provided start and end locations. It returns the JSON response containing traffic information.

**display\_traffic\_data(traffic\_data):** This function processes the API response and displays current traffic conditions, estimated travel time, and any reported incidents or delays. It extracts relevant details from the response and formats them for user readability.

**main():** The main function serves as the entry point of the application, prompting the user for a starting point and destination, calling the traffic data fetching function, and displaying the results.

**Assumptions made (if any):**

Valid User Input**:** The user inputs valid starting and destination points recognized by the API.

API Availability**:** The Google Maps API is available and responds accurately to requests.

API Key Validity: The provided API key is valid and has not exceeded usage limits.

**Limitations:**

API Dependency: The system relies on external APIs, making it vulnerable to downtime or changes in service terms, which can affect functionality.

Input Validity: It assumes users provide valid and recognized starting and destination points, which may lead to errors if the input is incorrect.

Real-Time Data Constraints: The accuracy of traffic updates is contingent on real-time data availability, which may not always reflect immediate conditions.

Limited Historical Context: The system does not analyze historical traffic patterns extensively, potentially overlooking trends that could inform better routing.

Single Route Suggestion: It primarily suggests one route based on current conditions, lacking the ability to offer multiple alternative routes for flexibility.

**Code:**

import socket

import time

import json

def get\_user\_input():

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

destination = input("Enter destination: ")

return start\_point, destination

def send\_api\_request(start\_point, destination, api\_key):

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

sock.connect(("maps.googleapis.com", 443))

request = f"GET /maps/api/directions/json?origin={start\_point}&destination={destination}&key={api\_key}&traffic\_model=best\_guess HTTP/1.1\r\nHost: maps.googleapis.com\r\n\r\n"

sock.sendall(request.encode())

response = b""

while True:

data = sock.recv(1024)

if not data:

break

response += data

sock.close()

# Split the response into headers and body

response\_parts = response.decode().split("\r\n\r\n", 1)

# Check the response status code

status\_code = int(response\_parts[0].split("\r\n")[0].split(" ")[1])

if status\_code == 200:

return json.loads(response\_parts[1])

else:

raise Exception(f"API request failed with status code {status\_code}")

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

if "routes" in traffic\_data:

for route in traffic\_data["routes"]:

for leg in route["legs"]:

print(f"Route: {leg['start\_address']} to {leg['end\_address']}")

print(f"Estimated Travel Time: {leg['duration']['text']}")

for step in leg["steps"]:

print(f"Step: {step['html\_instructions']}")

if "traffic\_speed\_entry" in step:

print(f"Traffic Speed: {step['traffic\_speed\_entry']['speed']} km/h")

if step['traffic\_speed\_entry']['congestion'] == True:

print("Congestion Detected")

print()

else:

print("Error: Unable to fetch traffic data.")

def main():

api\_key = "YOUR\_API\_KEY"

while True:

start\_point, destination = get\_user\_input()

try:

traffic\_data = send\_api\_request(start\_point, destination, api\_key)

display\_traffic\_data(traffic\_data)

except Exception as e:

print(f"Error: {e}")

print(f"Last updated: {time.strftime('%Y-%m-%d %H:%M:%S')}")

print()

input("Press Enter to continue...")

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

main()

**Sample Output in IDLE**



**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:**

**+-------------------+ +--------------------------+**

**| User Interface | | COVID-19 API |**

**| (Input Region) | | (disease.sh) |**

**+-------------------+ +--------------------------+**

**| |**

**| Fetch COVID-19 Data |**

**+----------------------------------->|**

**| |**

**| |**

**| Display COVID-19 Data |**

**|<-----------------------------------+**

**| |**

**+-------------------+ +--------------------------+**

**| Display Statistics | | COVID-19 Data |**

**| Information | | (JSON Response) |**

**+-------------------+ +--------------------------+**

**The key steps are:**

1. The user inputs a region (country, state, or city) into the application.
2. The application sends a request to the COVID-19 API to fetch the real-time statistics for the specified region.
3. The COVID-19 API processes the request and returns the current case, recovery, and death data.
4. The application receives the COVID-19 statistics and displays the information to the user.

**Pseudocode:**

BEGIN

DEFINE COVIDStats:

Region

Total Cases

Total Recoveries

Total Deaths

FUNCTION fetch\_covid\_data(region):

CALL COVID-19 API with specified region

IF API returns valid response:

RETURN COVID data

ELSE:

RETURN error message

FUNCTION display\_covid\_data(covid\_data):

PRINT total cases

PRINT total recoveries

PRINT total deaths

FUNCTION main():

PROMPT user for region (country/state/city)

covid\_data = fetch\_covid\_data(region)

IF covid\_data is not empty:

display\_covid\_data(covid\_data)

ELSE:

PRINT error message

END

**Detailed explanation of the actual code:**

fetch\_covid\_data(region): This function retrieves real-time COVID-19 statistics from the disease.sh API by making an HTTP GET request for the specified region. It returns the JSON response containing COVID-19 data.

display\_covid\_data(covid\_data): This function processes the API response and displays the total number of cases, recoveries, and deaths for the specified region, formatting the information for user readability.

main(): The main function serves as the entry point of the application, prompting the user for a region, calling the COVID data fetching function, and displaying the results.

**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.
* Valid User Input: The user inputs a valid region recognized by the API, ensuring accurate data retrieval.
* API Availability: The disease.sh API is available and responds accurately to requests without downtime.
* Internet Connectivity: The application assumes the user has a stable internet connection to fetch real-time data.

**Limitations:**

API Dependency: The system relies on external APIs, making it vulnerable to outages or changes in API structure, which can disrupt data retrieval.

Input Validity: It assumes users input valid region names recognized by the API, which may lead to errors if the input is misspelled or unclear.

Real-Time Data Constraints: The data fetched may not always be truly real-time, leading to potential delays in reporting the latest statistics.

Limited Regional Coverage: The application might not provide comprehensive coverage for all regions, especially for smaller countries or cities.

Basic User Interface: The console-based interface is simplistic, lacking advanced features for data visualization or user engagement compared to web or mobile applications.

**Code:**

import urllib.request

import json

# Function to get COVID-19 statistics from disease.sh API

def get\_covid\_data(location):

# Set the API endpoint and parameters

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

# Send a GET request to the API

with urllib.request.urlopen(url) as response:

# Load the JSON response

data = json.load(response)

# Extract the relevant COVID-19 data

cases = data["cases"]

recoveries = data["recovered"]

deaths = data["deaths"]

# Return the COVID-19 data as a dictionary

return {"cases": cases, "recoveries": recoveries, "deaths": deaths}

# Function to display the COVID-19 statistics

def display\_covid\_data(covid\_data):

# Print the COVID-19 data in a readable format

print("Current COVID-19 Statistics:")

print(f"Cases: {covid\_data['cases']}")

print(f"Recoveries: {covid\_data['recoveries']}")

print(f"Deaths: {covid\_data['deaths']}")

# Main function to run the program

def main():

# Get the location from the user

location = input("Enter the country, state, or city: ")

# Get the COVID-19 data

try:

covid\_data = get\_covid\_data(location)

except urllib.error.HTTPError:

print("Failed to retrieve COVID-19 data. Please check the location and try again.")

return

# Display the COVID-19 data

display\_covid\_data(covid\_data)

# Run the main function

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

main()

**Sample Output / Screen Shots**

