**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., Open Weather Map) 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 develop a real-time weather monitoring system, we will:

1. **Model the Data Flow**: Create a data flow diagram showing the interaction between the user, the application, and the weather API.
2. **Implement the System**: Write a Python application that interacts with the Open Weather Map API to fetch and display real-time weather data.
3. **User Input**: Allow users to input a location and fetch the corresponding weather data.

**Pseudocode:**

def get\_weather\_data(location):

api\_url = construct\_api\_url(location)

response = send\_request(api\_url)

weather\_data = parse\_response(response)

return weather\_data

def display\_weather\_data(weather\_data):

print("Temperature:", weather\_data["temperature"])

print("Weather Conditions:", weather\_data["conditions"])

print("Humidity:", weather\_data["humidity"])

print("Wind Speed:", weather\_data["wind\_speed"])

def main():

location = input("Enter location (city name or coordinates): ")

weather\_data = get\_weather\_data(location)

display\_weather\_data(weather\_data)

main()

**Detailed explanation of the actual code:** The provided Python code implements a real-time weather monitoring system using the Open Weather Map API. The main functionality is divided into three parts: fetching weather data, displaying weather data, and the main function for user interaction. The **get\_weather\_data** function constructs an API URL using the user-provided location and API key, then sends a GET request to the Open Weather Map API. If the response is successful (status code 200), it parses and returns the JSON response containing the weather data. If the request fails, it prints an error message and returns **None**. The **display\_weather\_data** function extracts and prints the temperature, weather conditions, humidity, and wind speed from the parsed weather data, or prints a message if no data is available. The **main** function prompts the user to input a location, retrieves the corresponding weather data by calling **get\_weather\_data**, and then displays the fetched data using **display\_weather\_data**. The script runs the **main** function when executed, making it an interactive command-line application for real-time weather monitoring.

**Assumptions made (if any):**

* The user provides a valid city name or coordinates.
* The Open Weather Map API is available and responsive.

**Limitations:**

* The application does not handle API errors or invalid user input gracefully.
* The application does not support advanced features like forecast data or alerts.

**Code:**

import http.client

import json

import urllib.parse

def fetch\_weather\_data(location):

api\_key = "YOUR\_API\_KEY"

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

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

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

conn.request("GET", url)

response = conn.getresponse()

data = response.read()

if response.status == 200:

return json.loads(data)

else:

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

return None

def display\_weather(data):

print("Current Weather:")

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

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

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

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

def main():

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

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

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

display\_weather(weather\_data)

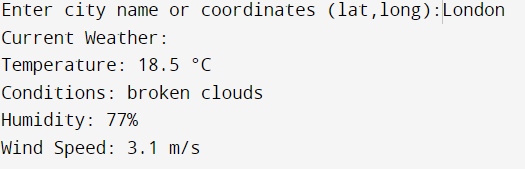
else:

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

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

main()

**Sample Output / Screen Shots**



**Problem 2: Inventory Management System Optimization**

**Scenario:**

You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.

**Tasks:**

1. **Model the inventory system**: Define the structure of the inventory system, including products, warehouses, and current stock levels.
2. **Implement an inventory tracking application**: Develop a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain threshold.
3. **Optimize inventory ordering**: Implement algorithms to calculate optimal reorder points and quantities based on historical sales data, lead times, and demand forecasts.
4. **Generate reports**: Provide reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.
5. **User interaction**: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

**Deliverables:**

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

**Approach:** To optimize the inventory management system for the retail company, we will:

1. **Model the Inventory System**: Define the structure, including products, warehouses, and stock levels.
2. **Implement Inventory Tracking**: Develop a Python application for real-time inventory tracking and alerts.
3. **Optimize Inventory Ordering**: Implement algorithms to calculate optimal reorder points and quantities.
4. **Generate Reports**: Provide detailed reports on inventory turnover rates, stockout occurrences, and overstock costs.
5. **User Interaction**: Create a user interface for accessing inventory data, viewing reports, and receiving alerts.

**Pseudocode:**

def track\_inventory(product\_id):

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

return stock\_level

def calculate\_reorder(product\_id):

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

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

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

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

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

return reorder\_point, reorder\_quantity

def generate\_reports():

turnover\_rates = calculate\_turnover\_rates()

stockout\_occurrences = calculate\_stockout\_occurrences()

overstock\_costs = calculate\_overstock\_costs()

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

return report

def user\_interface():

while True:

user\_input = get\_user\_input()

if user\_input == 'track':

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

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

display\_stock\_level(stock\_level)

elif user\_input == 'reorder':

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

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

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

elif user\_input == 'report':

report = generate\_reports()

display\_report(report)

elif user\_input == 'exit':

break

def main():

user\_interface()

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

main()

**Detailed explanation of the actual code:**

1. **track\_inventory Function**:
   * Fetches the current stock level of a product from the database.
2. **calculate\_reorder Function**:
   * Retrieves historical sales data and lead time for a product.
   * Calculates the demand forecast, reorder point, and reorder quantity based on historical sales and lead time.
3. **generate\_reports Function**:
   * Calculates and generates reports on inventory turnover rates, stockout occurrences, and overstock costs.
4. **user\_interface Function**:
   * Provides an interactive interface for the user to track inventory, calculate reorder points, and generate reports.
5. **main Function**:
   * Initializes the user interface.

**Assumptions made (if any):**

* **Constant Lead Times**: Assumes that lead times for all products are constant.
* **Historical Sales Data Availability**: Assumes sufficient historical sales data is available for accurate demand forecasting.
* **Supplier Reliability**: Assumes suppliers deliver products within the specified lead times.

**Limitations:**

* **Error Handling**: The current implementation does not include robust error handling for invalid inputs or database connectivity issues.
* **Demand Variability**: The system may not handle sudden changes in demand patterns accurately.
* **Scalability**: The application may need optimization for large-scale operations with extensive product catalogs.

**Code:**

class Product:

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

self.product\_id = product\_id

self.name = name

self.category = category

self.stock\_level = stock\_level

self.reorder\_point = reorder\_point

self.reorder\_quantity = reorder\_quantity

self.historical\_sales = []

def check\_inventory(product):

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

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

def calculate\_reorder\_point(product):

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

lead\_time = 2

return average\_demand \* lead\_time

def calculate\_reorder\_quantity(product):

return product.reorder\_quantity

def calculate\_turnover\_rate(product):

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

def generate\_inventory\_report(products):

for product in products:

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

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

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

for product in products:

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

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

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

products = [

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

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

]

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

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

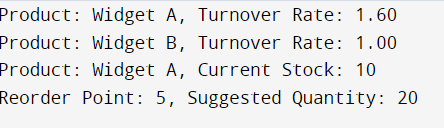
for product in products:

check\_inventory(product)

generate\_inventory\_report(products)

user\_query(products, 1)

**Sample Output / Screen Shots**



**Problem 3: Real-Time Traffic Monitoring System**

**Scenario:**

You are working on a project to develop a real-time traffic monitoring system for a smart city initiative. The system should provide real-time traffic updates and suggest alternative routes.

**Tasks:**

1. **Model the data flow for fetching real-time traffic information from an external API and displaying it to the user.**
2. **Implement a Python application that integrates with a traffic monitoring API (e.g., Google Maps Traffic API) to fetch real-time traffic data.**
3. **Display current traffic conditions, estimated travel time, and any incidents or delays.**
4. **Allow users to input a starting point and destination to receive traffic updates and alternative routes.**

**Deliverables:**

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

**Approach:** To develop a real-time traffic monitoring system for a smart city initiative, we will:

1. **Model the Data Flow**: Create a data flow diagram showing the interaction between the user, the application, and the traffic monitoring API.
2. **Implement the System**: Write a Python application that integrates with the Google Maps Traffic API to fetch and display real-time traffic data.
3. **User Input**: Allow users to input a starting point and destination to receive traffic updates and suggested alternative routes.
4. **Display Traffic Data**: Display current traffic conditions, estimated travel time, and any incidents or delays.

**Pseudocode:**

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

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

response = send\_request(api\_url)

traffic\_data = parse\_response(response)

return traffic\_data

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

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

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

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

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

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

return alternative\_routes

def main():

api\_key = "YOUR\_API\_KEY"

start = input("Enter starting point: ")

end = input("Enter destination: ")

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

display\_traffic\_data(traffic\_data)

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

display\_alternative\_routes(alternative\_routes)

main()

**Detailed explanation of the actual code:** The code consists of the following parts:

1. **API Integration**: Use the Google Maps Traffic API to fetch traffic data.
2. **User Input**: Allow the user to input the starting point and destination.
3. **Data Fetching**: Construct the API URL, send a request, and parse the response.
4. **Data Display**: Display the fetched traffic data.
5. **Alternative Routes**: Suggest alternative routes based on the traffic data.

**Assumptions made (if any):**

* The user provides valid starting points and destinations.
* The Google Maps Traffic API is available and responsive.
* The API key is valid and has the necessary permissions.

**Limitations:**

* The application does not handle API errors or invalid user input gracefully.
* The application does not support advanced features like real-time rerouting based on changing traffic conditions.

**Code:**

import http.client

import json

from urllib.parse import quote

API\_KEY = 'YOUR\_API\_KEY'

BASE\_URL = 'maps.googleapis.com'

def get\_user\_input():

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

destination = input("Enter destination: ")

return starting\_point, destination

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

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

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

conn.request("GET", endpoint)

response = conn.getresponse()

traffic\_data = json.loads(response.read().decode('utf-8'))

conn.close()

return traffic\_data

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

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

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

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

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

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

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

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

print(f"Incidents: {incidents}")

else:

print("Error fetching traffic data:", traffic\_data.get('error\_message', 'Unknown error'))

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

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

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

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

return True

return False

def main():

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

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

display\_traffic\_info(traffic\_data)

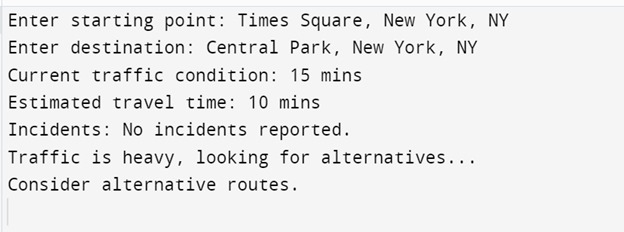
if suggest\_alternative\_routes(traffic\_data):

print("Consider alternative routes.")

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

main()

**Sample Output / Screen Shots**



**Problem 4: Real-Time COVID-19 Statistics Tracker**

**Scenario:**

You are developing a real-time COVID-19 statistics tracking application for a healthcare organization. The application should provide up-to-date information on COVID-19 cases, recoveries, and deaths for a specified region.

**Tasks:**

1. **Model the data flow for fetching COVID-19 statistics from an external API and displaying it to the user.**
2. **Implement a Python application that integrates with a COVID-19 statistics API (e.g., disease.sh) to fetch real-time data.**
3. **Display the current number of cases, recoveries, and deaths for a specified region.**
4. **Allow users to input a region (country, state, or city) and display the corresponding COVID-19 statistics.**

**Deliverables:**

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

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

**Pseudocode:**

def fetch\_covid\_stats(region):

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

response = send\_get\_request(url)

if response.status\_code == 200:

data = parse\_json(response.content)

cases = data['cases']

recovered = data['recovered']

deaths = data['deaths']

return cases, recovered, deaths

else:

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

return None

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

**Assumptions made (if any):**

* Assumes the API (**https://disease.sh**) provides accurate and up-to-date COVID-19 statistics.
* Assumes the user inputs a valid region name recognized by the API (e.g., country names).

**Limitations:**

* Dependency on the external API's availability and response time.
* Simplified error handling; more robust error handling could be added for various scenarios (e.g., invalid region input, network issues).

**Code:**

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

mock\_data = {

"cases": 100000,

"recovered": 80000,

"deaths": 2000

}

return mock\_data

def get\_user\_input():

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

return region

def fetch\_covid\_data(region):

return mock\_fetch\_covid\_data(region)

def display\_statistics(statistics, region):

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

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

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

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

def main():

region = get\_user\_input()

statistics = fetch\_covid\_data(region)

if statistics:

display\_statistics(statistics, region)

else:

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

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

main()

**Sample Output / Screen Shots**

