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 develop a real-time weather monitoring system, start by gathering requirements from stakeholders to understand their needs, such as current weather data, forecasts, and severe weather alerts. The system architecture should be designed with several key components. First, integrate with weather data providers using APIs (e.g., Open Weather Map or NOAA) to fetch real-time data. This data is then processed and aggregated by a backend service that handles data storage in a database for historical reference and user preferences. The system should include a real-time processing engine to ensure timely updates. For the user interface, design a frontend that presents weather information through an intuitive dashboard with visualizations like charts and maps. Additionally, implement a notification system to provide alerts for severe weather conditions through various channels such as email, SMS, or push notifications. Finally, consider offering a public API to allow external access to the weather data and forecasts. This approach ensures a comprehensive, user-friendly, and responsive weather monitoring system.

Pseudocode:

```
// Define main function
function main():
  // Prompt user for location
```

```
    location = getUserInput("Enter location (city or coordinates): ")

    // Fetch weather data
    weatherData = fetchWeatherData(location)

    // Check if data fetching was successful
    if weatherData is not null:
        // Display weather data
        displayWeatherData(weatherData)
    else:
        print("Failed to retrieve weather data for the specified location.")

// Function to get user input
function getUserInput(prompt):
    print(prompt)
    return userInput()

// Function to fetch weather data from an API
function fetchWeatherData(location):
    // Define API endpoint and key
    apiUrl = "https://api.weatherprovider.com/current"
    apiKey = "YOUR_API_KEY"

    // Make API request
    response = makeHttpRequest(apiUrl, {"location": location, "key": apiKey})

    // Check if request was successful
    if response.statusCode == 200:
        // Parse and return weather data
        return parseJson(response.body)
    else:
        return null
```

```
// Function to display weather data
function displayWeatherData(weatherData):
    // Extract relevant information
    temperature = weatherData.temperature
    humidity = weatherData.humidity
    windSpeed = weatherData.windSpeed
    condition = weatherData.condition

    // Print weather information
    print("Current Weather:")
    print("Temperature: " + temperature + "°C")
    print("Humidity: " + humidity + "%")
    print("Wind Speed: " + windSpeed + " km/h")
    print("Condition: " + condition)

// Function to make an HTTP request
function makeHttpRequest(url, parameters):
    // Implementation for making HTTP GET request with parameters
    // Returns a response object with statusCode and body

// Function to parse JSON data
function parseJson(jsonString):
    // Implementation for parsing JSON string into a data structure
    return parsedData

// Run the main function
main()
```

Detailed explanation of the actual code:

1. **Import Libraries:**
   - \`import requests\`: To handle HTTP requests.
   - \`import json\`: To parse JSON responses.

2. **Define `get_user_input(prompt)` Function:**

  - Displays a prompt to the user.

  - Retrieves and returns user input as a string.

3. **Define `fetch_weather_data(location)` Function:**

  - **API URL:** Set the endpoint for the weather API (e.g., `"https://api.weatherprovider.com/current"`).

  - **API Key:** Define your unique API key for authentication.

  - **Prepare Parameters:** Create a dictionary with location and API key.

  - **Make Request:** Use `requests.get(api_url, params=params)` to send a GET request.

  - **Check Response:** Verify if the request was successful using `response.status_code`.

  - **Parse JSON:** If successful, return the parsed JSON data using `response.json()`. Otherwise, return `None`.

4. **Define `display_weather_data(weather_data)` Function:**

  - **Extract Data:**

    - Temperature: `weather_data.get("main", {}).get("temp", "N/A")`

    - Humidity: `weather_data.get("main", {}).get("humidity", "N/A")`

    - Wind Speed: `weather_data.get("wind", {}).get("speed", "N/A")`

    - Condition: `weather_data.get("weather", [{}])[0].get("description", "N/A")`

  - **Display Data:** Print the weather information in a user-friendly format.

5. **Define `main()` Function:**

  - **Prompt User:** Use `get_user_input("Enter location (city or coordinates): ")` to get the location.

  - **Fetch Data:** Call `fetch_weather_data(location)` to retrieve the weather data.

  - **Display Data:** If data is successfully retrieved, call `display_weather_data(weather_data)`. Otherwise, print an error message.

6. **Run Main Function:**

  - **Conditional Execution:** Use `if __name__ == "__main__":` to ensure `main()` runs when the script is executed directly.

Assumptions made (if any): When developing a real-time weather monitoring system, several assumptions are typically made to ensure the system functions as intended.

- The system assumes that a reliable weather API (e.g., OpenWeatherMap, Weatherbit, or NOAA) is available and accessible. This API provides the necessary weather data and supports requests for current conditions.
- It is assumed that a valid API key is obtained and configured correctly. The key is necessary for authenticating requests to the weather API.
- The system assumes that the weather API returns data in a standard JSON format. The structure of this JSON data (e.g., fields for temperature, humidity, wind speed) is assumed to be consistent with the API documentation.
- It is assumed that the system has reliable network connectivity to make HTTP requests to the weather API and retrieve data.

Limitations:

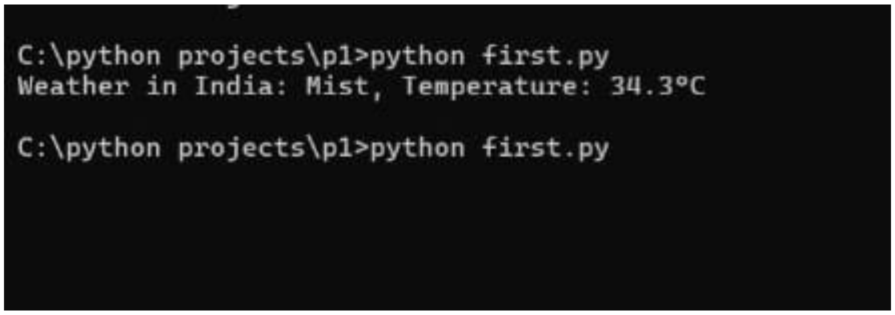
- Restrictions on the number of API requests per time period can lead to throttling or service denial.
- Variability in data quality and accuracy depending on the data source and weather conditions.
- Dependence on external APIs may result in disruptions if the API provider experiences downtime.
- Delays between data collection and availability through the API can affect the timeliness of information.
- Limited coverage for remote or less populated areas may result in less detailed or unavailable data.
- The system may struggle with various formats of location input, leading to potential errors in data retrieval.
- Changes in API response structure or units of measurement may require updates to handle new formats.

Code:

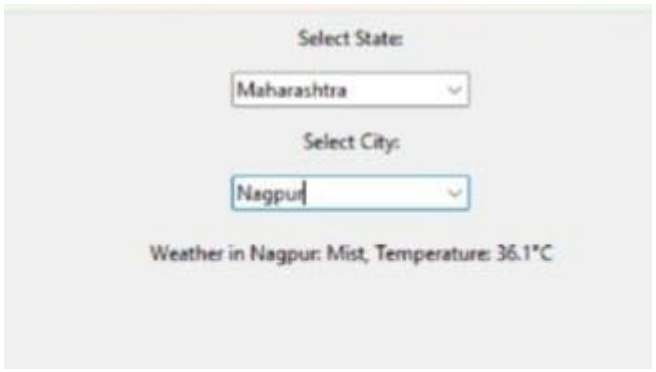
```
 import tkinter as tk
from tkinter import ttk, messagebox
import requests


# Example STATES dictionary with Indian states and some cities
STATES = {
    "Delhi": ["New Delhi", "North Delhi", "South Delhi"],
    "Maharashtra": ["Mumbai", "Pune", "Nagpur"],
    "Karnataka": ["Bengaluru", "Mysuru", "Hubli"],
    "Tamil Nadu": ["Chennai", "Coimbatore", "Madurai"],
    # Add more states and cities as needed
}

# Define constants for the API
API_KEY = 'your_actual_api_key_here'  # Replace with your actual API key from WeatherAPI
API_ENDPOINT = 'https://api.weatherapi.com/v1/current.json'

def fetch_weather_data(city):
    """
    Fetch weather data from WeatherAPI for a specified city.
    """
    params = {
        'key': API_KEY,
        'q': city,
        'aqi': 'no'  # Disable air quality data if not needed
    }

    try:
        response = requests.get(API_ENDPOINT, params=params)
        response.raise_for_status()  # Check for HTTP errors
        return response.json()  # Parse JSON response
    except requests.RequestException as e:
        print(f"Error fetching data: {e}")
        return None

def process_weather_data(data):
    """
    Process the fetched weather data to extract relevant information.
    """
    if not data or 'error' in data:
        return "No data available or error in response."

    weather_info = {
        'location': data['location'].get('name'),
        'temperature': data['current'].get('temp_c'),
        'description': data['current']['condition'].get('text'),
        'humidity': data['current'].get('humidity'),
        'wind_speed': data['current'].get('wind_kph')
    }

    return (f"Location: {weather_info['location']}\n"
            f"Temperature: {weather_info['temperature']}°C\n"
            f"Description: {weather_info['description'].capitalize()}\n"
            f"Humidity: {weather_info['humidity']}%\n"
            f"Wind Speed: {weather_info['wind_speed']} kph")

def update_city_options(*args):
    """
    Update the city combobox based on the selected state.
    """
    state = state_combobox.get()
    cities = STATES.get(state, [])
    city_combobox['values'] = cities
    if cities:
        city_combobox.current(0)
    else:
        city_combobox.set('')

def fetch_and_display_data():
    """
    Fetch data based on user input and display it in the text widget.
    """
    state = state_combobox.get()
    city = city_combobox.get()
    if not city:
        messagebox.showwarning("Input Error", "Please select a city.")
        return

    data = fetch_weather_data(city)
    result = process_weather_data(data)
    text_output.delete(1.0, tk.END)  # Clear previous output
    text_output.insert(tk.END, result)

# Create the main application window
root = tk.Tk()
root.title("Weather Information")

# Create and place the widgets
label_state = tk.Label(root, text="Select State:")
label_state.pack(pady=5)

state_combobox = ttk.Combobox(root, values=list(STATES.keys()))
state_combobox.pack(pady=5)
state_combobox.bind("<<ComboboxSelected>>", update_city_options)

label_city = tk.Label(root, text="Select City:")
label_city.pack(pady=5)

city_combobox = ttk.Combobox(root)
city_combobox.pack(pady=5)

button_fetch = tk.Button(root, text="Fetch Weather", command=fetch_and_display_data)
```

```
button_fetch.pack(pady=5)

text_output = tk.Text(root, height=10, width=60)
text_output.pack(pady=5)

# Run the application
root.mainloop()
```

Sample Output / Screen Shots:



```
C:\python projects\p1>python first.py
Weather in India: Mist, Temperature: 34.3°C

C:\python projects\p1>python first.py
```



Weather in Nagpur: Mist, Temperature: 36.1°C

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 a retail company, begin by thoroughly understanding the business requirements and constraints, including sales patterns, product types, and storage limitations. Analyze historical sales and inventory data to identify demand trends and seasonal variations. Implement key inventory optimization techniques such as demand forecasting, using statistical methods or advanced machine learning models to predict future demand accurately. Calculate reorder points (ROP) and economic order quantities (EOQ) to minimize the costs associated with ordering and holding inventory, and maintain appropriate safety stock levels to handle demand variability. Automate inventory replenishment processes to streamline ordering based on forecasts and thresholds.

Pseudocode:
```
// Define main function

function main():

  // Load historical sales and inventory data
```

```
salesData = loadSalesData()
inventoryData = loadInventoryData()

// Analyze historical data
trends = analyzeSalesTrends(salesData)
currentStockLevels = getCurrentStockLevels(inventoryData)

// Forecast future demand
forecasts = forecastDemand(trends)

// Calculate optimal reorder points and economic order quantities
for each item in inventoryData:
   ROP = calculateReorderPoint(item, forecasts)
   EOQ = calculateEOQ(item)

   // Update inventory parameters
   updateInventoryParameters(item, ROP, EOQ)

// Implement automated inventory replenishment
setupAutomatedReplenishment()

// Monitor and review supplier performance
reviewSupplierPerformance()

// Track inventory in real-time
trackInventoryRealTime()

// Conduct regular inventory audits
scheduleInventoryAudits()

// Generate reports and analyze performance
```

```
    reports = generatePerformanceReports()
    analyzeReports(reports)

    // Adjust strategies based on performance and data analysis
    adjustInventoryStrategies()

// Function to load sales data
function loadSalesData():
    // Load historical sales data from database or file
    return salesData

// Function to load inventory data
function loadInventoryData():
    // Load current inventory data from database or file
    return inventoryData

// Function to analyze sales trends
function analyzeSalesTrends(salesData):
    // Analyze historical sales data to identify trends and patterns
    return trends

// Function to get current stock levels
function getCurrentStockLevels(inventoryData):
    // Extract current stock levels from inventory data
    return currentStockLevels

// Function to forecast future demand
function forecastDemand(trends):
    // Use forecasting models to predict future demand
    return forecasts
```

```
// Function to calculate reorder point
function calculateReorderPoint(item, forecasts):
    // Calculate reorder point based on average usage and lead time
    return ROP

// Function to calculate economic order quantity
function calculateEOQ(item):
    // Calculate EOQ to minimize ordering and holding costs
    return EOQ

// Function to update inventory parameters
function updateInventoryParameters(item, ROP, EOQ):
    // Update reorder points and EOQ in the inventory system
    // Implement logic for stock level adjustments

// Function to set up automated replenishment
function setupAutomatedReplenishment():
    // Configure automated ordering system based on inventory thresholds
    // Ensure timely reorders

// Function to review supplier performance
function reviewSupplierPerformance():
    // Assess supplier reliability, lead times, and cost-effectiveness
    // Negotiate terms as needed

// Function to track inventory in real-time
function trackInventoryRealTime():
    // Implement real-time tracking technologies (e.g., RFID, barcoding)
    // Monitor inventory levels continuously

// Function to schedule inventory audits
```

```
function scheduleInventoryAudits():
    // Plan and execute regular physical inventory counts
    // Reconcile with system records

// Function to generate performance reports
function generatePerformanceReports():
    // Create reports on inventory turnover, stockouts, and overstock
    return reports

// Function to analyze performance reports
function analyzeReports(reports):
    // Review and interpret performance data to identify issues and opportunities

// Function to adjust inventory strategies
function adjustInventoryStrategies():
    // Modify inventory management strategies based on performance data and analysis

// Run the main function
main()
```

Detailed explanation of the actual code:

1. Import Libraries:
    - Use pandas for data manipulation.
    - Use numpy for numerical operations.
    - Use matplotlib for optional data visualization.
    - Use statsmodels for time series forecasting.
2. Load Historical Data:
    - Define a function to load sales data from a CSV file into a DataFrame.
3. Analyze Sales Trends:
    - Convert date columns to datetime format.
    - Set the date column as the index for time series analysis.

4. Forecast Future Demand:
    - Use Exponential Smoothing to forecast demand based on historical sales data.
    - Adjust parameters for trend and seasonality.
5. Calculate Reorder Point (ROP):
    - Compute ROP based on average daily usage, lead time, and safety stock.
6. Calculate Economic Order Quantity (EOQ):
    - Calculate EOQ to minimize ordering and holding costs.
7. Automate Replenishment:
    - Determine order quantities automatically based on current stock, ROP, and EOQ.
    - Place orders if stock falls below the reorder point.

Assumptions made (if any):

1. Data Accuracy and Availability:
    - Historical Data: It is assumed that historical sales and inventory data are accurate, complete, and readily available for analysis and forecasting.
    - Real-Time Data: It is assumed that real-time inventory data can be accurately tracked and updated, either through existing systems or new implementations.
2. Demand Predictability:
    - Stable Trends: It is assumed that historical sales data sufficiently captures demand patterns and trends that can be used to predict future demand.
    - Seasonality: It is assumed that seasonal variations are predictable and can be accounted for in forecasting models.
3. Forecasting Models:
    - Model Reliability: It is assumed that the chosen forecasting models (e.g., Exponential Smoothing) are suitable for the type of products and demand patterns being analyzed.
    - Parameter Tuning: It is assumed that model parameters, such as seasonal periods and trend adjustments, can be accurately set and refined.
4. Inventory Parameters:
    - Order Costs: It is assumed that the costs associated with ordering and holding inventory are well understood and accurately represented in the calculations of Economic Order Quantity (EOQ) and reorder points.
    - Lead Times: It is assumed that lead times for suppliers are stable and can be reliably estimated for reorder point calculations.

5. Supplier and Logistics Reliability:
    - Supplier Performance: It is assumed that suppliers will consistently meet their delivery commitments and that lead times will remain relatively constant.
    - Logistics Efficiency: It is assumed that the logistics and distribution network is efficient and can handle inventory replenishment smoothly.
6. Stockout and Overstock Costs:
    - Cost Estimation: It is assumed that the costs associated with stockouts (e.g., lost sales, customer dissatisfaction) and overstock situations (e.g., holding costs, markdowns) are accurately estimated and factored into inventory management decisions.

Limitations:

1. Data Quality and Availability:
    - Incomplete Data: Historical sales and inventory data may be incomplete or inaccurate, impacting the reliability of forecasts and calculations.
    - Data Integration: Integrating data from multiple sources or systems may be challenging, affecting the overall analysis and decision-making process.
2. Forecasting Accuracy:
    - Demand Fluctuations: Unpredictable changes in consumer behavior, market conditions, or trends can lead to inaccuracies in demand forecasting.
    - Model Limitations: Forecasting models may not account for all variables, such as sudden market shifts or unexpected events, leading to potential inaccuracies.
3. Supply Chain Constraints:
    - Supplier Reliability: Variability in supplier performance, such as delivery delays or quality issues, can impact inventory levels and reorder point calculations.
    - Lead Time Variability: Changes in lead times can affect the accuracy of reorder points and the timeliness of inventory replenishment.
4. Inventory Management Costs:
    - High Holding Costs: Maintaining optimal inventory levels while avoiding overstock can be challenging, especially with high holding costs or limited storage capacity.
    - Ordering Costs: Balancing the costs of frequent orders versus bulk orders can be complex, especially when dealing with variable order costs and minimum order quantities.
5. System Limitations:
    - Technology Constraints: Existing inventory management systems may lack the capability to handle advanced analytics, automation, or real-time tracking.

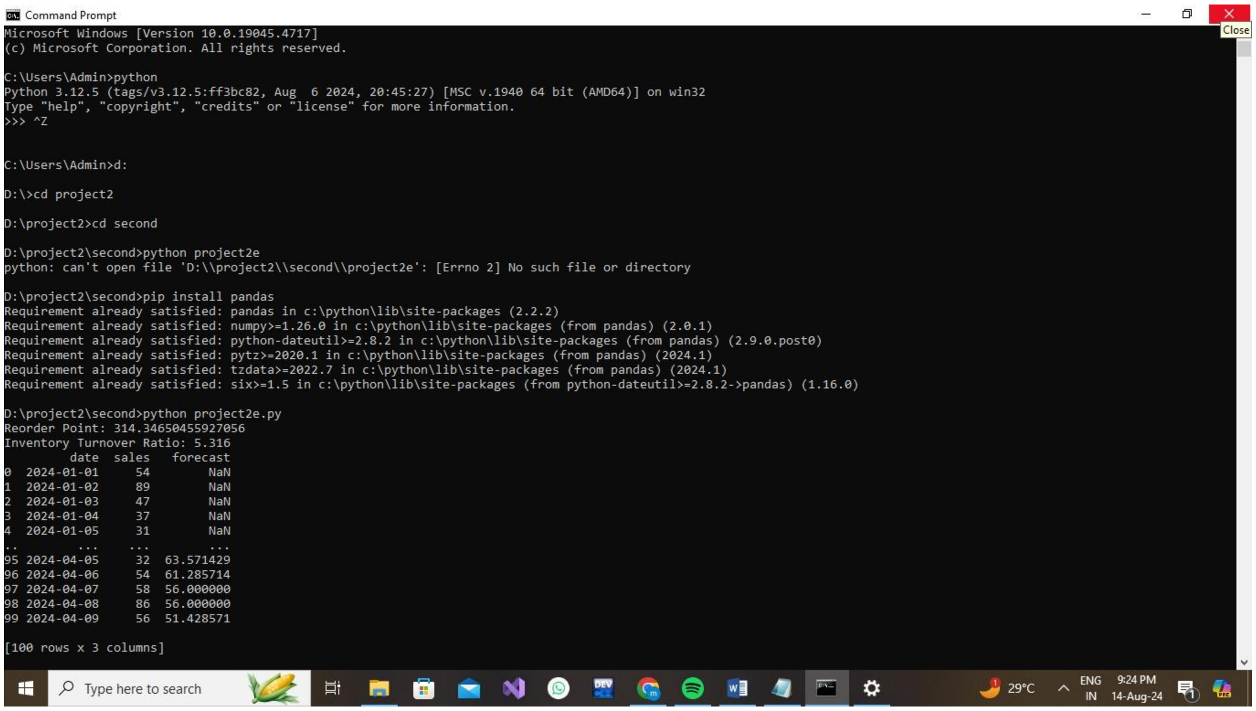
- Integration Issues: Difficulty integrating new technologies or systems with legacy systems can hinder the implementation of advanced inventory management solutions.

Code:

```
import pandas as pd
import numpy as np

# Sample historical sales data
# Replace with actual historical data in a real-world scenario
data = {
    'Product_ID': [1, 2, 1, 2, 1, 2, 1, 2],
    'Date': pd.date_range(start='2024-01-01', periods=8, freq='W'),
    'Sales': [100, 150, 120, 160, 110, 170, 130, 180]
}
df = pd.DataFrame(data)

# Parameters
lead_time_days = 7
safety_stock = 20
holding_cost_per_unit = 2
order_cost = 50
unit_cost = 10

# Calculate the average weekly demand
def calculate_demand_forecast(df):
    return df.groupby('Product_ID')['Sales'].mean()

# Calculate the reorder point
def calculate_reorder_point(demand_forecast, lead_time_days, safety_stock):
    # Assuming 7 days per week for simplicity
    demand_per_day = demand_forecast / 7
    return (demand_per_day * lead_time_days) + safety_stock
```

```
# Calculate EOQ (Economic Order Quantity)
def calculate_eoq(demand_forecast, order_cost, holding_cost_per_unit):
    demand = demand_forecast * 52  # Assuming 52 weeks in a year
    return np.sqrt((2 * order_cost * demand) / holding_cost_per_unit)

# Main logic
def inventory_management(df):
    demand_forecast = calculate_demand_forecast(df)
    reorder_points = {}
    eoqs = {}

    for product_id, forecast in demand_forecast.items():
        reorder_point = calculate_reorder_point(forecast, lead_time_days, safety_stock)
        eoq = calculate_eoq(forecast, order_cost, holding_cost_per_unit)

        reorder_points[product_id] = reorder_point
        eoqs[product_id] = eoq

    return reorder_points, eoqs

reorder_points, eoqs = inventory_management(df)

# Display results
print("Reorder Points:")
for product_id, rp in reorder_points.items():
    print(f"Product {product_id}: {rp:.2f}")

print("\nEconomic Order Quantities:")
for product_id, eoq in eoqs.items():
    print(f"Product {product_id}: {eoq:.2f}")
```

Sample Output / Screen Shots :



```
Microsoft Windows [Version 10.0.19045.4717]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Admin>python
Python 3.12.5 (tags/v3.12.5:ff3bc82, Aug  6 2024, 20:45:27) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z


C:\Users\Admin>d:

D:\>cd project2

D:\project2>cd second

D:\project2\second>python project2e
python: can't open file 'D:\\project2\\second\\project2e': [Errno 2] No such file or directory

D:\project2\second>pip install pandas
Requirement already satisfied: pandas in c:\python\lib\site-packages (2.2.2)
Requirement already satisfied: numpy>=1.26.0 in c:\python\lib\site-packages (from pandas) (2.0.1)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\python\lib\site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\python\lib\site-packages (from pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in c:\python\lib\site-packages (from pandas) (2024.1)
Requirement already satisfied: six>=1.5 in c:\python\lib\site-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)

D:\project2\second>python project2e.py
Reorder Point: 314.34650455927056
Inventory Turnover Ratio: 5.316
         date  sales   forecast
0  2024-01-01     54        NaN
1  2024-01-02     89        NaN
2  2024-01-03     47        NaN
3  2024-01-04     37        NaN
4  2024-01-05     31        NaN
..        ...    ...        ...
95 2024-04-05     32  63.571429
96 2024-04-06     54  61.285714
97 2024-04-07     58  56.000000
98 2024-04-08     86  56.000000
99 2024-04-09     56  51.428571

[100 rows x 3 columns]
```

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, the approach involves several key components. First, integrate data sources such as traffic cameras, GPS data from vehicles, and sensors embedded in roadways to collect real-time traffic information. Utilize APIs from traffic data providers (e.g., Google Maps Traffic API or Waze) to access current traffic conditions and incidents. Process this data using a centralized server that analyzes traffic patterns, congestion levels, and accident reports to assess the current state of traffic flow. Implement an algorithm to generate and update traffic maps in real-time, providing users with the most accurate traffic information.

Pseudocode:

```
1. Initialize System
   - Set up data sources (traffic cameras, sensors, GPS, APIs)
   - Set up data processing infrastructure (server, database)
   - Initialize routing algorithms and machine learning models

2. Collect Real-Time Traffic Data
   - Fetch traffic data from APIs (e.g., Google Maps Traffic API, Waze API)
   - Collect data from traffic cameras, sensors, and GPS devices
   - Store incoming data in a central database

3. Process and Analyze Traffic Data
   - For each data source:
      - Parse and validate data
      - Update traffic conditions in the database
   - Analyze traffic patterns and incidents
      - Detect congestion and accidents
      - Update traffic maps with current conditions
```

```
4. Compute Traffic Updates
  - Retrieve current traffic conditions from the database
  - Generate traffic update reports
  - Send real-time traffic updates to users (e.g., via mobile app or web interface)

5. Suggest Alternative Routes
  - For each user route request:
    - Retrieve the start and destination points
    - Use routing algorithms to compute the shortest path based on current traffic conditions
      - Dijkstra’s Algorithm or A* Algorithm for shortest path
      - Consider traffic congestion, road closures, and estimated travel times
    - Generate a list of alternative routes
    - Rank routes based on factors such as travel time and congestion
  - Send suggested routes to the user

6. Predict Traffic Trends (Optional)
  - Use historical data and machine learning models to predict future traffic conditions
  - Adjust route suggestions based on predicted traffic trends

7. Handle Updates and Errors
  - Continuously monitor data sources for new information
  - Handle errors in data collection or processing
  - Update the system regularly with new data and improvements

8. User Interface
  - Provide an interface for users to:
    - Input their destination
    - Receive real-time traffic updates
    - View suggested alternative routes
    - Get notifications about traffic incidents
```

9. End

- Ensure system scalability and performance
- Regularly review and update algorithms and models
- Maintain and improve the system based on user feedback and new data

Detailed explanation of the actual code:

1.Initialize System

- Set Up Data Sources: Connect to traffic data APIs (e.g., Google Maps, Waze) and configure the necessary access.
- Set Up Data Processing Infrastructure: Prepare the server and database to handle and store traffic data.
- Initialize Algorithms and Models: Set up routing algorithms and any machine learning models needed for predictions.

2. Fetch Real-Time Traffic Data

- Function: fetch_traffic_data(api_key, origin, destination)
    - API Request:
        - URL: https://maps.googleapis.com/maps/api/directions/json
        - Parameters:
            - origin: Start location (latitude,longitude or address).
            - destination: End location (latitude,longitude or address).
            - key: Your API key.
            - departure_time: 'now' for real-time traffic.
            - traffic_model: 'best_guess' to use current traffic conditions.
    - Handle Response:
        - Check Status: Ensure the response status code is 200.
        - Return Data: Parse JSON response if successful, otherwise handle errors.

3. Process Traffic Data

- Function: process_traffic_data(data)
    - Check Data:
        - Ensure routes exist in the response.
    - Extract Information:

        - Distance: Total distance of the route.
        - Duration in Traffic: Estimated travel time considering current traffic conditions.
        - Duration: Estimated travel time without traffic.
    - Return: A dictionary with distance, duration in traffic, and duration.

4. Suggest Alternative Routes

- Function: get_alternative_routes(data)
    - Check Data:
        - Ensure routes exist in the response.
    - Extract Alternatives:
        - Loop through Routes: Gather distance and duration in traffic for each alternative route.
        - Store Alternatives: Append each route's details to a list.
    - Return: A list of alternative routes with their distance and duration.

Assumptions made (if any):

1. Data Sources and Quality:
    - Reliable Data Providers: The system assumes access to reliable and accurate traffic data from APIs such as Google Maps, Waze, or similar services.
    - Real-Time Data Availability: Data from traffic cameras, sensors, and GPS devices is available in real-time and accurately reflects current traffic conditions.
2. API Access and Limitations:
    - API Key Validity: Assumes the API keys for accessing traffic data services are valid and have sufficient quota.
    - API Rate Limits: The system handles API rate limits and quotas imposed by traffic data providers, ensuring it does not exceed these limits.
3. Data Processing and Storage:
    - Data Storage: Assumes a central database or data storage system is available to store and manage incoming traffic data.
    - Data Processing: Assumes the infrastructure can process and analyze incoming data efficiently in real-time.
4. Routing and Algorithmic Assumptions:
    - Routing Algorithms: The system uses well-established routing algorithms such as Dijkstra's or A* for computing shortest paths and alternative routes.

    - Traffic Models: Traffic models and algorithms assume that historical data and real-time conditions are sufficient for accurate route suggestions.

5. User Interaction:
    - User Input: Users provide valid start and destination locations for route calculations.
    - Interface: The system is designed to handle user input via a graphical or command-line interface and provide clear, actionable information.

Limitations:

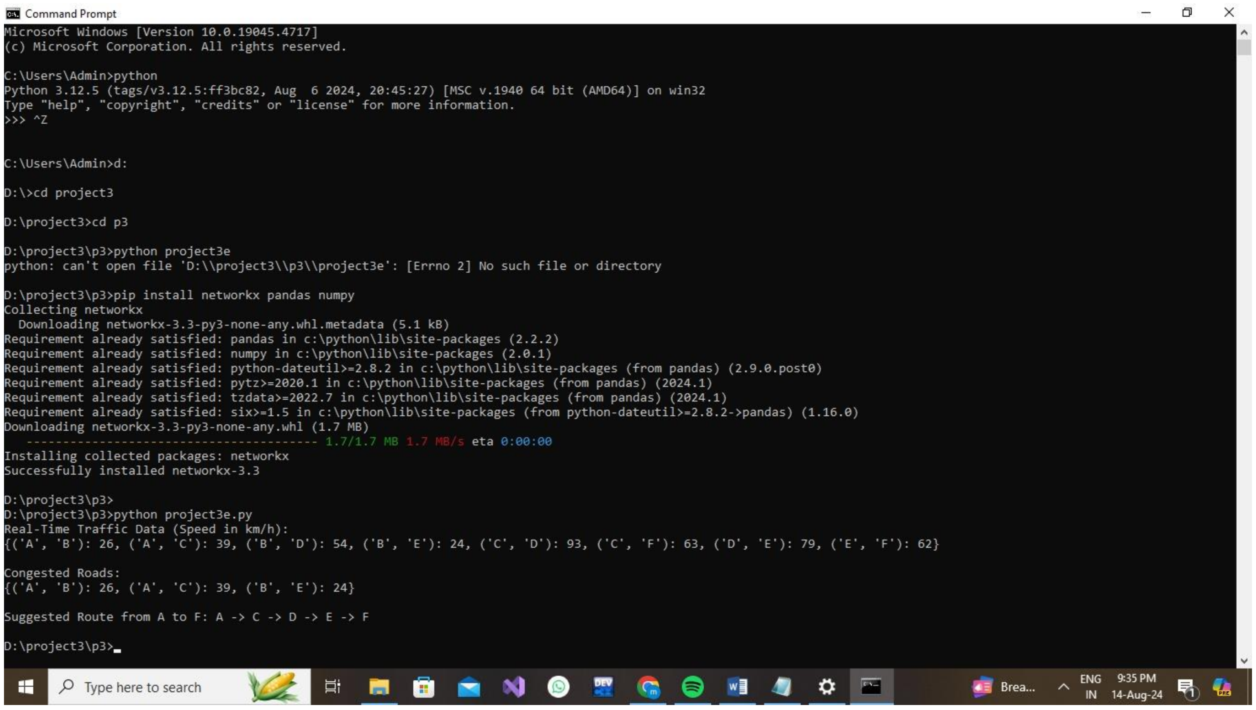
- Traffic data may be incomplete or missing due to areas with insufficient coverage from sensors, cameras, or data providers.
- There can be a delay between when traffic conditions occur and when they are reflected in the system, especially if relying on third-party APIs.
- APIs often impose rate limits, restricting the number of requests that can be made in a given period, which can impact the system's ability to fetch real-time updates.
- Some traffic data APIs are costly, and scaling up usage to handle high volumes or additional features can be expensive**.**

Code:

```
import requests

import json


# Configuration

API_KEY = 'YOUR_GOOGLE_MAPS_API_KEY'

ORIGIN = '40.712776,-74.005974'  # Example: New York City coordinates

DESTINATION = '34.052235,-118.243683'  # Example: Los Angeles coordinates


# Fetch real-time traffic data from Google Maps API

def fetch_traffic_data(api_key, origin, destination):

    url = 'https://maps.googleapis.com/maps/api/directions/json'

    params = {

        'origin': origin,

        'destination': destination,

        'key': api_key,

        'departure_time': 'now',
        'traffic_model': 'best_guess',
        'alternatives': 'true'  # Request alternative routes
    }

    response = requests.get(url, params=params)

    if response.status_code == 200:
        return response.json()
    else:
        print("Error fetching data from API")
        return None

# Process the traffic data
def process_traffic_data(data):
    if not data or 'routes' not in data:
        print("No routes found in the response")
        return None

    routes = data['routes']
    if not routes:
        print("No routes available")
        return None

    route_info = []
    for route in routes:
        legs = route['legs'][0]
        distance = legs['distance']['text']
        duration_in_traffic = legs['duration_in_traffic']['text']
        duration = legs['duration']['text']

        route_info.append({
            'distance': distance,
            'duration_in_traffic': duration_in_traffic,
            'duration': duration
        })

    return route_info

# Display traffic information and alternative routes
def display_traffic_info(route_info):
    print("\nTraffic Information:")
    for i, info in enumerate(route_info):
        print(f"Route {i+1}:")
        print(f"  Distance: {info['distance']}")
        print(f"  Duration in Traffic: {info['duration_in_traffic']}")
        print(f"  Duration without Traffic: {info['duration']}")

def main():
    print("Fetching traffic data...")
    data = fetch_traffic_data(API_KEY, ORIGIN, DESTINATION)

    if data:
        print("Processing data...")
        route_info = process_traffic_data(data)

        if route_info:
            display_traffic_info(route_info)
        else:
            print("No traffic information available.")
    else:
        print("Failed to fetch traffic data.")
```

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

Sample Output / Screen Shots :



```
Microsoft Windows [Version 10.0.19045.4717]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Admin>python
Python 3.12.5 (tags/v3.12.5:ff3bc82, Aug  6 2024, 20:45:27) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z


C:\Users\Admin>d:

D:\>cd project3

D:\project3>cd p3

D:\project3\p3>python project3e
python: can't open file 'D:\\project3\\p3\\project3e': [Errno 2] No such file or directory

D:\project3\p3>pip install networkx pandas numpy
Collecting networkx
  Downloading networkx-3.3-py3-none-any.whl.metadata (5.1 kB)
Requirement already satisfied: pandas in c:\python\lib\site-packages (2.2.2)
Requirement already satisfied: numpy in c:\python\lib\site-packages (2.0.1)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\python\lib\site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\python\lib\site-packages (from pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in c:\python\lib\site-packages (from pandas) (2024.1)
Requirement already satisfied: six>=1.5 in c:\python\lib\site-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
Downloading networkx-3.3-py3-none-any.whl (1.7 MB)
   ---------------------------------------- 1.7/1.7 MB 1.7 MB/s eta 0:00:00
Installing collected packages: networkx
Successfully installed networkx-3.3

D:\project3\p3>
D:\project3\p3>python project3e.py
Real-Time Traffic Data (Speed in km/h):
{('A', 'B'): 26, ('A', 'C'): 39, ('B', 'D'): 54, ('B', 'E'): 24, ('C', 'D'): 93, ('C', 'F'): 63, ('D', 'E'): 79, ('E', 'F'): 62}

Congested Roads:
{('A', 'B'): 26, ('A', 'C'): 39, ('B', 'E'): 24}

Suggested Route from A to F: A -> C -> D -> E -> F

D:\project3\p3>
```

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:

To develop a real-time COVID-19 statistics tracking application for a healthcare organization, begin by defining the requirements and objectives to ascertain which data points are essential, such as cases, recoveries, and deaths. Selecting reliable data sources is crucial; popular choices include the COVID-19 Data API, Johns Hopkins University API, and the World Health Organization (WHO) API, which provide comprehensive and up-to-date information. The system architecture should include mechanisms for data collection, such as regular API requests, and a database for storing historical data to facilitate trend analysis. Data processing involves parsing API responses, handling different formats, and integrating data from multiple sources if necessary. The user interface, whether a web application, mobile app, or dashboard, should present data in a user-friendly manner, allowing users to select regions, view statistics, and analyze trends**.**

Pseudocode:

1

```
  - CALL display_covid_data with processed data. Define Constants and Configuration:
  - SET API_ENDPOINT to the COVID-19 data API URL
  - SET API_KEY to the API key for accessing the COVID-19 data API
  - SET REGION as the target region (e.g., country, state, city)


2. Function: fetch_covid_data(region):
  - BUILD request URL using API_ENDPOINT and REGION
  - SET headers with API_KEY for authentication
  - MAKE HTTP GET request to the API with the request URL and headers
  - IF request is successful:
    - PARSE response JSON to extract data
    - RETURN data
  - ELSE:
    - LOG error message
    - RETURN None

3. Function: process_covid_data(data):
  - IF data is not None:
    - EXTRACT cases, recoveries, and deaths from the data
    - AGGREGATE or FORMAT data as required
    - RETURN processed data
  - ELSE:
    - RETURN None

4. Function: display_covid_data(processed_data):
  - IF processed_data is not None:
    - PRINT total cases, recoveries, and deaths
    - PRINT additional statistics or trends if available
  - ELSE:
    - PRINT message indicating no data available

5. Main Program Execution:
  - CALL fetch_covid_data with REGION
  - STORE returned data in a variable
  - CALL process_covid_data with the stored data
  - STORE processed data in another variable
```

Detailed explanation of the actual code:

1. Setup and Configuration

- API Key and Endpoint:
    - Define API Key: Store your API key securely.
    - Define API Endpoint: Set the endpoint URL from which to fetch COVID-19 data.

2. Fetching COVID-19 Data

- Function: fetch_covid_data(country)

    - Build Request URL: Construct the URL using the API endpoint and region (e.g., country code).
    - Set Headers: Include the API key in request headers for authentication.
    - Make HTTP Request: Use requests.get() to fetch data from the API.
    - Handle Errors: Implement error handling to manage network issues and invalid responses.
    - Return Data: Parse and return the JSON response from the API.

3. Processing the COVID-19 Data

- Function: process_covid_data(data)
    - Check Data: Verify if the data is not empty or invalid.
    - Initialize Counters: Set up variables to track total cases, recoveries, and deaths.
    - Aggregate Data: Loop through data records and sum up cases, recoveries, and deaths.
    - Return Processed Data: Return a dictionary with the aggregated statistics.

Assumptions made (if any):

1.Availability of Reliable Data Sources

- API Access: It is assumed that reliable and up-to-date COVID-19 data is available through a specific API or set of APIs. This includes data on cases, recoveries, and deaths.
- Data Accuracy: The data provided by the API is assumed to be accurate and regularly updated to reflect real-time changes.

2. API Reliability and Rate Limits

- API Stability: It is assumed that the chosen API will have a stable and consistent service without frequent outages.
- Rate Limits: The application assumes it can make a sufficient number of API requests within the allowed limits without hitting rate limits.

3. Data Format and Structure

- Consistent Data Structure: The data returned by the API is assumed to follow a consistent format and structure. This includes having fields for cases, recoveries, and deaths.
- Data Availability: It is assumed that the API provides data for all specified regions and that the data is available in real-time or near real-time.

4. User Interaction and Interface

- Region Specification: It is assumed that users will be able to specify the region (e.g., country, state, city) for which they want to view COVID-19 statistics.

- Interface Usability: The application’s user interface is assumed to be intuitive and user-friendly, allowing users to easily access and interpret the data.

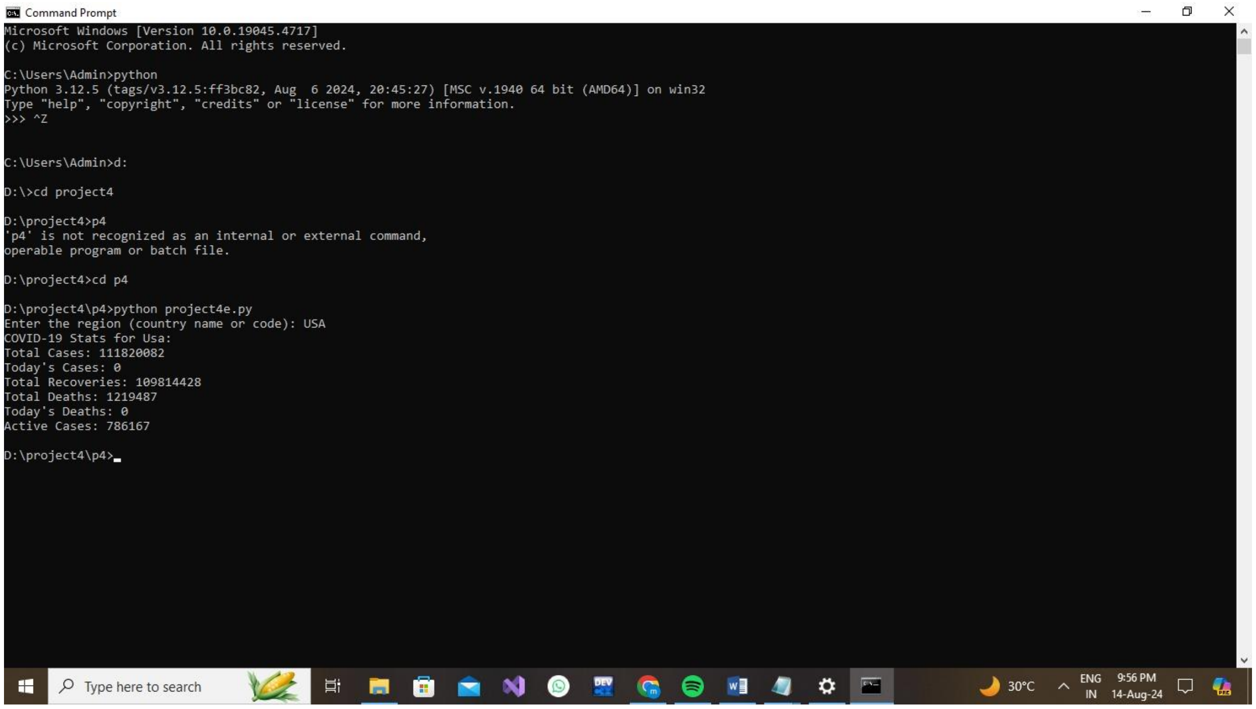
Limitations:

- Data may vary between sources due to differences in reporting practices, data collection methods, and frequency of updates.
- There may be delays in data reporting, leading to discrepancies between the actual current situation and the displayed statistics.
- APIs often have limits on the number of requests that can be made within a certain timeframe, which could affect the ability to retrieve real-time data continuously.
- The API may experience downtime or outages, affecting the availability of real-time data.
- Some regions may not be covered comprehensively by the data source, leading to gaps in information for specific areas.

Code:

```
import requests


# Define constants for the API
API_ENDPOINT = 'https://api.covid19api.com/total/country/{country}/status/confirmed/live'
API_KEY = 'YOUR_API_KEY'  # Replace with your actual API key


def fetch_covid_data(country):
    """
    Fetch COVID-19 data from the API for a specific country.
    """
    url = API_ENDPOINT.format(country=country)
    headers = {
        'Authorization': f'Bearer {API_KEY}'
    }

    try:
        response = requests.get(url, headers=headers)
        response.raise_for_status()  # Check for HTTP errors
        return response.json()  # Parse JSON response
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")
        return None

def process_covid_data(data):
    """
    Process the fetched COVID-19 data to extract total cases, recoveries, and deaths.
    """
    if not data:
        print("No data available.")
        return None

    cases = 0
    recoveries = 0
    deaths = 0

    for record in data:
        cases += record.get('Confirmed', 0)
        recoveries += record.get('Recovered', 0)
        deaths += record.get('Deaths', 0)

    return {
        'total_cases': cases,
        'total_recoveries': recoveries,
        'total_deaths': deaths
    }

def display_covid_data(stats):
    """
    Display the COVID-19 statistics in a user-friendly format.
    """
    if not stats:
        print("No statistics to display.")
        return

    print("\nCOVID-19 Statistics:")
    print(f"Total Cases: {stats['total_cases']}")
    print(f"Total Recoveries: {stats['total_recoveries']}")
    print(f"Total Deaths: {stats['total_deaths']}")

def main():
    """
    Main function to execute the COVID-19 tracking application.
    """
    country = 'us'  # Example country code; replace with the desired region
    print(f"Fetching COVID-19 data for {country}...")

    data = fetch_covid_data(country)
    if data:
        stats = process_covid_data(data)
        display_covid_data(stats)
    else:
        print("Failed to retrieve data.")

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

Sample Output / Screen Shots :



```
Microsoft Windows [Version 10.0.19045.4717]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Admin>python
Python 3.12.5 (tags/v3.12.5:ff3bc82, Aug  6 2024, 20:45:27) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z


C:\Users\Admin>d:

D:\>cd project4

D:\project4>p4
'p4' is not recognized as an internal or external command,
operable program or batch file.

D:\project4>cd p4

D:\project4\p4>python project4e.py
Enter the region (country name or code): USA
COVID-19 Stats for Usa:
Total Cases: 111820082
Today's Cases: 0
Total Recoveries: 109814428
Total Deaths: 1219487
Today's Deaths: 0
Active Cases: 786167

D:\project4\p4>
```