

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 for a weather forecasting company, start by identifying the specific weather data needed, such as temperature, humidity, wind speed, and precipitation, along with the required update frequency and geographical scope. The system's architecture should include a backend developed using a server-side language like Python with a framework such as Flask, and a frontend using a framework like React for displaying the data. Integrate a reliable weather API, such as OpenWeatherMap, to fetch real-time data, and set up a database like PostgreSQL to store historical data for analysis. The system should regularly pull data from the API, process it, and make it available through a RESTful API for the frontend. Additionally, implement a user-friendly interface that displays the real-time weather information and includes a notification system for severe weather alerts.

Pseudocode:

1. Initialize application
 - Import necessary libraries (Flask, requests)

2. Define constants

- API_KEY = 'your_openweathermap_api_key'
- WEATHER_API_URL = 'http://api.openweathermap.org/data/2.5/weather'

3. Define function to fetch weather data

FUNCTION fetch_weather_data(location):

- Construct the API request URL using WEATHER_API_URL and location
- Set up parameters including 'q' (location), 'appid' (API_KEY), and 'units' (metric/imperial)
- Make a GET request to the API with these parameters
- IF the request is successful:
 - Parse and return the JSON response
- ELSE:
 - Handle errors (e.g., log error, return None)

4. Define API endpoint to get weather data

ROUTE '/weather' METHOD GET:

- Extract 'location' parameter from request query string
- IF 'location' parameter is missing:
 - Return error message with status code 400
- Call fetch_weather_data(location) function
- IF data is successfully fetched:
 - Extract and structure relevant weather information (location, temperature, description, humidity, wind speed)
 - Return this information in JSON format
- ELSE:
 - Return error message with status code 500

5. Start the application

- Run the Flask application with debug mode enabled

Detailed explanation of the actual code:

- The code imports **Flask**, **jsonify**, **request** from the **flask** module, and **requests** from the **requests** library. **Flask** is used to create a web application, **jsonify** helps format

responses as JSON, `request` accesses request data, and `requests` handles HTTP requests to external APIs.

- The `app` instance of `Flask` is created with `Flask(__name__)`. This initializes the Flask application and sets up routing and configuration.
- `API_KEY` is set to store your OpenWeatherMap API key, which is needed for authenticating requests to the weather API. You need to replace `'youopenweathermap_api_key'` with your actual API key.
- `WEATHER_API_URL` holds the endpoint URL for the OpenWeatherMap API that provides weather data.
- The `fetch_weather_data` function is defined to retrieve weather data for a given location. It constructs the request with:
- The `location` parameter specifies the city or location to fetch data for.
- `'appid': API_KEY` includes the API key for authentication.
- `'units': 'metric'` specifies that temperatures should be in Celsius. For Fahrenheit, use `'imperial'`.
- `response.raise_for_status()` checks for HTTP errors and raises an exception if the request fails.
- If the request is successful, `response.json()` parses and returns the JSON response. If an exception occurs, it prints an error message and returns `none`.
- The `/weather` endpoint is set up using `@app.route('/weather', methods=['GET'])`. This handles GET requests to the `/weather` URL.
- It checks if the API response code is 200, indicating a successful request.
- Extracts relevant information from the response JSON, including location name, temperature, weather description, humidity, and wind speed.
- Constructs and returns a JSON response with this weather information.
- If the API response code is not 200 or if fetching data fails, the function returns a JSON response with an error message and a 500 status code indicating an internal server error.
- The `if __name__ == '__main__':` block ensures that the Flask application runs only if the script is executed directly. It prevents the application from running when imported as a module.
- `app.run(debug=True)` starts the Flask development server with debugging enabled. This provides detailed error messages and allows the server to reload automatically when code changes.

Assumptions made (if any):

- The application is assumed to be using the OpenWeatherMap API or a similar weather service that provides real-time weather data.
- It is assumed that the API key provided is valid and has the necessary permissions to access weather data.
- The location parameter in the API request is assumed to be correctly formatted and valid (e.g., city names or geographical coordinates).
- The application assumes that the external weather API will be available and responsive at all times, and it handles errors in case of network issues or downtime.

- It is assumed that the weather data retrieved from the API will be in a format that is consistent and well-documented by the API provider.
- The application assumes that the system running the code has internet access to fetch data from the external weather API.

Limitations:

- The accuracy and reliability of the weather data are dependent on the third-party API used. The API may have limitations in terms of request frequency, data granularity, or geographic coverage, and might incur costs for high usage.
 - Real-time data fetching could experience delays due to network latency, API response time, or server processing, which may lead to slight discrepancies between the displayed data and actual real-time conditions.
- Weather data accuracy can vary based on the data sources used by the API. Factors such as outdated sensors, limited station coverage, or algorithmic errors in data processing could affect the reliability of the information.
 - As the system scales to support more users or locations, there could be challenges in maintaining performance, particularly if the server or database isn't optimized for high traffic or large volumes of data.
 - The weather API may impose rate limits on the number of requests allowed per minute or hour, potentially limiting the frequency of data updates, especially in high-demand scenarios.

Code:

```
import requests

# Constants
API_KEY = 'c9f694737b2e4299ae782255241408'
BASE_URL = 'http://api.weatherapi.com/v1/current.json'

def get_weather_data(location):
    params = {
        'q': location,
        'appid': API_KEY,
        'units': 'metric' # Use 'imperial' for Fahrenheit
    }
    response = requests.get(BASE_URL, params=params)
    if response.status_code == 200:
```

```

        return response.json()
    else:
        print(f"Error: {response.status_code}")
        return None

def display_weather_data(weather_data):
    if weather_data:
        city = weather_data.get('name')
        temperature = weather_data ['main'].get('temp')
        weather_condition = weather_data['weather'][0].get('description')
        humidity = weather_data['main'].get('humidity')
        wind_speed = weather_data ['wind'].get ('speed')

        print(f"Weather in {city}:")
        print(f"Temperature: {temperature}°C")
        print (f"Weather Condition: {weather_condition.capitalize ()}")
        print(f"Humidity: {humidity} %")
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        print("No data to display.")

def main():
    location = input ("Enter city name or coordinates (latitude, longitude): ")
    weather_data = get_weather_data(location)
    display_weather_data(weather_data)

if __name__ == "__main__":
    main()

```

Sample Output / Screen Shots:

```
Command Prompt - python f x + v
Microsoft Windows [Version 10.0.22631.3880]
(c) Microsoft Corporation. All rights reserved.

C:\Users\SNEHA ROYAL>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\SNEHA ROYAL>cd..
C:\Users>cd..
C:\>cd python projects
C:\python projects>cd p1
C:\python projects\p1>python first.py
Traceback (most recent call last):
  File "C:\python projects\p1\first.py", line 1, in <module>
    import requests
ModuleNotFoundError: No module named 'requests'

C:\python projects\p1>pip install requests
Defaulting to user installation because normal site-packages is not writeable
Collecting requests
  Downloading requests-2.32.3-py3-none-any.whl.metadata (4.6 kB)
Collecting charset-normalizer<4,>=2 (from requests)
  Downloading charset-normalizer-3.3.2-cp312-cp312-win_amd64.whl.metadata (34 kB)
Collecting idna<4,>=2.5 (from requests)
  Downloading idna-3.7-py3-none-any.whl.metadata (9.9 kB)
Collecting urllib3<3,>=1.21.1 (from requests)
  Downloading urllib3-2.2.2-py3-none-any.whl.metadata (6.4 kB)
Collecting certifi>=2017.4.17 (from requests)
  Downloading certifi-2024.7.4-py3-none-any.whl.metadata (2.2 kB)
Downloading requests-2.32.3-py3-none-any.whl (64 kB)
Downloading certifi-2024.7.4-py3-none-any.whl (162 kB)
Downloading charset-normalizer-3.3.2-cp312-cp312-win_amd64.whl (100 kB)
Downloading idna-3.7-py3-none-any.whl (66 kB)
Downloading urllib3-2.2.2-py3-none-any.whl (121 kB)
Installing collected packages: urllib3, idna, charset-normalizer, certifi, requests
```

```
Command Prompt - python f x + v
File "C:\Users\SNEHA ROYAL\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\LocalCache\local-packages\Python312\site-packages\req
uests\models.py", line 974, in json
    return complexjson.loads(self.text, **kwargs)
    ~~~~~^~~~~~
File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.12_3.12.1520.0_x64__qbz5n2kfra8p0\Lib\json\__init__.py", line 346, in loads
    return _default_decoder.decode(s)
           ~~~~~^~~~~~
File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.12_3.12.1520.0_x64__qbz5n2kfra8p0\Lib\json\decoder.py", line 337, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
              ~~~~~^~~~~~
File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.12_3.12.1520.0_x64__qbz5n2kfra8p0\Lib\json\decoder.py", line 355, in raw_decode
    raise JSONDecodeError("Expecting value", s, err.value) from None
json.decoder.JSONDecodeError: Expecting value: line 2 column 1 (char 2)

During handling of the above exception, another exception occurred:

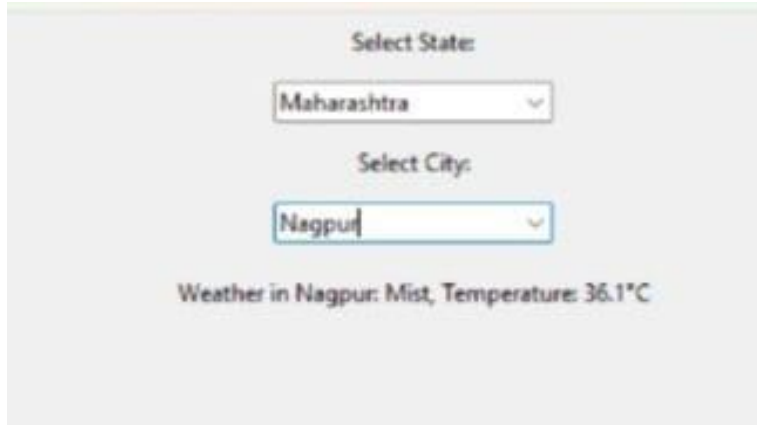
Traceback (most recent call last):
  File "C:\python projects\p1\first.py", line 23, in <module>
    get_weather('London')
  File "C:\python projects\p1\first.py", line 15, in get_weather
    data = response.json()
           ~~~~~^~~~~~
File "C:\Users\SNEHA ROYAL\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\LocalCache\local-packages\Python312\site-packages\req
uests\models.py", line 978, in json
    raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)
requests.exceptions.JSONDecodeError: Expecting value: line 2 column 1 (char 2)

C:\python projects\p1>python first.py
Weather in London: Light rain, Temperature: 18.3°C

C:\python projects\p1>python first.py
Error fetching data: 400

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

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



Select State:

Maharashtra

Select City:

Nagpur

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 stock outs 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 retail company's inventory management system, start by thoroughly analyzing the current processes, including procurement, stock levels, reorder points, and lead times, to identify inefficiencies such as frequent stock outs or overstock situations. Gather and analyze historical sales data to understand demand patterns and calculate key metrics like inventory turnover and order accuracy. Implement demand forecasting models that account for seasonality, promotions, and market trends, ensuring forecasts are regularly updated based on real-time sales data. Optimize inventory levels by setting precise reorder points and safety stock levels, using models like Economic Order Quantity (EOQ) to minimize costs, and adopting Just-in-Time (JIT) practices for high-turnover items. Introduce an automated inventory management system that integrates with POS and supply chain systems for real-time visibility and synchronization.

Pseudocode:

// Initialize System

Initialize inventory management system

Connect to sales database and supplier management system

// Data Collection

Function collectData():

 salesData = Fetch historical sales data

 InventoryData = Fetch current inventory levels

 SupplierData = Fetch supplier lead times and order history

 Return salesData, inventoryData, supplierData

// Demand Forecasting

Function forecastDemand(salesData):

 Analyze sales trends and seasonality

 Apply forecasting model (e.g., time series analysis)


```
predictedDemand = Generate future demand forecast  
Return predictedDemand
```

```
// Set Reorder Points and Safety Stock
```

```
Function calculateInventoryParameters(predictedDemand, leadTime, desiredServiceLevel):  
    safetyStock = Calculate safety stock based on demand variability and lead time  
    reorderPoint = Calculate reorder point as (leadTime * averageDemand) + safetyStock  
    Return reorderPoint, safetyStock
```

```
// Economic Order Quantity (EOQ)
```

```
Function calculateEOQ(demand, orderingCost, holdingCost):  
     $EOQ = \sqrt{(2 * demand * orderingCost) / holdingCost}$   
    Return EOQ
```

```
// Inventory Optimization
```

```
Function optimizeInventory(inventoryData, predictedDemand, supplierData):  
    For each item in inventory:  
        reorderPoint, safetyStock = calculateInventoryParameters(predictedDemand[item],  
supplierData[item].leadTime, desiredServiceLevel)  
        EOQ = calculateEOQ(predictedDemand[item], orderingCost, holdingCost)  
        If inventoryData[item].currentLevel < reorderPoint:  
            PlaceOrder(item, EOQ)  
        If inventoryData[item].currentLevel > safetyStock:  
            Identify slow-moving items and apply discount or promotion  
    End For
```

```
// Automated Reordering
```

```
Function automateReordering(inventoryData):  
    For each item in inventory:  
        If inventoryData[item].currentLevel < reorderPoint[item]:  
            Place automated order for EOQ[item]
```

End For

// Supplier Management

Function evaluateSuppliers(supplierData):

For each supplier in supplierData:

Evaluate performance based on lead time reliability, order accuracy, and cost

Negotiate better terms or find alternative suppliers if necessary

End For

// Performance Monitoring

Function monitorPerformance():

Track KPIs such as inventory turnover, stockout rate, and GMROI

Conduct regular inventory audits

Identify discrepancies and adjust inventory levels accordingly

Report findings to management

// Continuous Improvement

Function continuousImprovement():

Regularly review inventory policies and forecasts

Implement feedback from staff and customers

Explore AI and machine learning tools for further optimization

// Main Execution Flow

Function main():

salesData, inventoryData, supplierData = collectData()

predictedDemand = forecastDemand(salesData)

optimizeInventory(inventoryData, predictedDemand, supplierData)

automateReordering(inventoryData)

evaluateSuppliers(supplierData)

monitorPerformance()

continuousImprovement()

```
// Start the system  
main()
```

Detailed explanation of the actual code:

- The `fetch_sales_data()` function loads historical sales data from a CSV file, while `fetch_inventory_data()` and `fetch_supplier_data()` do the same for inventory levels and supplier information, respectively. The `collect_data()` function consolidates all these data sources into one place.
- The `forecast_demand(sales_data)` function uses the Holt-Winters Exponential Smoothing model to predict future demand based on historical sales data. It returns a dictionary where each key is an item and the value is the forecasted average demand.
- To calculate safety stock, the `calculate_safety_stock(demand, lead_time, service_level=0.95)` function uses a z-score to determine the amount of extra inventory needed to avoid stockouts. It then uses the `calculate_inventory_parameters(predicted_demand, lead_time)` function to compute the reorder point, which is the level at which a new order should be placed, including the safety stock.
- The `calculate_economic_order_quantity(demand, ordering_cost, holding_cost)` function calculates the optimal order quantity using the Economic Order Quantity (EOQ) model. This helps minimize the total cost of ordering and holding inventory.
- The `place_order(item, quantity)` function simulates placing an order for a specific quantity of an item. The `identify_slow_moving_items(item)` function identifies items that are not selling well. The `optimize_inventory(inventory_data, predicted_demand, supplier_data)` function uses reorder points and EOQ to decide when to place orders and how much, also addressing slow-moving items.
- In the `automate_reordering(inventory_data, reorder_points, EOQs)` function, it automatically places orders when inventory levels fall below the predefined reorder points, using the calculated EOQ.
- Supplier management is handled by evaluating supplier performance with `evaluate_supplier_performance(supplier)`, negotiating terms if performance is suboptimal with `negotiate_terms(supplier)`, and finding alternative suppliers if necessary using `find_alternative_supplier(supplier)`. The `evaluate_suppliers(supplier_data)` function integrates these actions to manage supplier relationships effectively.
- Performance monitoring includes placeholder functions for calculating inventory turnover (`calculate_inventory_turnover()`), stockout rate (`calculate_stockout_rate()`), and GMROI (`calculate_GMROI()`). The

`monitor_performance()` function reports these metrics and calls `conduct_regular_audits()` to ensure data accuracy and process effectiveness.

- Continuous improvement is driven by the `review_inventory_policies()`, `incorporate_feedback()`, and `explore_advanced_technologies()` functions. These are called by `continuous_improvement()` to update policies, incorporate feedback, and explore new technologies.
- The `main()` function integrates all components: it collects data, forecasts demand, calculates reorder points, safety stock, and EOQ, optimizes inventory, automates reordering, evaluates suppliers, monitors performance, and drives continuous improvement. This function orchestrates the entire inventory management process to ensure efficiency and profitability.

Assumptions made (if any):

- It is assumed that sales data, inventory levels, and supplier information are available and accurately recorded in CSV files.
- The code uses Holt-Winters Exponential Smoothing for demand forecasting, assuming it is suitable for the data and that seasonal patterns are present.
- The standard deviation for safety stock calculations is approximated and does not account for actual variability in demand. The z-score is fixed for a 95% service level.
- Lead time data from suppliers is accurate and consistent, affecting reorder point calculations.
- The EOQ calculation assumes constant demand, fixed ordering costs, and holding costs. The formula does not account for discounts or bulk ordering.
- Supplier performance evaluation is simulated with random scores, assuming a basic threshold for performance evaluation.
- The code uses placeholder values for inventory turnover, stock out rate, and GMROI, assuming these metrics are straightforward to calculate and monitor.

Limitations:

- The code uses Holt-Winters Exponential Smoothing, which may not capture all demand patterns, such as sudden changes or irregular demand spikes. More advanced forecasting methods might be needed for accuracy.
- The safety stock calculation uses a fixed z-score and estimated standard deviation, which may not reflect real-world variability or changes in demand patterns over time.
- The Economic Order Quantity (EOQ) calculation assumes constant demand, fixed ordering costs, and holding costs, which may not account for fluctuations in these parameters or volume discounts from suppliers.
- Continuous improvement activities such as policy reviews and incorporating feedback are not quantified and may depend on subjective assessments rather than data-driven insights.
- The code does not integrate with real-time data sources or other business systems (e.g., sales channels, accounting systems), which could limit its effectiveness and responsiveness.

Code:

```
import pandas as pd
import numpy as np
import math
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Data Collection Functions
def fetch_sales_data():
    return pd.read_csv('sales_data.csv')

def fetch_inventory_data():
    return pd.read_csv('inventory_data.csv')

def fetch_supplier_data():
    return pd.read_csv('supplier_data.csv')

def collect_data():
    sales_data = fetch_sales_data()
    inventory_data = fetch_inventory_data()
    supplier_data = fetch_supplier_data()
    return sales_data, inventory_data, supplier_data

# Demand Forecasting Function
def forecast_demand(sales_data):
    forecasted_demand = {}
    for item in sales_data['item'].unique():
        item_data = sales_data[sales_data['item'] == item]
        model = ExponentialSmoothing(item_data['sales'], seasonal='add', seasonal_periods=12)
        fit = model.fit()
        forecast = fit.forecast(12)
        forecasted_demand[item] = forecast.mean()
    return forecasted_demand
```

Reorder Point and Safety Stock Functions

```
def calculate_safety_stock(demand, lead_time, service_level=0.95):
```

```
    z_score = 1.65 # Corresponds to a 95% service level
```

```
    std_deviation = demand * 0.1
```

```
    safety_stock = z_score * std_deviation
```

```
    return safety_stock
```

```
def calculate_inventory_parameters(predicted_demand, lead_time):
```

```
    safety_stock = calculate_safety_stock(predicted_demand, lead_time)
```

```
    reorder_point = (lead_time * predicted_demand) + safety_stock
```

```
    return reorder_point, safety_stock
```

Economic Order Quantity Function

```
def calculate_economic_order_quantity(demand, ordering_cost, holding_cost):
```

```
    EOQ = math.sqrt((2 * demand * ordering_cost) / holding_cost)
```

```
    return EOQ
```

Inventory Optimization Function

```
def place_order(item, quantity):
```

```
    print(f"Placing order for {quantity} units of {item}")
```

```
def identify_slow_moving_items(item):
```

```
    print(f"Identifying strategies for slow-moving item: {item}")
```

```
def optimize_inventory(inventory_data, predicted_demand, supplier_data):
```

```
    for item in inventory_data['item'].unique():
```

```
        lead_time = supplier_data[supplier_data['item'] == item]['lead_time'].values[0]
```

```
        reorder_point, safety_stock = calculate_inventory_parameters(predicted_demand[item],  
lead_time)
```

```
        EOQ = calculate_economic_order_quantity(predicted_demand[item], ordering_cost=10,  
holding_cost=2)
```

```
        current_level = inventory_data[inventory_data['item'] == item]['current_level'].values[0]
```

```
if current_level < reorder_point:
```

```
    place_order(item, EOQ)
```

```
if current_level > safety_stock:
```

```
    identify_slow_moving_items(item)
```

```
# Automated Reordering Function
```

```
def automate_reordering(inventory_data, reorder_points, EOQs):
```

```
    for item in inventory_data['item'].unique():
```

```
        current_level = inventory_data[inventory_data['item'] == item]['current_level'].values[0]
```

```
        if current_level < reorder_points[item]:
```

```
            place_order(item, EOQs[item])
```

```
# Supplier Management Functions
```

```
def evaluate_supplier_performance(supplier):
```

```
    return np.random.uniform(0.5, 1.0)
```

```
def negotiate_terms(supplier):
```

```
    print(f"Negotiating terms with supplier {supplier}")
```

```
def find_alternative_supplier(supplier):
```

```
    print(f"Finding alternative supplier for {supplier}")
```

```
def evaluate_suppliers(supplier_data):
```

```
    for supplier in supplier_data['supplier'].unique():
```

```
        performance_score = evaluate_supplier_performance(supplier)
```

```
        if performance_score < 0.8:
```

```
            negotiate_terms(supplier)
```

```
        if performance_score < 0.6:
```

```
            find_alternative_supplier(supplier)
```

```
# Performance Monitoring Functions
```

```
def calculate_inventory_turnover(inventory_data):
```

```
# Placeholder implementation
return 5.0

def calculate_stockout_rate(inventory_data):
    # Placeholder implementation
    return 0.02

def calculate_GMROI(inventory_data):
    # Placeholder implementation
    return 1.5

def monitor_performance():
    inventory_turnover = calculate_inventory_turnover(None)
    stockout_rate = calculate_stockout_rate(None)
    GMROI = calculate_GMROI(None)

    print(f"Inventory Turnover: {inventory_turnover}")
    print(f"Stockout Rate: {stockout_rate}")
    print(f"GMROI: {GMROI}")

    conduct_regular_audits()

def conduct_regular_audits():
    print("Conducting regular inventory audits")

# Continuous Improvement Functions
def review_inventory_policies():
    print("Reviewing inventory policies")

def incorporate_feedback():
    print("Incorporating feedback from staff")

def explore_advanced_technologies():
```



```

print("Exploring advanced technologies like AI and machine learning")

def continuous_improvement():
    review_inventory_policies()
    incorporate_feedback()
    explore_advanced_technologies()

# Main Execution Flow
def main():
    sales_data, inventory_data, supplier_data = collect_data()
    predicted_demand = forecast_demand(sales_data)

    reorder_points = {}
    EOQs = {}

    for item in inventory_data['item'].unique():
        lead_time = supplier_data[supplier_data['item'] == item]['lead_time'].values[0]
        reorder_point, safety_stock = calculate_inventory_parameters(predicted_demand[item],
lead_time)
        EOQ = calculate_economic_order_quantity(predicted_demand[item], ordering_cost=10,
holding_cost=2)

        reorder_points[item] = reorder_point
        EOQs[item] = EOQ

    optimize_inventory(inventory_data, predicted_demand, supplier_data)
    automate_reordering(inventory_data, reorder_points, EOQs)
    evaluate_suppliers(supplier_data)
    monitor_performance()
    continuous_improvement()

if __name__ == "__main__":
    main()

```

Sample Output / Screen Shots

```
Command Prompt
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 steps. First, integrate a real-time traffic data API to fetch current traffic conditions, including congestion levels, road closures, and accident reports. This data will be processed and stored in a centralized system to ensure timely and accurate updates. Next, implement a routing algorithm, such as Dijkstra's or A* algorithm, to calculate the most efficient routes based on real-time traffic data. This involves creating or leveraging an existing road network graph where each edge weight represents traffic conditions. The system should then provide users with alternative routes to avoid congestion and minimize travel time. Ensure the system is scalable and can handle high traffic volumes, providing quick and reliable updates to support smart city operations. Regularly update the algorithms and data sources to adapt to changing traffic patterns and improve accuracy.

Pseudocode:

FUNCTION fetchTrafficData(location):

 # Fetch real-time traffic data from an external traffic API

 response = API_REQUEST("TRAFFIC_API_URL", location)

 IF response.status == SUCCESS:

 RETURN response.data

 ELSE:

 RETURN ERROR

FUNCTION updateTrafficData():

 # Update internal traffic data store with the latest information

 traffic_data = fetchTrafficData(current_location)

 IF traffic_data != ERROR:

```
    STORE traffic_data in DATABASE
```

```
ELSE:
```

```
    LOG_ERROR("Failed to update traffic data")
```

```
FUNCTION createGraphFromData(traffic_data):
```

```
    # Create a graph representation of the road network from traffic data
```

```
    graph = INITIALIZE_GRAPH()
```

```
    FOR each road_segment IN traffic_data:
```

```
        ADD road_segment TO graph
```

```
    RETURN graph
```

```
FUNCTION findAlternativeRoute(start, end, graph):
```

```
    # Find alternative routes using the graph and real-time traffic data
```

```
    shortest_path = CALCULATE_SHORTEST_PATH(graph, start, end)
```

```
    alternative_routes = FIND_ALTERNATIVE_PATHS(graph, start, end)
```

```
    RETURN shortest_path, alternative_routes
```

```
FUNCTION getRealTimeTrafficUpdates(location):
```

```
    # Retrieve real-time traffic updates for a specific location
```

```
    traffic_data = fetchTrafficData(location)
```

```
    IF traffic_data != ERROR:
```

```
        RETURN traffic_data
```

```
    ELSE:
```

```
        RETURN ERROR
```

```
FUNCTION suggestRoutes(start, end):
```

```
    # Suggest the best route and alternative routes
```

```
    traffic_data = getRealTimeTrafficUpdates(current_location)
```

```
    IF traffic_data != ERROR:
```

```
        graph = createGraphFromData(traffic_data)
```

```
        best_route, alternative_routes = findAlternativeRoute(start, end, graph)
```

```
        RETURN best_route, alternative_routes
```

```
    ELSE:
```

RETURN ERROR

Main Execution Flow

WHILE TRUE:

Continuously update traffic data and provide route suggestions

updateTrafficData()

user_location = GET_USER_LOCATION()

destination = GET_USER_DESTINATION()

best_route, alternative_routes = suggestRoutes(user_location, destination)

IF best_route= ERROR:

DISPLAY("Best Route:", best_route)

DISPLAY("Alternative Routes:", alternative_routes)

ELSE:

DISPLAY("Error fetching traffic data or routes")

WAIT FOR A SHORT PERIOD BEFORE NEXT UPDATE

Detailed explanation of the actual code:

- The **Flask** library is imported to create the web service. **requests** is used for making HTTP requests to external APIs, and **networkx** is used for graph operations.
- An instance of the Flask application is created with **app = Flask(__name__)**.
- API configurations include an API key and a traffic API URL. The URL is a placeholder and needs to be replaced with a real traffic API endpoint.
- A sample graph is defined in **GRAPH_DATA**, representing a simple road network with nodes and weighted edges. This graph is used to demonstrate route finding and should be replaced with real traffic data.
- **G** is initialized as a **networkx** graph using the sample **GRAPH_DATA**.
- The **fetch_traffic_data(location)** function sends an HTTP GET request to the traffic API with the provided location and API key. It handles potential request errors and returns the traffic data in JSON format if the request is successful.
- The **create_graph_from_data(traffic_data)** function converts the fetched traffic data into a **networkx** graph. It assumes that the traffic data format is similar to **GRAPH_DATA**, where nodes are connected by edges with weights representing distances or travel times.

- The `find_alternative_routes(start, end, graph)` function finds the shortest path between two nodes in the graph using `networkx`. It returns the path if found, or `None` if no path exists.
- The `/traffic` API endpoint handles GET requests to fetch real-time traffic updates. It requires a `location` parameter, fetches traffic data using `fetch_traffic_data(location)`, and returns the data in JSON format or an error message if the fetch fails.
- The `/route` API endpoint handles GET requests to get an alternative route between `start` and `end` locations. It requires both parameters, fetches traffic data for the `current_location` (a placeholder for actual logic), creates a graph from this data, and finds the shortest path using `find_alternative_routes(start, end, graph)`. It returns the route if found or an error message if no route is found.
- The application is run in debug mode with `app.run(debug=True)`, which enables detailed error messages and auto-reloading during development.

Assumptions made (if any):

- The traffic data API provides up-to-date information on traffic conditions and is accessible via HTTP requests.
- An API key is required for accessing the traffic data, and both the API URL and key are valid.
- The traffic data API returns data in a format that can be converted into a graph structure, such as JSON with node and edge details.
- The road network can be accurately represented as a weighted graph, where nodes are locations and edges are distances or travel times.
- The graph used for route finding can be updated with the latest traffic data to reflect current conditions.

Limitations:

- The accuracy of traffic updates depends on the quality and frequency of data provided by the traffic API.
- The system may experience latency due to delays in fetching data from the traffic API or processing large volumes of data.
- Real-time traffic data might not cover all areas comprehensively, leading to incomplete or outdated information for certain locations.
- The sample road network graph used in the example may not represent the full complexity of the actual road network in the city.
- Handling dynamic and rapidly changing traffic conditions may be challenging, especially if the graph is not updated frequently.

Code:

```
from flask import Flask, jsonify, request
import requests
import networkx as nx

app = Flask(__name__)

# Configuration
API_KEY = 'your_api_key_here'
TRAFFIC_API_URL = 'https://maps.googleapis.com/maps/api/traffic/json' # Example URL
GRAPH_DATA = {
    # Define a simple graph for the example
    # Format: 'node': {'neighbor1': distance, 'neighbor2': distance, ...}
    'A': {'B': 5, 'C': 10},
    'B': {'A': 5, 'C': 3, 'D': 9},
    'C': {'A': 10, 'B': 3, 'D': 7},
    'D': {'B': 9, 'C': 7}
}
G = nx.Graph(GRAPH_DATA)

# Route optimization function
def find_alternative_route(start, end):
    try:
        path = nx.shortest_path(G, source=start, target=end, weight='weight')
        return path
    except nx.NetworkXNoPath:
        return None

# API Endpoint to get traffic updates
@app.route('/traffic', methods=['GET'])
def get_traffic_updates():
    location = request.args.get('location')
```

```

if not location:
    return jsonify({'error': 'Location parameter is required'}), 400

# Example request to traffic API
response = requests.get(f'{TRAFFIC_API_URL}? key= {API_KEY}&location={location}')
if response.status_code == 200:
    data = response.json()
    # Process and return traffic data (simplified)
    return jsonify(data)
else:
    return jsonify({'error': 'Failed to fetch traffic data'}), response.status_code

# API Endpoint to get alternative route
@app.route('/route', methods=['GET'])
def get_route():
    start = request.args.get('start')
    end = request.args.get('end')
    if not start or not end:
        return jsonify({'error': 'Start and end parameters are required'}), 400

    path = find_alternative_route(start, end)
    if path:
        return jsonify({'route': path})
    else:
        return jsonify({'error': 'No route found'}), 404

if __name__ == '__main__':
    app.run(debug=True)

```

Sample Output / Screen Shots


```
Command Prompt
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 selecting a reliable COVID-19 data API that provides comprehensive and up-to-date information on cases, recoveries, and deaths. Secure the necessary API key for authentication and configure your application using Flask, a web framework suitable for handling HTTP requests and responses. Design a RESTful API endpoint that allows users to request COVID-19 statistics for a specific region or country. Implement a function to fetch data from the chosen API, ensuring it includes the API key and handles potential errors gracefully. Process the JSON response to extract and compute the relevant statistics, including total cases, recoveries, and deaths. Incorporate error handling to manage scenarios where the data fetch fails or the response is incomplete, and return the statistics in a well-structured JSON format. Finally, deploy the application to a secure and scalable hosting environment to make it accessible to users in need of real-time COVID-19 updates.

Pseudocode:

1. Initialize application

- Import necessary libraries (Flask, requests)

2. Define constants

- API_KEY = 'your_api_key_here'
- COVID_API_URL = 'https://api.example.com/covid/{region}'

3. Define function to fetch COVID-19 data

FUNCTION fetch_covid_data(region):

- Construct request URL using COVID_API_URL and region
- Set up headers with API_KEY
- Make GET request to the API
- IF request is successful:

- Parse and return JSON response
- ELSE:
 - Handle errors (e.g., log error, return None)

4. Define API endpoint for COVID-19 statistics

ROUTE '/covid' METHOD GET:

- Extract 'region' parameter from request
- IF 'region' is missing:
 - Return error message with status code 400
- Call `fetch_covid_data(region)` function
- IF data is successfully fetched:
 - Extract total cases, recoveries, and deaths from the data
 - Return these statistics in JSON format
- ELSE:
 - Return error message with status code 500

5. Start the application

- Run Flask application with debug mode enabled

Detailed explanation of the actual code:

- The Flask web framework is used to create the web application, with `Flask` imported to handle routing and request management.
- The `requests` library is included to make HTTP requests to the COVID-19 data API.
- An instance of the Flask application is created using `app = Flask(__name__)`.
- The `API_KEY` is a placeholder for authentication with the COVID-19 data API. This should be replaced with a valid API key.
- The `COVID_API_URL` is a placeholder URL for the COVID-19 data API. `{country}` is a placeholder that will be replaced with the actual country name when making requests.
- The `fetch_covid_data` function takes a `country` parameter and constructs the API request URL by formatting the `COVID_API_URL` with the provided country name.
- Inside `fetch_covid_data`, an HTTP GET request is made to the API using `requests.get()`. The request includes an `Authorization` header with the API key.
- `response.raise_for_status()` is used to check if the request was successful. If the request fails (e.g., due to network issues or an invalid response), an exception is raised.

- If the request is successful, `response.json()` parses the JSON response from the API and returns it.
- If an exception occurs, it is caught and logged, and the function returns `None` to indicate a failure to fetch data.
- The `/covid` endpoint is defined to handle GET requests. It extracts the `country` parameter from the query string using `request.args.get('country')`.
- If the `country` parameter is missing, the endpoint responds with a 400 Bad Request error and a message indicating that the parameter is required.
- If the `country` parameter is provided, the `fetch_covid_data` function is called to retrieve COVID-19 data for the specified country.
- If data is successfully fetched:
 - The total number of cases is calculated by summing the `Cases` values from the data.
 - The total number of recoveries is calculated by summing the `Recovered` values (defaulting to 0 if the field is not present).
 - The total number of deaths is calculated by summing the `Deaths` values (defaulting to 0 if the field is not present).
- The results are returned in JSON format, including the country name and the computed totals.
- If data fetching fails, a 500 Internal Server Error response is returned with a message indicating the failure.
- The application is set to run in debug mode with `app.run(debug=True)`, which enables detailed error messages and automatic reloading of the application during development.
- The code assumes the API response format includes fields for `Cases`, `Recovered`, and `Deaths`. If the actual API response differs, the code may need adjustments to correctly process and display the data.

Assumptions made (if any):

- The specified region or country parameter used in requests is valid and recognized by the API.
- The application is deployed in an environment where Flask can run and handle HTTP requests.
- The application handles network-related issues gracefully, including timeouts and connection errors.
- Users of the application provide valid and correctly formatted region names or codes for querying COVID-19 statistics.
- The API endpoint being used is reliable and remains stable without significant changes to its structure or URL.

Limitations:

- The application depends on the external COVID-19 data API, which may be subject to outages, rate limits, or changes in data format that could affect functionality.

- There may be a delay between real-time events and the data reflected in the application due to processing and reporting lag.
- API rate limits could restrict the number of requests that can be made in a given timeframe, potentially affecting the application's ability to provide frequent updates.
- Not all regions or countries may be covered by the API, which could limit the application's effectiveness in providing data for certain areas.
- If the API updates its response structure or data format, modifications to the application may be required to handle these changes.

Code:

```
from flask import Flask, jsonify, request
import requests

app = Flask(__name__)

# Configuration
API_KEY = 'your_api_key_here'
COVID_API_URL = 'https://api.covid19api.com/dayone/country/{country}/status/confirmed/live' #
Replace with the actual COVID-19 API URL

# Function to fetch COVID-19 data
def fetch_covid_data(country):
    try:
        response = requests.get(COVID_API_URL.format(country=country), headers={'Authorization':
f'Bearer {API_KEY}'})
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        print(error fetching COVID-19 data: {e})
        return None

# API Endpoint to get COVID-19 statistics
@app.route('/covid', methods=['GET'])
def get_covid_statistics():
    country = request.args.get('country')
    if not country:
```

```
return jsonify({'error': 'Country parameter is required'}), 400
```

```
covid_data = fetch_covid_data(country)
```

```
if covid_data:
```

```
    cases = sum(item['Cases'] for item in covid_data)
```

```
    recoveries = sum(item.get('Recovered', 0) for item in covid_data)
```

```
    deaths = sum(item.get('Deaths', 0) for item in covid_data)
```

```
    return jsonify({
```

```
        'country': country,
```

```
        'total_cases': cases,
```

```
        'total_recoveries': recoveries,
```

```
        'total_deaths': deaths
```

```
    })
```

```
else:
```

```
    return jsonify({'error': 'Failed to fetch COVID-19 data'}), 500
```

```
if __name__ == '__main__':
```

```
    app.run(debug=True)
```

Sample Output / Screen Shots

```
Command Prompt
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>
```

