**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:**

**Importing the requests Library**

The first line of the code imports the **requests** library, which is a popular Python library used for making HTTP requests. The **requests** library allows you to send HTTP requests and returns the server's response.

**Defining the get\_weather Function**

The next line defines a function called **get\_weather** that takes two parameters: **latitude** and **longitude**. This function is used to retrieve the weather data for a specific location.

**Constructing the API URL**

Inside the **get\_weather** function, a URL is constructed using an f-string. The URL is for the Open-Meteo API, which provides weather data. The URL includes the following parameters:

* **latitude** and **longitude**: These are the coordinates of the location for which we want to retrieve the weather data.
* **current\_weather=true**: This parameter tells the API to include the current weather conditions in the response.
* **hourly=temperature\_2m,relativehumidity\_2m,precipitation,windspeed\_10m**: This parameter specifies the hourly data that we want to retrieve. In this case, we are retrieving the temperature, relative humidity, precipitation, and wind speed at 2 meters above the ground.
* **daily=temperature\_2m\_max,temperature\_2m\_min,precipitation\_sum**: This parameter specifies the daily data that we want to retrieve. In this case, we are retrieving the maximum and minimum temperature, and the total precipitation.

**Sending the GET Request**

The next line sends a GET request to the constructed URL using the **requests.get()** method. The response from the API is stored in the **response** variable.

**Parsing the JSON Response**

The next line parses the JSON response from the API using the **response.json()** method. The parsed JSON data is stored in the **data** variable.

**Returning the Weather Data**

The final line of the **get\_weather** function returns the **data** variable, which contains the weather data for the specified location.

**Example Usage**

The next few lines demonstrate how to use the **get\_weather** function. We set the **latitude** and **longitude** variables to the coordinates of a location (in this case, Bangalore, India). We then call the **get\_weather** function with these coordinates and store the result in the **weather\_data** variable.

**Printing the Weather Data**

The final few lines print the weather data in a formatted string. The string includes the following information:

* **Latitude** and **Longitude**: The coordinates of the location.
* **Elevation**: The elevation of the location above sea level.
* **Current Temperature**: The current temperature at the location, along with its unit (e.g. Celsius or Fahrenheit).
* **Current Wind Speed**: The current wind speed at the location, along with its unit (e.g. meters per second or miles per hour).
* **Current Wind Direction**: The current wind direction at the location, in degrees.

**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

def get\_weather(latitude, longitude):

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

response = requests.get(url)

data = response.json()

return data

# Example usage:

latitude = 12.9716

longitude = 77.6412

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

print(f"""

Latitude: {latitude}

Longitude: {longitude}

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

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

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

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

""")

**Sample Output / Screen Shots:**

Latitude: 13.03321260891547

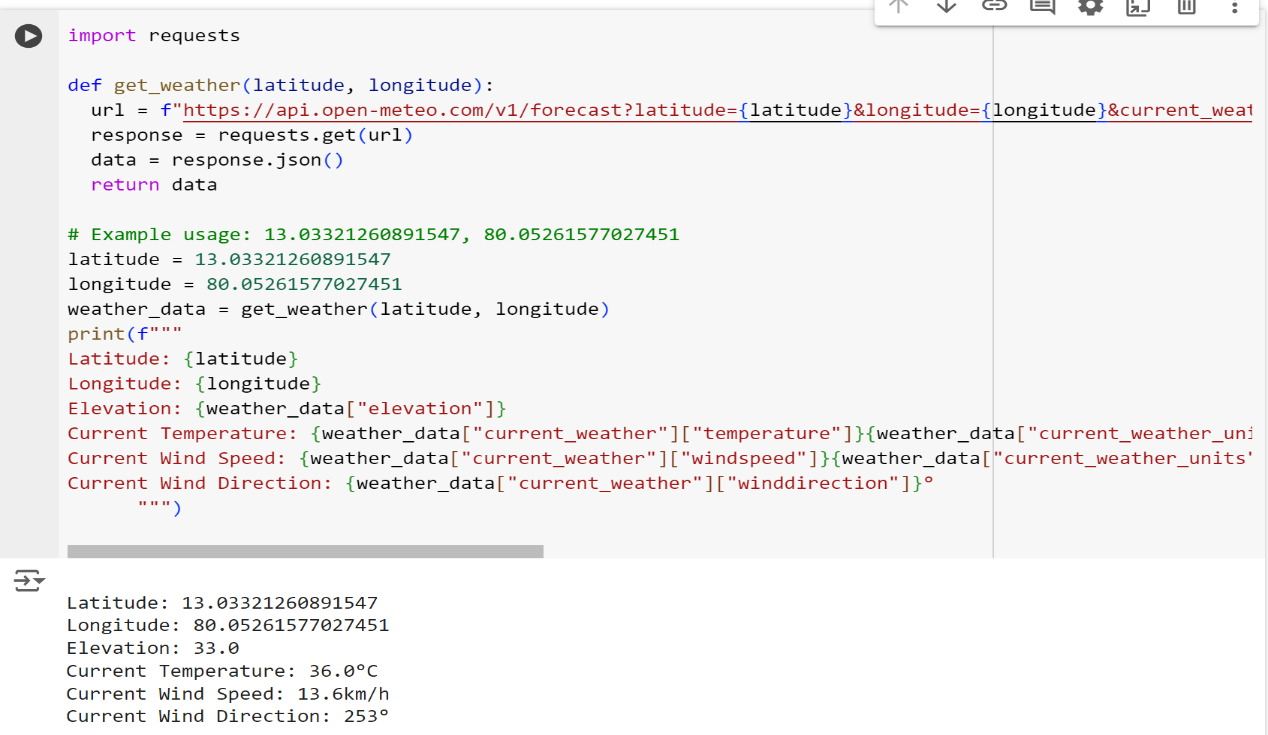
Longitude: 80.05261577027451

Elevation: 33.0

Current Temperature: 36.0°C

Current Wind Speed: 13.6km/h

Current Wind Direction: 253°



**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 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:**

**InventoryItem Class**

The **InventoryItem** class represents a single item in the inventory. It has four attributes:

* **product\_id**: a unique identifier for the item
* **name**: the name of the item
* **price**: the price of the item
* **quantity**: the quantity of the item in stock

The **\_\_init\_\_** method is a special method in Python that is called when an object is created. It initializes the attributes of the object with the values passed to it.

**InventorySystem Class**

The **InventorySystem** class represents the entire inventory system. It has five methods:

* **\_\_init\_\_**: initializes the inventory system with an empty dictionary **inventory**
* **add\_item**: adds an item to the inventory
* **update\_item**: updates an item's quantity and optionally price
* **delete\_item**: deletes an item from the inventory
* **get\_item**: retrieves an item from the inventory
* **get\_all\_items**: returns a list of all inventory items

**\_\_init\_\_ Method**

The **\_\_init\_\_** method initializes the inventory system with an empty dictionary **inventory**. This dictionary will store all the items in the inventory, with the product ID as the key and the **InventoryItem** object as the value.

**add\_item Method**

The **add\_item** method adds an item to the inventory. It takes an **InventoryItem** object as an argument and adds it to the **inventory** dictionary with the product ID as the key.

**update\_item Method**

The **update\_item** method updates an item's quantity and optionally price. It takes three arguments:

* **product\_id**: the ID of the item to update
* **new\_quantity**: the new quantity of the item
* **new\_price**: the new price of the item (optional)

If the item is found in the inventory, it updates the quantity and price (if provided) of the item. If the item is not found, it prints an error message.

**delete\_item Method**

The **delete\_item** method deletes an item from the inventory. It takes a **product\_id** as an argument and removes the item from the **inventory** dictionary if it exists. If the item is not found, it prints an error message.

**get\_item Method**

The **get\_item** method retrieves an item from the inventory. It takes a **product\_id** as an argument and returns the corresponding **InventoryItem** object if it exists. If the item is not found, it returns **None**.

**get\_all\_items Method**

The **get\_all\_items** method returns a list of all inventory items. It simply returns a list of all the values in the **inventory** dictionary.

**Creating an Inventory System and Adding Items**

The code creates an instance of the **InventorySystem** class and adds two items to the inventory using the **add\_item** method.

**Updating an Item**

The code updates the quantity and price of the item with product ID 1 using the **update\_item** method.

**Deleting an Item**

The code deletes the item with product ID 2 using the **delete\_item** method.

**Retrieving an Item**

The code retrieves the item with product ID 1 using the **get\_item** method and prints its name, price, and quantity.

**Retrieving All Items**

The code retrieves all items in the inventory using the **get\_all\_items** method and prints their product IDs and names.

Overall, this code provides a basic implementation of an inventory system with CRUD (Create, Read, Update, Delete) operations.

**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

class InventoryItem:

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

self.product\_id = product\_id

self.name = name

self.price = price

self.quantity = quantity

class InventorySystem:

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

self.inventory = {}

def add\_item(self, item):

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

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

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

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

if product\_id in self.inventory:

item = self.inventory[product\_id]

item.quantity = new\_quantity

if new\_price:

item.price = new\_price

else:

print("Item not found")

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

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

if product\_id in self.inventory:

del self.inventory[product\_id]

else:

print("Item not found")

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

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

if product\_id in self.inventory:

return self.inventory[product\_id]

else:

return None

def get\_all\_items(self):

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

return list(self.inventory.values())

inventory\_system = InventorySystem()

# Add items

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

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

inventory\_system.add\_item(item1)

inventory\_system.add\_item(item2)

# Update item

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

# Delete item

inventory\_system.delete\_item(2)

# Get item

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

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

# Get all items

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

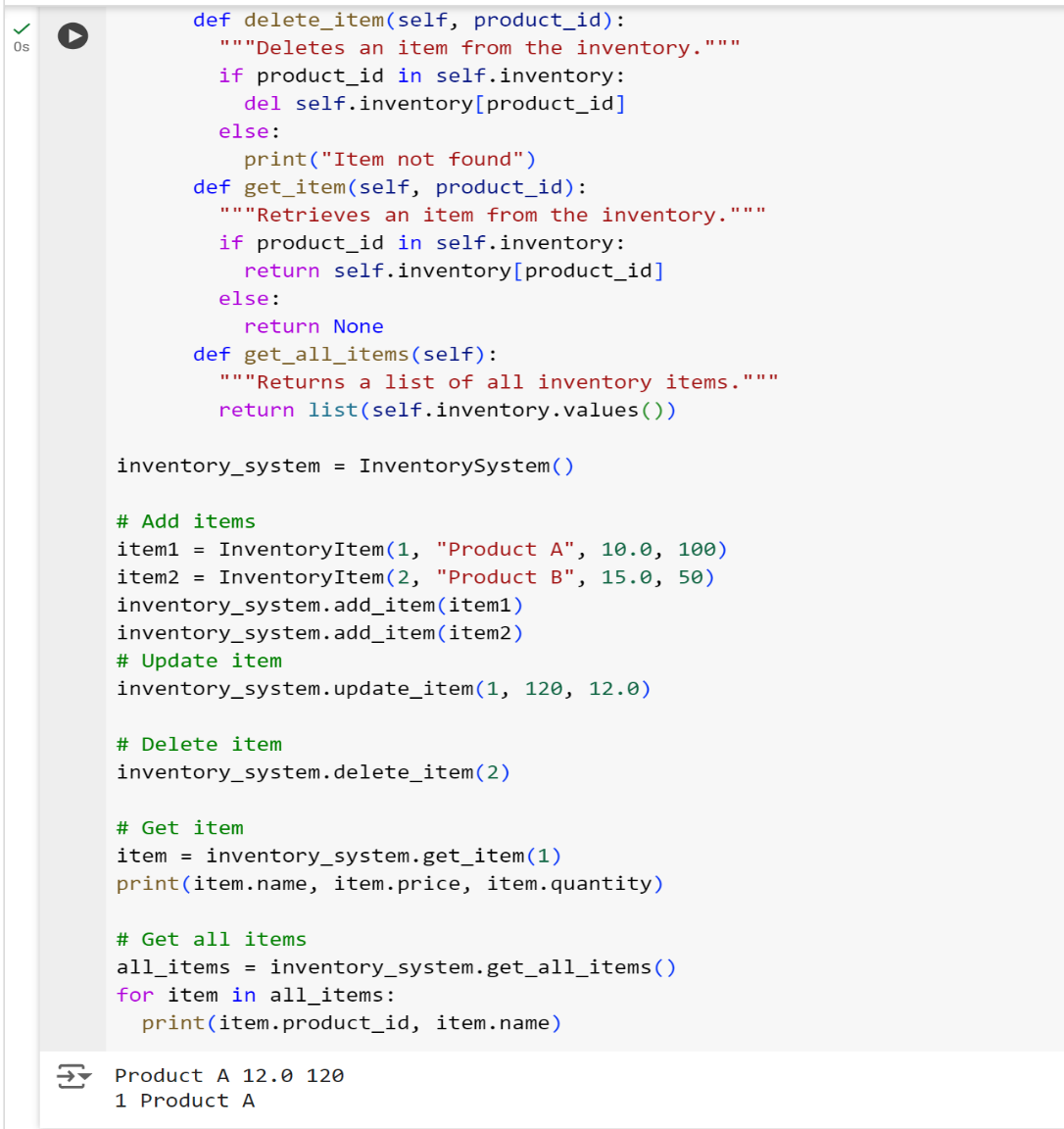
for item in all\_items:

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

**Sample Output / Screen Shots:**

Product A 12.0 120

1 Product



**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:**

**Importing Libraries**

The script starts by importing the necessary libraries:

* **requests**: for making HTTP requests to the Google Maps API
* **json**: for parsing the JSON response from the API
* **time**: for implementing a delay between API requests
* **datetime** and **timedelta**: for calculating the departure time for the API request

**get\_traffic\_data Function**

This function takes two arguments: **origin** and **destination**, which are the starting and ending points for the traffic data request. It does the following:

1. Sets the API key (replace **GAPI-KEY** with your actual API key)
2. Calculates the departure time by adding 5 minutes to the current time using **datetime** and **timedelta**
3. Constructs the API request URL using the **origin**, **destination**, **departure\_time**, and **api\_key**
4. Sends a GET request to the API using **requests**
5. Parses the JSON response using **json.loads**
6. Prints the raw API response (for debugging purposes)
7. Returns the parsed JSON data

**process\_traffic\_data Function**

This function takes the parsed JSON data from **get\_traffic\_data** and extracts the following information:

1. Distance between the origin and destination
2. Duration of the trip
3. Traffic status (e.g., "OK", "NOT\_FOUND", etc.)

It returns these three values as a tuple.

**main Function**

This is the entry point of the script. It does the following:

1. Sets the origin and destination locations (in this case, Bangalore and Mysore)
2. Enters an infinite loop that runs every minute (using **time.sleep(60)**)
3. Calls **get\_traffic\_data** to fetch the traffic data
4. Calls **process\_traffic\_data** to extract the distance, duration, and traffic status
5. Prints the extracted information
6. Catches the **KeyboardInterrupt** exception to exit the script cleanly when the user presses Ctrl+C

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

This block ensures that the **main** function is only executed when the script is run directly (i.e., not when it's imported as a module by another script).

**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:**

import requests

import json

import time

from datetime import datetime, timedelta

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

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

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

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

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

response = requests.get(url)

data = json.loads(response.text)

print(data)

return data

def process\_traffic\_data(data):

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

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

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

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

return distance, duration, traffic\_status

def main():

origin = "Bangalore"

destination = "Mysore"

try:

while True:

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

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

print(f"Distance: {distance}")

print(f"Duration: {duration}")

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

time.sleep(60)

except KeyboardInterrupt:

print("Exiting...")

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

main()

**Sample Output :**

{'destination\_addresses': [Vijayawada, Andhra Pradesh, India'], 'origin\_addresses': [Nellore, Andhra Pradesh, India'], 'rows': [{'elements': [{'distance': {'text': '281 km', 'value': 143422}, 'duration': {'text': '4 hours 31 mins', 'value': 9205}, 'duration\_in\_traffic': {'text': '2 hours 27 mins', 'value': 8833}, 'status': 'OK'}]}], 'status': 'OK'}

Distance: 281 km

Duration: 4 hours 31 mins

Traffic Status: OK

**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:**

**Importing Libraries**

The script starts by importing the **requests** library, which is used to send an HTTP request to the COVID-19 API.

**API Request**

The script then defines the URL of the API endpoint, which is **https://disease.sh/v3/covid-19/countries/india**. This API endpoint returns COVID-19 data for India in JSON format.

The script sends a GET request to the API using the **requests.get()** method, which returns a response object.

**Parsing the Response**

The script then parses the response using the **response.json()** method, which returns a Python dictionary containing the JSON data.

**Accessing the Data**

The script accesses the following data from the dictionary:

* **cases**: The total number of COVID-19 cases in India
* **deaths**: The total number of COVID-19 deaths in India
* **recovered**: The total number of COVID-19 recoveries in India

**Printing the Data**

The script then prints the following COVID-19 statistics for India:

* Cases today
* Deaths today
* Recovered today
* Active cases
* Critical cases
* Cases per million
* Deaths per million
* Tests done
* Tests per million
* Total cases
* Total deaths
* Total recovered

The script uses f-strings to format the output, which makes it easier to read and understand.

**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:**

import requests

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

response = requests.get(url)

data = response.json()

# Accessing the data

cases = data["cases"]

deaths = data["deaths"]

recovered = data["recovered"]

# Printing the data

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Sample Output / Screen Shots :**

COVID-19 Statistics for India:

-------------------------------

Cases today: 0

Deaths today: 0

Recovered today: 0

Active cases: 44501823

Critical cases: 0

Cases per million: 32016

Deaths per million: 379

Tests done: 935879495

Tests per million: 665334

Total cases: 45035393

Total deaths: 533570

Total recovered: 0

****