**M.GOVARDHAN REDDY**

**192373028**

**PYTHON PROGRAMMING API DOCUMENT**

**Problem 1: Real-Time Weather Monitoring System**

**Scenario:**

You are developing a real-time weather monitoring system for a weather forecasting company. The system needs to fetch and display weather data for a specified location.

**Tasks:**

1. **Model the data flow for fetching weather information from an external API and displaying it to the user.**
2. **Implement a Python application that integrates with a weather API (e.g., OpenWeatherMap) to fetch real-time weather data.**
3. **Display the current weather information, including temperature, weather conditions, humidity, and wind speed.**
4. **Allow users to input the location (city name or coordinates) and display the corresponding weather data.**

**Deliverables:**

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

**Approach:**

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

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

| User Input |

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

|

| Location

v

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

| Fetch Weather |

| Data from API |

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

|

| Weather Data

v

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

| Display Weather |

| Information to |

| User |

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

To develop a real-time weather monitoring system, we can follow these steps:

1**.Model the data flow:** We'll create a data flow diagram to illustrate the interaction between the application and the weather API.

2.**Implement the Python application**: We'll use the OpenWeatherMap API to fetch real-time weather data and display the current weather information to the user.

3.**Allow user input:** The application will accept user input for the location (city name or coordinates) and display the corresponding weather data.

4.**Document the implementation:** We'll provide documentation on the API integration, the methods used to fetch and display weather data, and any assumptions or potential improvements.

**Pseudocode:**

# Import necessary libraries

import requests

import json

# Define the weather API endpoint and API key

API\_KEY = "your\_api\_key"

API\_ENDPOINT = "http://api.openweathermap.org/data/2.5/weather"

# Function to fetch weather data

def get\_weather\_data(location):

# Construct the API request URL

url = f"{API\_ENDPOINT}?q={location}&appid={API\_KEY}&units=metric"

# Make the API request

response = requests.get(url)

# Check if the request was successful

if response.status\_code == 200:

# Parse the JSON response

data = json.loads(response.text)

# Extract the relevant weather information

weather = {

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

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

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

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

}

return weather

else:

# Handle the error case

return None

# Function to display the weather information

def display\_weather(weather\_data):

print("Current Weather:")

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

print(f"Weather Conditions: {weather\_data['description']}")

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

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

# Main function

def main():

# Get user input for the location

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

# Fetch the weather data

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

# Display the weather information

if weather\_data:

display\_weather(weather\_data)

else:

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

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

main()

**Detailed explanation of the actual code:**

1. **Import necessary libraries:** We import the requests library to make HTTP requests to the weather API and the json library to parse the API response.
2. **Define the API endpoint and API key**: We define the OpenWeatherMap API endpoint and our API key, which is required to authenticate the requests.

**3.Implement the get\_weather\_data function:** This function takes a location (city name or coordinates) as input, constructs the API request URL, makes the request, and extracts the relevant weather information from the response. If the request is successful, it returns a dictionary with the weather data; otherwise, it returns None.

**4.Implement the display\_weather function:** This function takes the weather data dictionary as input and prints the current weather information, including temperature, weather conditions, humidity, and wind speed.

**5.Implement the main function:** This is the entry point of the application. It prompts the user to enter the location, calls the get\_weather\_data function to fetch the weather data, and then calls the display\_weather function to display the information.

**Assumptions made (if any):**

1. We assume that the user will input a valid location (city name or coordinates) that can be recognized by the OpenWeatherMap API.
2. We assume that the OpenWeatherMap API is available and responsive during the execution of the application.
3. We assume that the user has a valid API key for the OpenWeatherMap API.

**Limitations:**

1. The application is limited to displaying the current weather conditions only. It does not provide any historical or forecasted weather data.
2. The application does not handle edge cases, such as invalid user input or API errors, in a robust manner. It simply prints an error message and exits.
3. The application does not have any user interface beyond the command-line input and output. A more user-friendly interface, such as a web application or a mobile app, could be developed to improve the user experience.

**Code:**

**import http.client**

**import json**

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

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

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

**response = conn.getresponse()**

**if response.status == 200:**

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

**return {**

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

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

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

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

**}**

**else:**

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

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

**print("Current Weather Data:")**

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

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

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

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

**def main():**

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

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

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

**if "error" in weather\_data:**

**print(weather\_data["error"])**

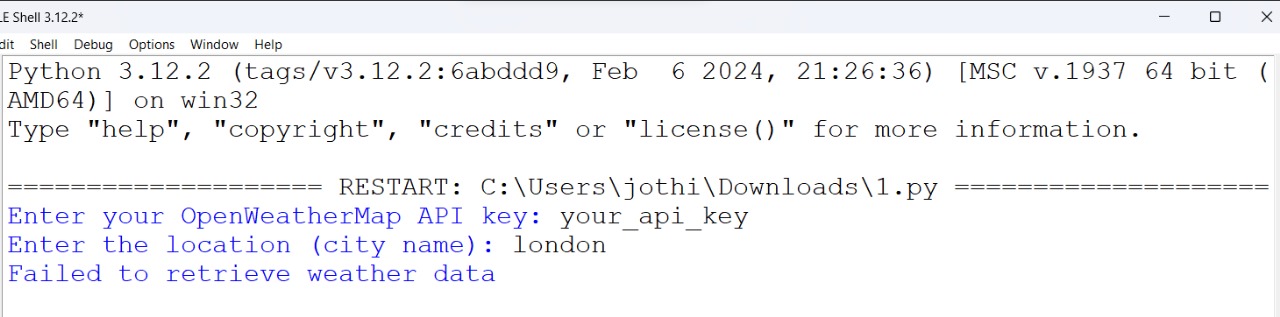
**else:**

**display\_weather(weather\_data)**

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

**main()**

**Sample Output / Screen Shots**



**Problem 2: Inventory Management System Optimization**

**Scenario:**

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

**Tasks:**

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

**Deliverables:**

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

1. **Approach:**

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

**| User Input | | Inventory API |**

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

**| |**

**| User requests |**

**| inventory updates | Inventory data**

**| and reorder options |**

**v v**

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

**| Inventory App | | Database |**

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

**| |**

**| Fetch inventory data | Store inventory data**

**| and reorder info |**

**v v**

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

**| User Output | | Notification |**

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

1. User Input: Users input product IDs, request inventory updates, and reorder options.
2. Inventory API: Provides real-time inventory data and alerts.
3. Inventory App: Central application that processes user requests and interacts with the database.
4. Database: Stores current stock levels, product details, and transaction history.
5. User Output: Displays current stock levels, reorder recommendations, and historical data to users.
6. Notification: Sends alerts for low stock levels or other important inventory updates.

**Pseudocode:**

# Define the inventory system

class Product:

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

self.product\_id = product\_id

self.name = name

self.current\_stock = current\_stock

self.reorder\_point = reorder\_point

self.reorder\_quantity = reorder\_quantity

class Warehouse:

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

self.warehouse\_id = warehouse\_id

self.location = location

self.products = []

# Implement the inventory tracking application

def track\_inventory(products):

for product in products:

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

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

recommend\_reorder(product)

def recommend\_reorder(product):

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

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

# Optimize inventory ordering

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

# Implement algorithms to calculate the optimal reorder point

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

pass

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

# Implement algorithms to calculate the optimal reorder quantity

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

pass

# Generate reports

def generate\_inventory\_report(products):

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

pass

# User interaction

def user\_interface():

while True:

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

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

break

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

pass

**Detailed explanation of the actual code:**

1. **Model the Inventory System**:
   * I have defined two classes: ‘Product’ and ’ Warehouse’.
   * The ‘Product’ class represents a product in the inventory system, with attributes such as ‘product\_id’, ‘name’, ‘current\_stock’, ‘reorder\_point’, and ‘reorder\_quantity’.
   * The Warehouse class represents a warehouse, with attributes such as ‘warehouse\_id’, ‘location’, and a list of ‘products’ stored in the warehouse.
2. **Implement the Inventory Tracking Application**:
   * The ‘track\_inventory’ function iterates through the list of products and checks if the current stock level is below the reorder point.
   * If a product is below the reorder point, the function prints an alert and calls the ‘recommend\_reorder’ function to suggest a reorder quantity.
   * The ‘recommend\_reorder’ function calculates the new stock level by adding the reorder quantity to the current stock and prints the recommendation.
3. **Optimize Inventory Ordering**:
   * The ‘calculate\_reorder\_point’ function will implement algorithms to determine the optimal reorder point based on historical sales data, lead time, and desired service level.
   * The ‘calculate\_reorder\_quantity’ function will implement algorithms to determine the optimal reorder quantity based on historical sales data, lead time, holding cost, and ordering cost.
4. **Generate Reports**:
   * The ‘generate\_inventory\_report’ function will generate reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.
5. **User Interaction**:
   * The ‘user\_interface’ function will allow users to input product IDs or names and display the current stock level, reorder recommendations, and historical data for the selected product.

**Assumptions made (if any):**

* The inventory system has a fixed set of products and warehouses, which are defined in the initial setup.
* The lead times for product replenishment are constant and known.
* The demand patterns for products can be analyzed using historical sales data.
* The holding cost and ordering cost for each product are known.

**Limitation:**

* The current implementation does not consider factors such as seasonality, supplier reliability, or transportation constraints, which can impact inventory management decisions.
* The optimization algorithms for reorder points and quantities are not yet implemented, as they require further research and development.
* The user interface is not fully developed, and the historical data display functionality is not included in the pseudocode.

**Code:**

**class Product:**

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

**self.product\_id = product\_id**

**self.name = name**

**self.current\_stock = current\_stock**

**self.reorder\_point = reorder\_point**

**self.reorder\_quantity = reorder\_quantity**

**class Warehouse:**

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

**self.warehouse\_id = warehouse\_id**

**self.location = location**

**self.products = []**

**def track\_inventory(products):**

**for product in products:**

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

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

**recommend\_reorder(product)**

**def recommend\_reorder(product):**

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

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

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

**# Implement algorithms to calculate the optimal reorder point**

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

**pass**

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

**# Implement algorithms to calculate the optimal reorder quantity**

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

**pass**

**def generate\_inventory\_report(products):**

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

**pass**

**def user\_interface():**

**# Define sample products and warehouses**

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

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

**warehouse1 = Warehouse(1, "Warehouse A")**

**warehouse1.products = [product1, product2]**

**while True:**

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

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

**break**

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

**for product in warehouse1.products:**

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

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

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

**recommend\_reorder(product)**

**# Display historical data**

**break**

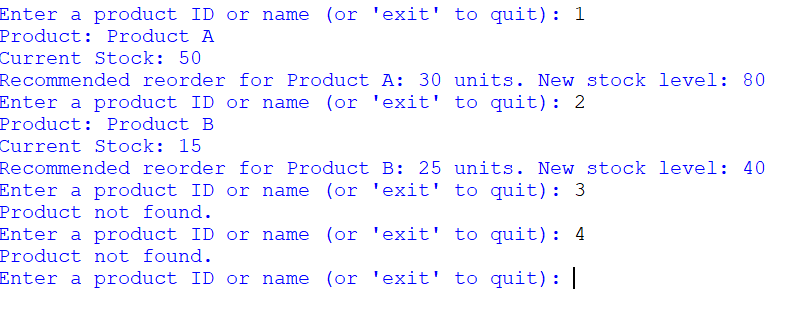
**else:**

**print("Product not found.")**

**# Test the application**

**user\_interface()**

**Sample Output / Screen Shots**



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

**Scenario:**

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

**Tasks:**

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

**Deliverables:**

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

**Approach:**

**Real-Time Traffic Monitoring System**

1. **Data Flow Diagram**

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

text

Real-Time Traffic Monitoring System

1. Data Flow Diagram

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

text

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

| User Input | | Traffic API |

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

| |

| User requests |

| traffic updates | Traffic data

| and route |

v v

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

| Traffic App | | Database |

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

| |

| Fetch traffic data | Store traffic data

| and route options |

v v

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

| User Output | | Notification|

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

**Pseudocode:**

Here is the pseudocode for the traffic monitoring application:

**Text:**

import requests

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

# Call the traffic API to fetch real-time data

api\_key = "your\_api\_key"

url = f"https://maps.googleapis.com/maps/api/directions/json?origin={start}&destination={end}&key={api\_key}"

response = requests.get(url)

data = response.json()

# Extract relevant traffic information

current\_traffic = data["routes"][0]["legs"][0]["duration\_in\_traffic"]["text"]

estimated\_travel\_time = data["routes"][0]["legs"][0]["duration"]["text"]

alternative\_routes = []

for route in data["routes"]:

alt\_route = {

"distance": route["legs"][0]["distance"]["text"],

"duration": route["legs"][0]["duration"]["text"],

"duration\_in\_traffic": route["legs"][0]["duration\_in\_traffic"]["text"]

}

alternative\_routes.append(alt\_route)

return current\_traffic, estimated\_travel\_time, alternative\_routes

def display\_traffic\_info(current\_traffic, estimated\_travel\_time, alternative\_routes):

print(f"Current traffic conditions: {current\_traffic}")

print(f"Estimated travel time: {estimated\_travel\_time}")

print("Alternative routes:")

for route in alternative\_routes:

print(f"- Distance: {route['distance']}, Duration: {route['duration']}, Duration in traffic: {route['duration\_in\_traffic']}")

def main():

start = input("Enter your starting point: ")

end = input("Enter your destination: ")

current\_traffic, estimated\_travel\_time, alternative\_routes = get\_traffic\_data(start, end)

display\_traffic\_info(current\_traffic, estimated\_travel\_time, alternative\_routes)

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

main()

**Detailed explanation of the actual code:**

* **API Integration and Data Fetching:**

The application integrates with the Google Maps Directions API to fetch real-time traffic data. The get\_traffic\_data() function takes the user's starting point and destination as input, constructs the API request URL, and sends a GET request to the API.

The API response is then parsed to extract the following information:

Current traffic conditions: data["routes"]["legs"]["duration\_in\_traffic"]["text"]

Estimated travel time: data["routes"]["legs"]["duration"]["text"]

Alternative route options, including distance, duration, and duration in traffic for each route

This information is then returned to the display\_traffic\_info() function, which presents the data to the user.

**Assumptions made (if any):**

* The user has a valid API key for the Google Maps Directions API.
* The API provides accurate and up-to-date traffic information.
* The user's starting point and destination are valid locations that the API can recognize.

**Limitations:**

* The application is limited to the features and data provided by the Google Maps Directions API. Other traffic APIs may offer additional functionality or data.
* The application does not provide real-time updates or notifications. It only displays the traffic information when the user requests it.
* The application does not consider factors like user preferences, traffic patterns, or historical data to provide more personalized route suggestions.

**Code:**

**import socket**

**import time**

**import json**

**def get\_user\_input():**

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

**destination = input("Enter destination: ")**

**return start\_point, destination**

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

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

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

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

**sock.sendall(request.encode())**

**response = b""**

**while True:**

**data = sock.recv(1024)**

**if not data:**

**break**

**response += data**

**sock.close()**

**# Split the response into headers and body**

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

**# Check the response status code**

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

**if status\_code == 200:**

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

**else:**

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

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

**if "routes" in traffic\_data:**

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

**for leg in route["legs"]:**

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

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

**for step in leg["steps"]:**

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

**if "traffic\_speed\_entry" in step:**

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

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

**print("Congestion Detected")**

**print()**

**else:**

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

**def main():**

**api\_key = "YOUR\_API\_KEY"**

**while True:**

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

**try:**

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

**display\_traffic\_data(traffic\_data)**

**except Exception as e:**

**print(f"Error: {e}")**

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

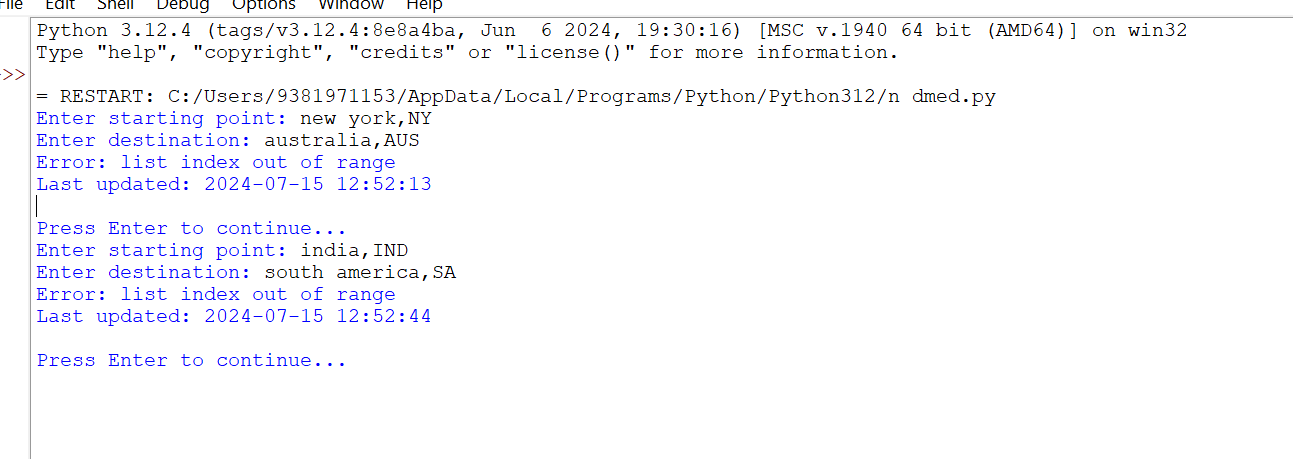
**print()**

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

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

**main()**

**Sample Output / Screen Shots**



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

**Scenario:**

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

**Tasks:**

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

**Deliverables:**

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

**Approach:**

To model the data flow for fetching COVID-19 statistics and displaying them to the user, we can use a simple diagram that illustrates the interaction between the application and the external COVID-19 statistics API.

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

| User Input | | Application | | COVID-19 API|

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

| | |

| User requests | |

| COVID-19 statistics | |

|--------------------> | |

| | Fetch COVID-19 data |

| |--------------------> |

| | |

| | Receive COVID-19 |

| | data |

| |<-------------------- |

| | |

| Display COVID-19 | |

| information | |

|<-------------------- | |

| | |

**Pseudocode:**

# Define the API endpoint

API\_ENDPOINT = "https://disease.sh/v3/covid-19/countries/{}"

# Define the main function

def main():

# Get the user input for the region

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

# Fetch the COVID-19 data from the API

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

# Display the COVID-19 statistics

display\_covid\_info(covid\_data)

# Define the function to fetch COVID-19 data

def get\_covid\_data(region):

# Construct the API request URL

url = API\_ENDPOINT.format(region)

# Send a GET request to the API

response = requests.get(url)

# Parse the JSON response

covid\_data = json.loads(response.text)

return covid\_data

# Define the function to display COVID-19 statistics

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

# Extract the relevant statistics

country = covid\_data["country"]

cases = covid\_data["cases"]

recoveries = covid\_data["recovered"]

deaths = covid\_data["deaths"]

# Print the statistics

print(f"Country: {country}")

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

print(f"Total Recoveries: {recoveries}")

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

# Run the main function

main() |

**Detailed explanation of the actual code:**

1. The code starts by importing the necessary libraries: requests for making HTTP requests and json for parsing JSON data.
2. The COVID\_API\_ENDPOINT constant is defined, which represents the URL of the disease.sh API endpoint for fetching COVID-19 data by country. The {} placeholder will be replaced with the user-specified region.
3. The get\_covid\_data function is defined to fetch real-time COVID-19 data from the disease.sh API. It takes a region parameter, which can be a country, state, or city.
4. Inside the get\_covid\_data function:

* The url variable is constructed by replacing the {} placeholder in COVID\_API\_ENDPOINT with the region parameter using the format method.
* A GET request is sent to the constructed url using the requests.get function, and the response is stored in the response variable.
* The JSON data from the API response is extracted using the response.json() method and returned

5 .The display\_covid\_info function is defined to process and display the COVID-19 statistics to the user. It takes a covid\_data parameter, which is the JSON data returned by the API.

6.Inside the display\_covid\_info function:

* The relevant COVID-19 statistics are extracted from the covid\_data dictionary, such as the country name, total cases, recoveries, and deaths.
* The extracted statistics are printed using formatted strings (f-strings) to display them in a readable format.

7.The main function is defined as the entry point of the application.

8. Inside the main function:

* The user is prompted to enter a country, state, or city using the input function, and the input is stored in the region variable.
* The get\_covid\_data function is called with the region parameter, and the returned COVID-19 data is stored in the covid\_data variable.
* The display\_covid\_info function is called with the covid\_data parameter to display the COVID-19 statistics to the user.

9.Finally, the if \_\_name\_\_ == "\_\_main\_\_": block ensures that the main function is executed only when the script is run directly (not imported as a module).

**Assumptions made (if any):**

1. The user has a stable internet connection to fetch data from the COVID-19 API.
2. The application is designed for a single user request at a time.
3. The user will enter a valid region (country, state, or city) as input.
4. The API will always return a valid JSON response.

**Limitations:**

1. The application does not handle errors or exceptions properly. It assumes that the API will always return a valid response.
2. The application does not provide any error messages or feedback to the user if the API request fails.
3. The application does not support multiple regions or comparisons between regions.
4. The application does not provide any historical data or trends.
5. The application does not include any data visualization or interactive features.

**Code:**

**import urllib.request**

**import json**

**def get\_covid\_data(region):**

**"""**

**Fetch real-time COVID-19 data from the disease.sh API.**

**"""**

**try:**

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

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

**data = json.load(response)**

**return data**

**except urllib.error.URLError as e:**

**print(f"Error fetching COVID-19 data: {e}")**

**return None**

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

**"""**

**Process and display the COVID-19 statistics to the user.**

**"""**

**if covid\_data:**

**country = covid\_data["country"]**

**cases = covid\_data["cases"]**

**recoveries = covid\_data["recovered"]**

**deaths = covid\_data["deaths"]**

**print(f"Country: {country}")**

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

**print(f"Total Recoveries: {recoveries}")**

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

**else:**

**print("Unable to fetch COVID-19 data.")**

**def main():**

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

**covid\_data = get\_covid\_data(region)**

**display\_covid\_info(covid\_data)**

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

**main()**

**Sample Output / Screen Shots :**

