**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 |**

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

**Pseudocode:**

# Function to fetch weather data from OpenWeatherMap API

def fetch\_weather\_data(location):

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

api\_url = f"http://api.openweathermap.org/data/2.5/weather?q={location}&appid={api\_key}"

response = requests.get(api\_url)

if response.status\_code == 200:

data = response.json()

return data

else:

return None

# Function to display weather data to the user

def display\_weather\_data(data):

if data:

print(f"Current Weather in {data['name']}:")

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

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

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

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

else:

print("Error fetching weather data.")

# Main function to handle user input and display weather data

def main():

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

data = fetch\_weather\_data(location)

display\_weather\_data(data)

# Run the main function

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

main()

**Detailed explanation of the actual code:**

1. fetch\_weather\_data: This function takes a location as input and fetches the weather data from the OpenWeatherMap API. It uses the requests library to send a GET request to the API with the location and API key. If the response is successful (200 status code), it parses the JSON response and returns the data. Otherwise, it returns None.
2. display\_weather\_data: This function takes the weather data as input and displays the current weather information to the user. It prints the temperature, weather conditions, humidity, and wind speed if the data is available. If there is an error fetching the data, it prints an error message.
3. main: This function handles user input and calls the fetch\_weather\_data and display\_weather\_data functions. It prompts the user to enter a location, fetches the weather data, and then displays the data to the user.

**Assumptions made (if any):**

1. The user will always enter a valid location.
2. The OpenWeatherMap API will always return data in the expected format.

**Limitations:**

1. The application does not handle errors in the API response.
2. It does not validate user input for location.
3. It does not provide any additional features like historical weather data or weather forecasts.

**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.

**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 structure

define Product, Warehouse, and InventoryLevel classes

# Implement the inventory tracking application

class InventoryTrackingApp:

def \_\_init\_\_(self, products, warehouses):

self.products = products

self.warehouses = warehouses

self.inventory\_levels = self.initialize\_inventory\_levels()

def initialize\_inventory\_levels(self):

# Initialize inventory levels based on the provided data

inventory\_levels = []

for product in self.products:

for warehouse in self.warehouses:

inventory\_level = InventoryLevel(product, warehouse, initial\_stock)

inventory\_levels.append(inventory\_level)

return inventory\_levels

def track\_inventory(self):

# Monitor inventory levels and send alerts when stock falls below threshold

for inventory\_level in self.inventory\_levels:

if inventory\_level.current\_stock < inventory\_level.reorder\_point:

send\_alert(inventory\_level)

# Optimize inventory ordering

def calculate\_reorder\_point(product, warehouse, historical\_sales, lead\_time):

# Calculate the optimal reorder point based on historical sales and lead time

safety\_stock = calculate\_safety\_stock(historical\_sales, lead\_time)

reorder\_point = safety\_stock + (average\_daily\_sales \* lead\_time)

return reorder\_point

def calculate\_reorder\_quantity(product, warehouse, historical\_sales, lead\_time, reorder\_point):

# Calculate the optimal reorder quantity based on historical sales, lead time, and reorder point

economic\_order\_quantity = calculate\_economic\_order\_quantity(historical\_sales, holding\_cost, ordering\_cost)

reorder\_quantity = max(economic\_order\_quantity, product.minimum\_order\_quantity)

return reorder\_quantity

# Generate reports

def generate\_inventory\_turnover\_report(inventory\_levels):

# Calculate and report on inventory turnover rates

pass

def generate\_stockout\_report(inventory\_levels):

# Report on stockout occurrences and their cost implications

pass

def generate\_overstock\_report(inventory\_levels):

# Report on overstock situations and their cost implications

pass

# User interaction

class InventoryManagementUI:

def \_\_init\_\_(self, inventory\_tracking\_app):

self.inventory\_tracking\_app = inventory\_tracking\_app

def display\_inventory\_information(self, product\_id):

# Allow users to view current stock levels, reorder recommendations, and historical data

pass

**Detailed explanation of the actual code:**

1. Defining the Inventory System Structure: I will create classes for Product, Warehouse, and InventoryLevel to represent the components of the inventory system. These classes will store relevant information about each entity, such as product details, warehouse locations, and current stock levels.
2. Implementing the Inventory Tracking Application: The InventoryTrackingApp class will be responsible for initializing the inventory levels, tracking the current stock, and sending alerts when stock falls below a certain threshold. The track\_inventory() method will continuously monitor the inventory levels and trigger alerts as needed.
3. Optimizing Inventory Ordering: The calculate\_reorder\_point() and calculate\_reorder\_quantity() functions will implement algorithms to determine the optimal reorder point and quantity for each product in each warehouse. These calculations will be based on historical sales data, lead times, and other relevant factors.
4. Generating Reports: The generate\_inventory\_turnover\_report(), generate\_stockout\_report(), and generate\_overstock\_report() functions will generate the required reports on inventory performance, stockouts, and overstock situations, respectively.
5. User Interaction: The InventoryManagementUI class will provide a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts. Users will be able to input product IDs or names to retrieve the desired information**.**

**Assumptions made (:if any)**

1. The company has a well-defined product catalog and warehouse locations.
2. Historical sales data and lead times are available for the inventory optimization algorithms.
3. The company has defined thresholds for reorder points and minimum order quantities.
4. The cost of holding inventory and placing orders are known.

**Limitations:**

1. The current implementation assumes constant lead times, which may not always be the case in real-world scenarios.
2. The demand forecasting algorithms are not included in the pseudocode, as they can be complex and require more detailed analysis of the company's sales patterns.
3. The user interface is only briefly mentioned, and a more comprehensive design would be required for a production-ready system.

**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:**

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|**

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

The key steps in the data flow are:

* The user inputs their starting point and destination into the traffic monitoring application.
* The application sends a request to the external traffic API to fetch real-time traffic data and route options.
* The traffic API provides the requested data, which the application stores in a local database.
* The application processes the traffic data and displays the current conditions, estimated travel time, and alternative route suggestions to the user.
* The application may also send notifications to the user about any significant traffic incidents or delays.

**Pseudocode:**

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:**

1. 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.
2. The API response is then parsed to extract the following information:
3. Current traffic conditions: data["routes"]["legs"]["duration\_in\_traffic"]["text"]
4. Estimated travel time: data["routes"]["legs"]["duration"]["text"]
5. Alternative route options, including distance, duration, and duration in traffic for each route
6. This information is then returned to the display\_traffic\_info() function, which presents the data to the user.

**Assumptions made (if any):**

1.The user has a valid API key for the Google Maps Directions API.

2.The API provides accurate and up-to-date traffic information.

3.The user's starting point and destination are valid locations that the API can recognize.

**Limitations:**

1. 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.
2. The application does not provide real-time updates or notifications. It only displays the traffic information when the user requests it.
3. 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**

**4Scenario:**

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:**

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

**| User Input |**

**| (Region Query) |**

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

**|**

**v**

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

**| Application Logic |**

**| (Fetch Data from API)|**

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

**|**

**v**

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

**| External COVID-19 |**

**| Statistics API |**

**| (e.g., disease.sh) |**

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

**|**

**v**

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

**| Display Results |**

**| (Cases, Recoveries, |**

**| Deaths for Region) |**

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

**The key steps are:**

1. The user inputs a region (country, state, or city) into the application.
2. The application sends a request to the COVID-19 API to fetch the real-time statistics for the specified region.
3. The COVID-19 API processes the request and returns the current case, recovery, and death data.
4. The application receives the COVID-19 statistics and displays the information to the user.

**Pseudocode:**

import requests

def get\_covid\_stats(region):

"""

Fetch COVID-19 statistics for the specified region from the API.

"""

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

response = requests.get(api\_url)

data = response.json()

cases = data["cases"]

recoveries = data["recovered"]

deaths = data["deaths"]

return cases, recoveries, deaths

def display\_covid\_stats(cases, recoveries, deaths):

"""

Display the COVID-19 statistics in a user-friendly format.

"""

print(f"Current COVID-19 Statistics:")

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

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

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

def main():

"""

Main function to handle user input and display COVID-19 statistics.

"""

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

cases, recoveries, deaths = get\_covid\_stats(region)

display\_covid\_stats(cases, recoveries, deaths)

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

main()

**Detailed explanation of the actual code:**

* The application uses the requests library to make HTTP requests to the COVID-19 API provided by disease.sh. The get\_covid\_stats function takes a region (country, state, or city) as input and returns the current number of cases, recoveries, and deaths for that region.
* The display\_covid\_stats function is responsible for formatting and printing the COVID-19 statistics in a user-friendly way. It takes the cases, recoveries, and deaths data as input and displays them with appropriate formatting (e.g., adding commas to large numbers).
* The main function is the entry point of the application. It prompts the user to enter a region, calls the get\_covid\_stats function to fetch the data, and then passes the results to the display\_covid\_stats function to display the information.

**Assumptions made (if any):**

* The application assumes that the disease.sh API is available and providing accurate real-time COVID-19 data.
* The application assumes that the user will input a valid region (country, state, or city) that the API can recognize.
* Potential Improvements:
* Add error handling to the application to gracefully handle API errors or invalid user input.
* Provide additional features, such as the ability to display historical COVID-19 data, trends, or visualizations.
* Integrate the application with a user interface (e.g., a web application or a mobile app) to improve the user experience.
* Allow users to select multiple regions and compare the COVID-19 statistics side-by-side.
* Provide the ability to set alerts or notifications for significant changes in COVID-19 statistics.

**Limitations:**

1. The API may have rate limits that restrict the number of requests.
2. The data may not always be up-to-date due to delays in reporting.
3. The application currently only handles countries; state and city-level queries may require additional endpoints.

**Code:**

import urllib.request

import json

# Function to get COVID-19 statistics from disease.sh API

def get\_covid\_data(location):

# Set the API endpoint and parameters

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

# Send a GET request to the API

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

# Load the JSON response

data = json.load(response)

# Extract the relevant COVID-19 data

cases = data["cases"]

recoveries = data["recovered"]

deaths = data["deaths"]

# Return the COVID-19 data as a dictionary

return {"cases": cases, "recoveries": recoveries, "deaths": deaths}

# Function to display the COVID-19 statistics

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

# Print the COVID-19 data in a readable format

print("Current COVID-19 Statistics:")

print(f"Cases: {covid\_data['cases']}")

print(f"Recoveries: {covid\_data['recoveries']}")

print(f"Deaths: {covid\_data['deaths']}")

# Main function to run the program

def main():

# Get the location from the user

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

# Get the COVID-19 data

try:

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

except urllib.error.HTTPError:

print("Failed to retrieve COVID-19 data. Please check the location and try again.")

return

# Display the COVID-19 data

display\_covid\_data(covid\_data)

# Run the main function

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

main()

**Sample Output / Screen Shots**



