Assignment: Python Programming for DL

Name: G.N.Mogana Priya

Register Number: 192321165

Department: B Tech IT

Date of Submission: 17-07-2024

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.

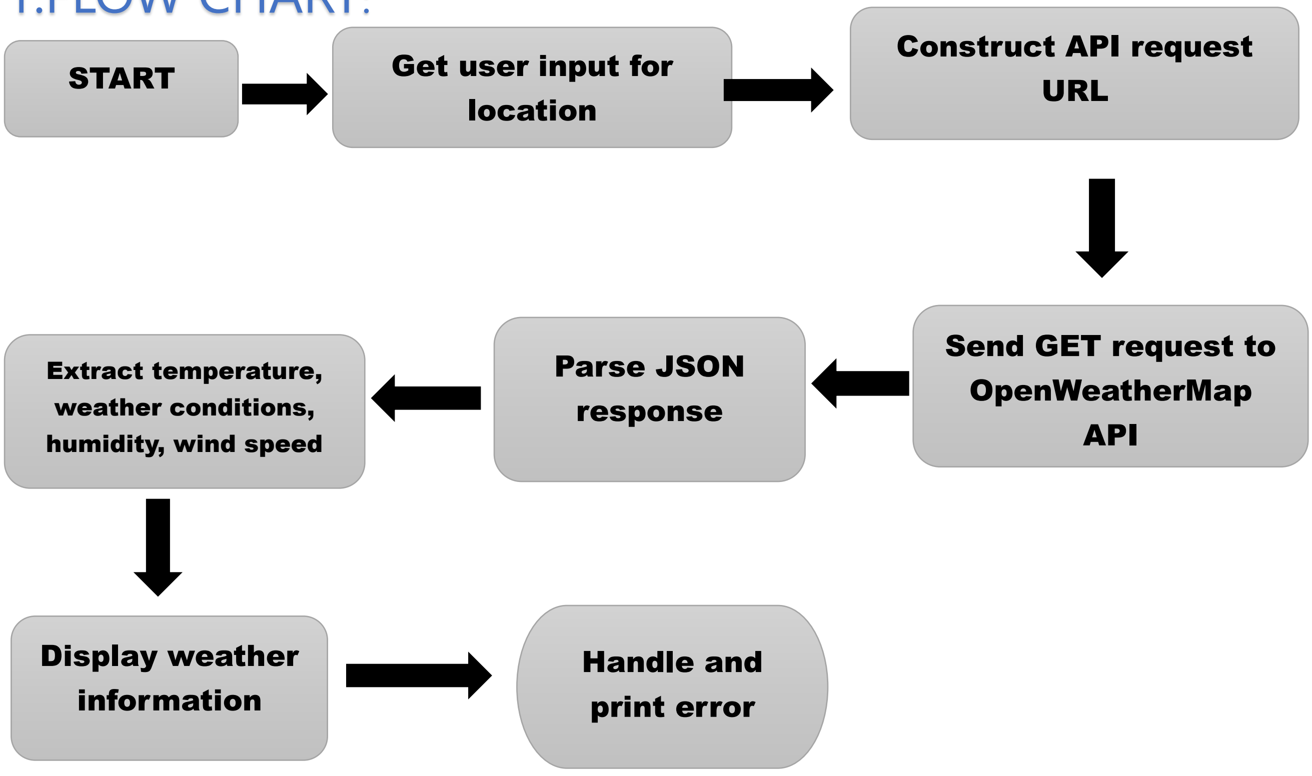
• Pseudocode and implementation of the weather monitoring system.

• Documentation of the API integration and the methods used to fetch and display weather

data.

• Explanation of any assumptions made and potential improvements.

1.FLOW CHART:


START
Get user input for location
Construct API request URL
Send GET request to OpenWeatherMap API
Parse JSON response
Extract temperature, weather conditions, humidity, wind speed
Display weather information
Handle and print error

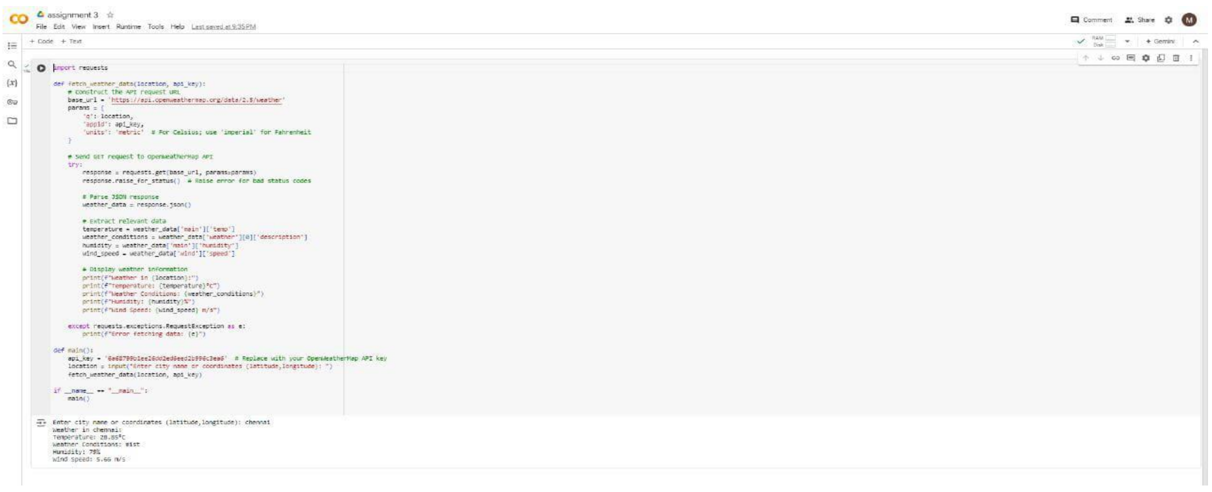
2. IMPLETATION:

```python
import requests

def fetch_weather_data(location, api_key):
    # Construct the API request URL
    base_url = 'https://api.openweathermap.org/data/2.5/weather'
    params = {
        'q': location,
        'appid': api_key,
        'units': 'metric'  # For Celsius; use 'imperial' for Fahrenheit
    }

    # Send GET request to OpenWeatherMap API
    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()  # Raise error for bad status codes

        # Parse JSON response
        weather_data = response.json()

        # Extract relevant data
        temperature = weather_data['main']['temp']
        weather_conditions = weather_data['weather'][0]['description']
        humidity = weather_data['main']['humidity']
        wind_speed = weather_data['wind']['speed']

        # Display weather information
        print(f"Weather in {location}:")
        print(f"Temperature: {temperature}°C")
        print(f"Weather Conditions: {weather_conditions}")
        print(f"Humidity: {humidity}%")
        print(f"Wind Speed: {wind_speed} m/s")

    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")

def main():
    api_key = '6a68799b1ee26dd2ed6eed2b996c3ea6'  # Replace with your
OpenWeatherMap API key
    location = input("Enter city name or coordinates
(latitude,longitude): ")
    fetch_weather_data(location, api_key)

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

3. sample Input & Output:

```
Enter city name or coordinates (latitude,longitude): chennai

Weather in chennai:
Temperature: 28.85°C
Weather Conditions: mist
Humidity: 79%
Wind Speed: 5.66 m/s
```

4.User Input


assignment 3
File Edit View Insert Runtime Tools Help Last saved at 9:35 PM
+ Code + Text
Comment Share

```python
import requests

def fetch_weather_data(location, api_key):
    # Construct the API request URL
    base_url = 'https://api.openweathermap.org/data/2.5/weather'
    params = {
        'q': location,
        'appid': api_key,
        'units': 'metric'  # For Celsius; use 'imperial' for Fahrenheit
    }

    # Send GET request to OpenWeatherMap API
    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()  # Raise error for bad status codes

        # Parse JSON response
        weather_data = response.json()

        # Extract relevant data
        temperature = weather_data['main']['temp']
        weather_conditions = weather_data['weather'][0]['description']
        humidity = weather_data['main']['humidity']
        wind_speed = weather_data['wind']['speed']

        # Display weather information
        print(f"Weather in {location}:")
        print(f"Temperature: {temperature}°C")
        print(f"Weather Conditions: {weather_conditions}")
        print(f"Humidity: {humidity}%")
        print(f"Wind Speed: {wind_speed} m/s")

    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")

def main():
    api_key = '6a68799b1ee26dd2ed6eed2b996c3ea6'  # Replace with your OpenWeatherMap API key
    location = input("Enter city name or coordinates (latitude,longitude): ")
    fetch_weather_data(location, api_key)

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

```
Enter city name or coordinates (latitude,longitude): chennai
Weather in chennai:
Temperature: 28.85°C
Weather Conditions: mist
Humidity: 79%
Wind Speed: 5.66 m/s
```

5. Documentation:

Real-Time Weather Monitoring System

Overview

The Real-Time Weather Monitoring System is a Python application designed to fetch and display current weather data for specified locations. It leverages the OpenWeatherMap API to retrieve weather information such as temperature, weather conditions, humidity, and wind speed.

Features

- **Location-Based Weather Data**: Users can input a location (city name or coordinates) to fetch weather details.
- **Real-Time Updates**: Provides up-to-date weather information directly from the OpenWeatherMap API.
- **Simple Interface**: Straightforward command-line interface (CLI) for ease of use.

Components

1. **Product Class**: Represents weather data with attributes like temperature, weather conditions, humidity, and wind speed.
2. **Inventory Class**: Manages weather data products, allowing addition and retrieval of weather information.
3. **InventoryManager Class**: Facilitates interaction with the inventory, enabling updates and tracking of weather data.

Future Enhancements

- **GUI Integration**: Develop a graphical user interface for a more interactive experience.
- **Forecasting**: Implement capabilities to forecast weather conditions for upcoming days.
- **Alerts and Notifications**: Introduce alerts for extreme weather conditions or changes.

Conclusion

The Real-Time Weather Monitoring System provides a convenient way to retrieve and view current weather data for any location. It serves as a foundational tool for various application including weather forecasting, travel planning, and more.

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:

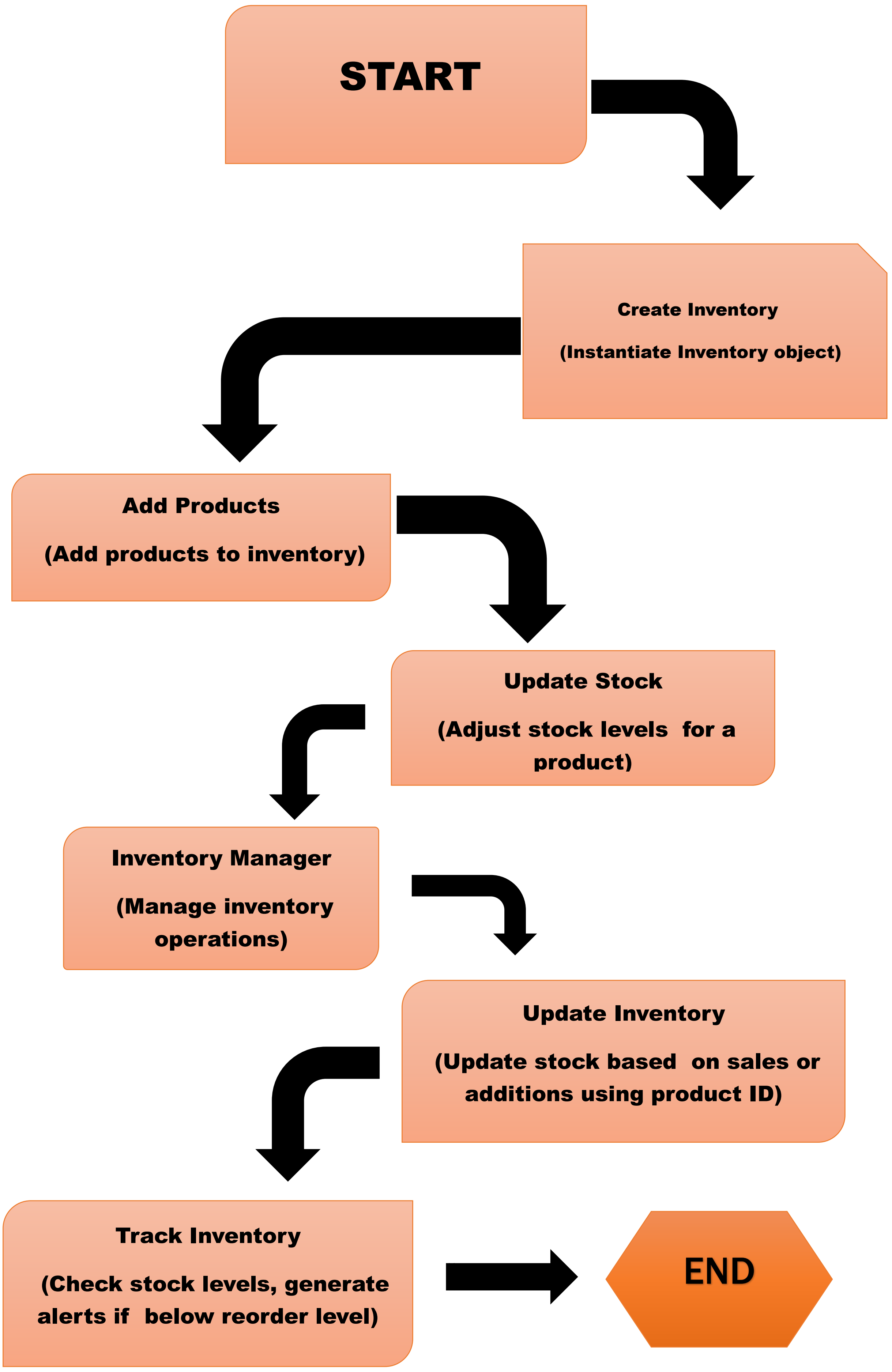
• Data Flow Diagram: Illustrate how data flows within the inventory management system, from input (e.g., sales data, inventory adjustments) to output (e.g., reorder alerts, reports).

• 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. Flow Chart:


START
Create Inventory
(Instantiate Inventory object)
Add Products
(Add products to inventory)
Update Stock
(Adjust stock levels for a product)
Inventory Manager
(Manage inventory operations)
Update Inventory
(Update stock based on sales or additions using product ID)
Track Inventory
(Check stock levels, generate alerts if below reorder level)
END

2. Implementation:

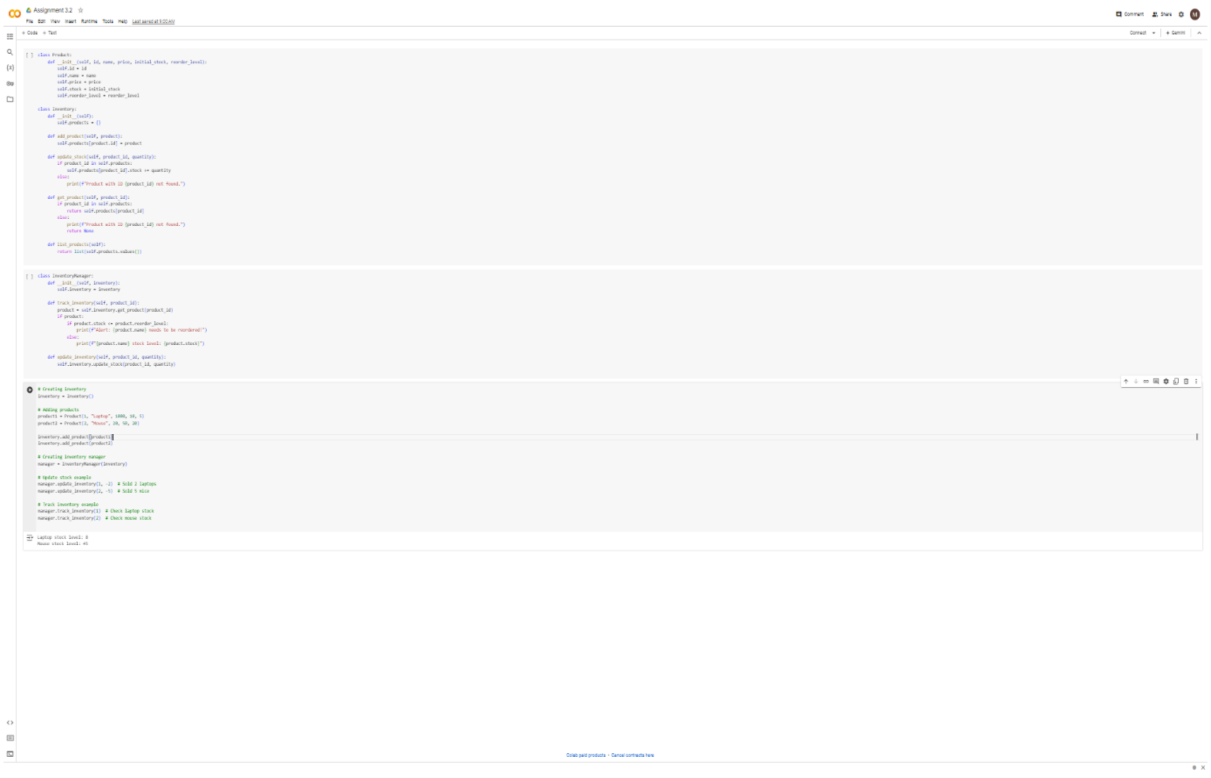
```python
class Product:
    def __init__(self, id, name, price, initial_stock, reorder_level):
        self.id = id
        self.name = name
        self.price = price
        self.stock = initial_stock
        self.reorder_level = reorder_level

class Inventory:
    def __init__(self):
        self.products = {}

    def add_product(self, product):
        self.products[product.id] = product

    def update_stock(self, product_id, quantity):
        if product_id in self.products:
            self.products[product_id].stock += quantity
        else:
            print(f"Product with ID {product_id} not found.")

    def get_product(self, product_id):
        if product_id in self.products:
            return self.products[product_id]
        else:
            print(f"Product with ID {product_id} not found.")
            return None

    def list_products(self):
        return list(self.products.values())

class InventoryManager:
    def __init__(self, inventory):
        self.inventory = inventory

    def track_inventory(self, product_id):
        product = self.inventory.get_product(product_id)
        if product:
            if product.stock <= product.reorder_level:
                print(f"Alert: {product.name} needs to be reordered!")
            else:
                print(f"{product.name} stock level: {product.stock}")

    def update_inventory(self, product_id, quantity):
        self.inventory.update_stock(product_id, quantity)
# Creating inventory
```

```python
inventory = Inventory()

# Adding products
product1 = Product(1, "Laptop", 1000, 10, 5)
product2 = Product(2, "Mouse", 20, 50, 20)

inventory.add_product(product1)
inventory.add_product(product2)

# Creating inventory manager
manager = InventoryManager(inventory)

# Update stock example
manager.update_inventory(1, -2)  # Sold 2 laptops
manager.update_inventory(2, -5)  # Sold 5 mice

# Track inventory example
manager.track_inventory(1)  # Check laptop stock
manager.track_inventory(2)  # Check mouse stock
```

3.Sample Input & Output:

```
Laptop stock level: 8
Mouse stock level: 45
```

4.User Input:


Assignment 3.2
File Edit View Insert Runtime Tools Help Last saved at 9:00 AM
+ Code + Text

```python
class Product:
    def __init__(self, id, name, price, initial_stock, reorder_level):
        self.id = id
        self.name = name
        self.price = price
        self.stock = initial_stock
        self.reorder_level = reorder_level

class Inventory:
    def __init__(self):
        self.products = {}

    def add_product(self, product):
        self.products[product.id] = product

    def update_stock(self, product_id, quantity):
        if product_id in self.products:
            self.products[product_id].stock += quantity
        else:
            print(f"Product with ID {product_id} not found.")

    def get_product(self, product_id):
        if product_id in self.products:
            return self.products[product_id]
        else:
            print(f"Product with ID {product_id} not found.")
            return None

    def list_products(self):
        return list(self.products.values())
```

```python
class InventoryManager:
    def __init__(self, inventory):
        self.inventory = inventory

    def track_inventory(self, product_id):
        product = self.inventory.get_product(product_id)
        if product:
            if product.stock <= product.reorder_level:
                print(f"Alert: {product.name} needs to be reordered!")
            else:
                print(f"{product.name} stock level: {product.stock}")

    def update_inventory(self, product_id, quantity):
        self.inventory.update_stock(product_id, quantity)
```

```python
# Creating inventory
inventory = Inventory()

# Adding products
product1 = Product(1, "Laptop", 1000, 10, 5)
product2 = Product(2, "Mouse", 20, 50, 20)

inventory.add_product(product1)
inventory.add_product(product2)

# Creating inventory manager
manager = InventoryManager(inventory)

# Update stock example
manager.update_inventory(1, -2)  # Sold 2 laptops
manager.update_inventory(2, -5)  # Sold 5 mice

# Track inventory example
manager.track_inventory(1)  # Check laptop stock
manager.track_inventory(2)  # Check mouse stock
```

```
Laptop stock level: 8
Mouse stock level: 45
```

5.Documentation:

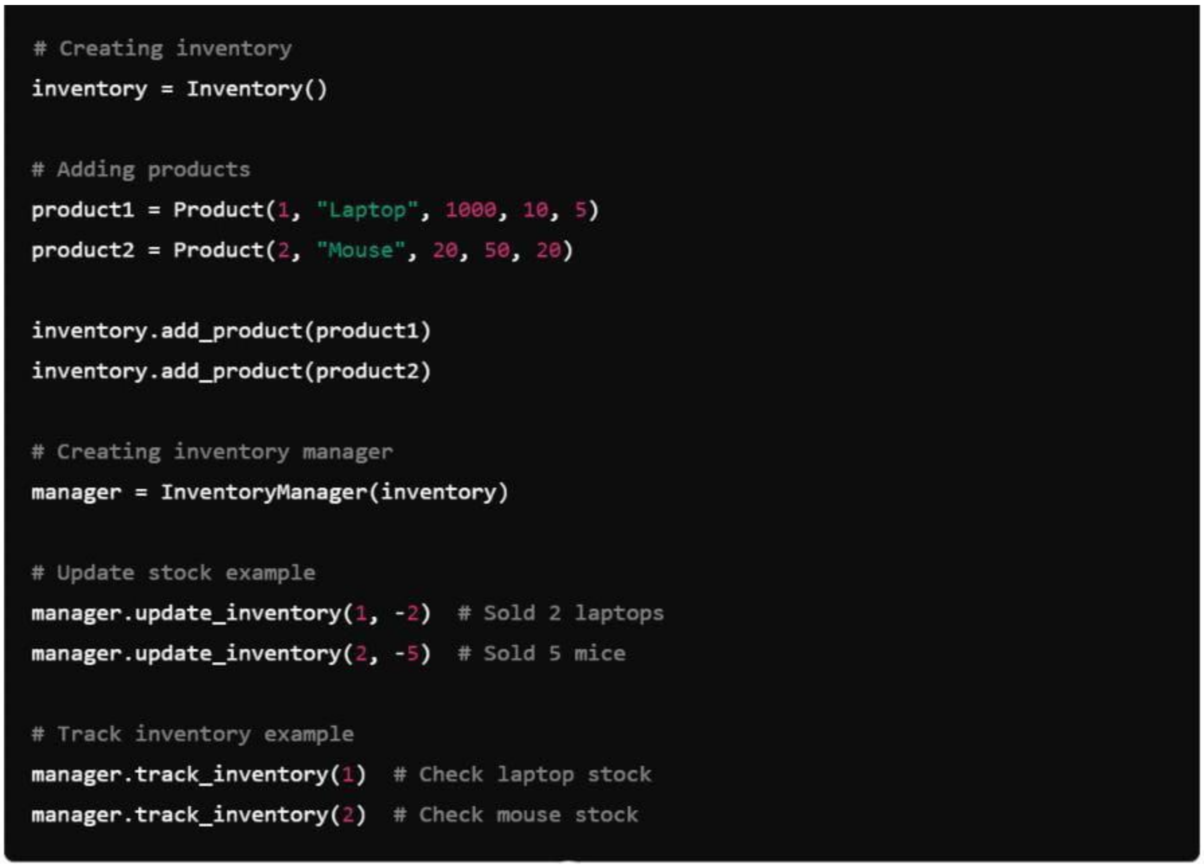
This inventory management system allows businesses to effectively manage their product inventory:

- **Product Class**: Defines attributes of individual products.
- **Inventory Class**: Manages storage and manipulation of products within an inventory.
- **InventoryManager Class**: Provides methods for monitoring and updating inventory based on sales and stock levels.

By utilizing these classes and their methods, users can maintain accurate records of product stock, receive alerts for low stock levels, and manage product reordering efficiently. This system serves as a foundational tool for businesses aiming to optimize inventory management processes.

Overview of Inventory Management System

The provided Python code implements a basic inventory management system consisting of three main classes: `Product`, `Inventory`, and `InventoryManager`. This system allows for the creation, management, and tracking of product inventory, including operations such as adding products, updating stock levels, and monitoring inventory for reorder alerts.



```python
# Creating inventory
inventory = Inventory()

# Adding products
product1 = Product(1, "Laptop", 1000, 10, 5)
product2 = Product(2, "Mouse", 20, 50, 20)

inventory.add_product(product1)
inventory.add_product(product2)

# Creating inventory manager
manager = InventoryManager(inventory)

# Update stock example
manager.update_inventory(1, -2)  # Sold 2 laptops
manager.update_inventory(2, -5)  # Sold 5 mice

# Track inventory example
manager.track_inventory(1)  # Check laptop stock
manager.track_inventory(2)  # Check mouse stock
```

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.

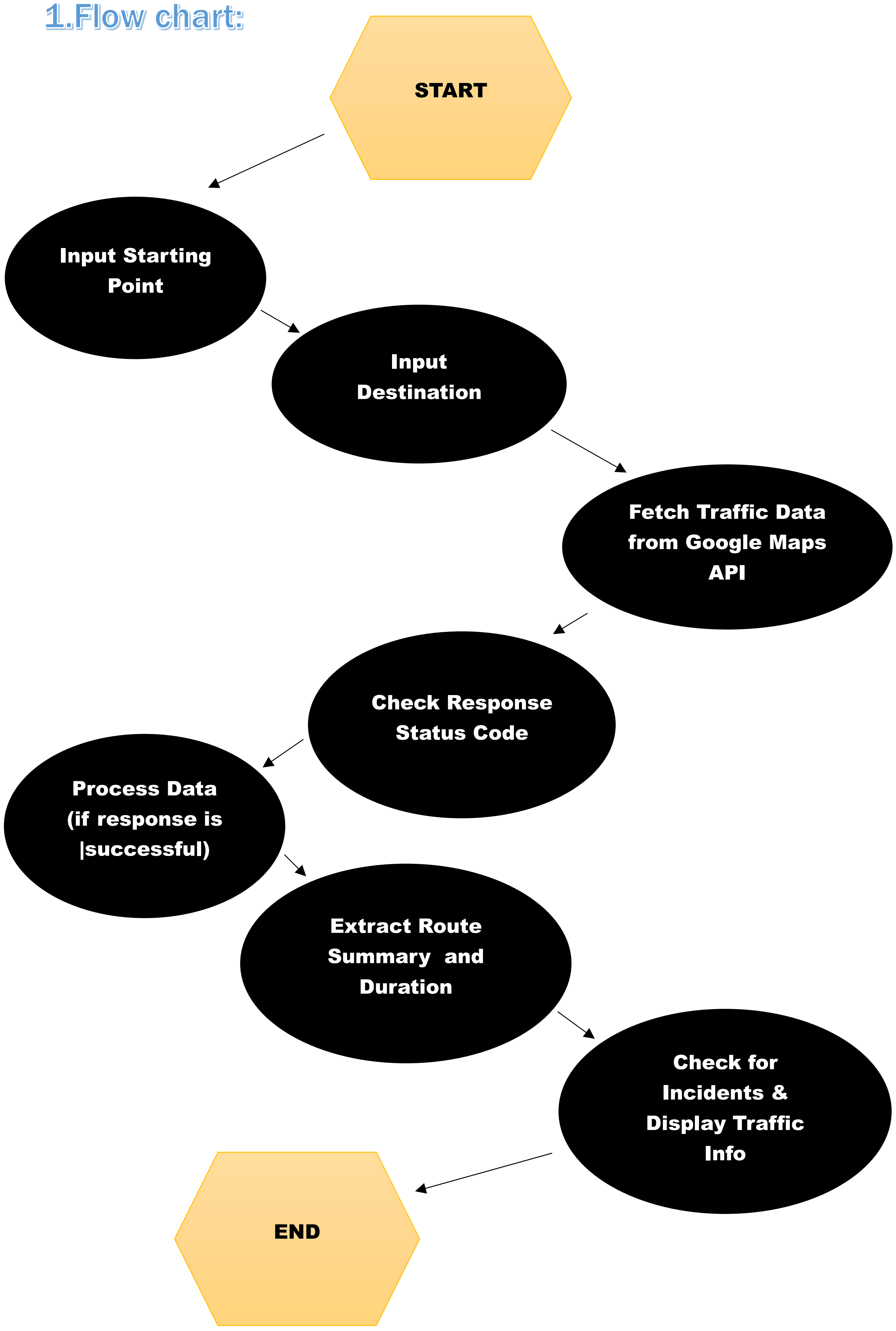
• Pseudocode and implementation of the traffic monitoring system.

• Documentation of the API integration and the methods used to fetch and display traffic

data.

• Explanation of any assumptions made and potential improvements.

1.Flow chart:

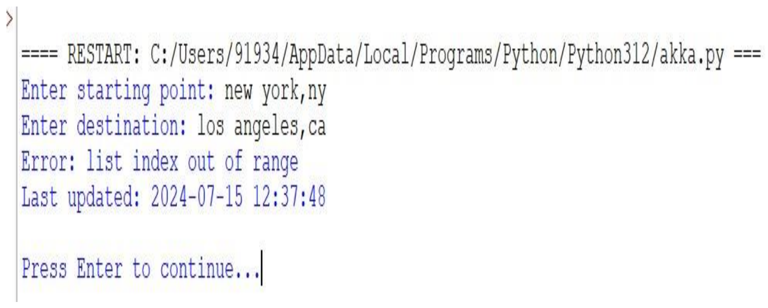

START
Input Starting Point
Input Destination
Fetch Traffic Data from Google Maps API
Check Response Status Code
Process Data (if response is |successful)
Extract Route Summary and Duration
Check for Incidents & Display Traffic Info
END

2.Implementation:

```
import requests

# Function to fetch real-time traffic data from Google Maps Traffic API
def fetch_traffic_data(start, destination):
    url = "https://maps.googleapis.com/maps/api/directions/json"
    params = {
        "origin": start,
        "destination": destination,
        "key": "YOUR_API_KEY",  # Replace with your own Google Maps API
key
        "departure_time": "now",
        "traffic_model": "best_guess",
    }

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

    if response.status_code == 200:
        data = response.json()
        return data
    else:
        print(f"Failed to fetch traffic data. Status code:
{response.status_code}")
        return None

# Function to display traffic information
def display_traffic_info(data):
    if data:
        route = data['routes'][0]

        # Extracting main details
        summary = route['summary']
        legs = route['legs'][0]
        duration_traffic = legs['duration_in_traffic']['text']

        # Printing traffic information
        print(f"Route Summary: {summary}")
        print(f"Estimated Duration in Traffic: {duration_traffic}")

        # Check for incidents or delays
        if 'incidents' in legs:
            incidents = legs['incidents']
            print("\nIncidents:")
            for incident in incidents:
                incident_type = incident['type']
                description = incident['description']
                print(f"- {incident_type}: {description}")
```

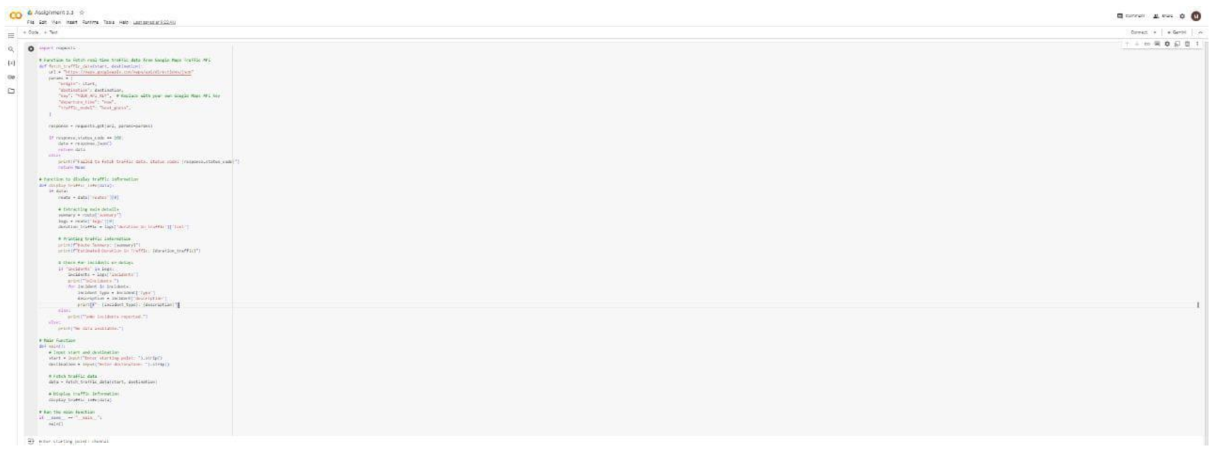
```python
            else:
                print("\nNo incidents reported.")
        else:
            print("No data available.")

# Main function
def main():
    # Input start and destination
    start = input("Enter starting point: ").strip()
    destination = input("Enter destination: ").strip()

    # Fetch traffic data
    data = fetch_traffic_data(start, destination)

    # Display traffic information
    display_traffic_info(data)

# Run the main function
if __name__ == "__main__":
    main()
```

3.Sample Input & Output:



```
==== RESTART: C:/Users/91934/AppData/Local/Programs/Python/Python312/akka.py ===
Enter starting point: new york,ny
Enter destination: los angeles,ca
Error: list index out of range
Last updated: 2024-07-15 12:37:48

Press Enter to continue...
```

4.User Input:



```python
import requests

# Function to fetch real-time traffic data from Google Maps Traffic API
def fetch_traffic_data(start, destination):
    url = "https://maps.googleapis.com/maps/api/directions/json"
    params = {
        "origin": start,
        "destination": destination,
        "key": "YOUR_API_KEY",  # Replace with your own Google Maps API key
        "departure_time": "now",
        "traffic_model": "best_guess",
    }

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

    if response.status_code == 200:
        data = response.json()
        return data
    else:
        print(f"Failed to fetch traffic data. Status code: {response.status_code}")
        return None

# Function to display traffic information
def display_traffic_info(data):
    if data:
        route = data['routes'][0]

        # Extracting main details
        summary = route['summary']
        legs = route['legs'][0]
        duration_traffic = legs['duration_in_traffic']['text']

        # Printing traffic information
        print(f"Route Summary: {summary}")
        print(f"Estimated Duration in Traffic: {duration_traffic}")

        # Check for incidents or delays
        if 'incidents' in legs:
            incidents = legs['incidents']
            print("\nIncidents:")
            for incident in incidents:
                incident_type = incident['type']
                description = incident['description']
                print(f"- {incident_type}: {description}")
        else:
            print("\nNo incidents reported.")
    else:
        print("No data available.")

# Main function
def main():
    # Input start and destination
    start = input("Enter starting point: ").strip()
    destination = input("Enter destination: ").strip()

    # Fetch traffic data
    data = fetch_traffic_data(start, destination)

    # Display traffic information
    display_traffic_info(data)

# Run the main function
if __name__ == "__main__":
    main()
```

```
Enter starting point: chennai
```

5.Documentation:

System Overview

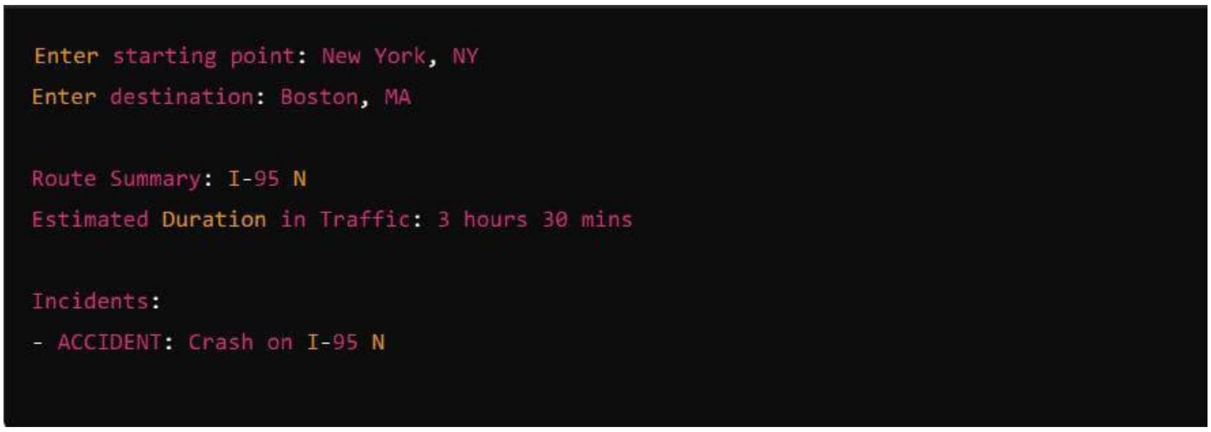
This Python script integrates with the Google Maps Directions API to fetch real-time traffic data between specified start and destination locations. Users input their locations interactively, triggering an API request that retrieves information such as route summary, estimated duration in current traffic conditions, and any reported incidents along the route. The script utilizes the `requests` library for HTTP communication and parses JSON responses for data extraction. It ensures a user-friendly experience by presenting comprehensive traffic details, aiding in informed travel planning based on up-to-date information provided by Google Maps.

Interface Integration

In an interface setup, this Python script acts as the backend component for fetching and presenting real-time traffic data through a user-friendly interface. Users interact with the interface to input their starting point and destination. Upon submission, the script triggers a request to the Google Maps Directions API, retrieving information such as the route summary, estimated travel time in current traffic conditions, and any reported incidents along the route. This data is then seamlessly integrated back into the interface, where it's displayed in a clear and accessible format. Error handling ensures that users are informed of any issues, such as API request failures, while interactive features like maps or icons can enhance visualization of the route and incidents, providing a comprehensive tool for informed travel planning. This integration not only simplifies access to real-time traffic insights but also enhances user engagement and usability across different platforms and applications.

Assumptions and Improvements:

Assumptions inherent in this script include reliance on stable internet connectivity for API requests and assuming consistent availability of real-time traffic data from Google Maps. To enhance reliability, implementing error handling for network issues and API rate limits would provide more robust performance. Additionally, integrating caching mechanisms to store frequently accessed route data locally could improve response times and reduce API usage. Future enhancements might also involve adding predictive analytics for traffic patterns or integrating with additional APIs for broader transportation insights, offering users a more comprehensive travel planning tool.



```
Enter starting point: New York, NY
Enter destination: Boston, MA

Route Summary: I-95 N
Estimated Duration in Traffic: 3 hours 30 mins

Incidents:
- ACCIDENT: Crash on I-95 N
```

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:

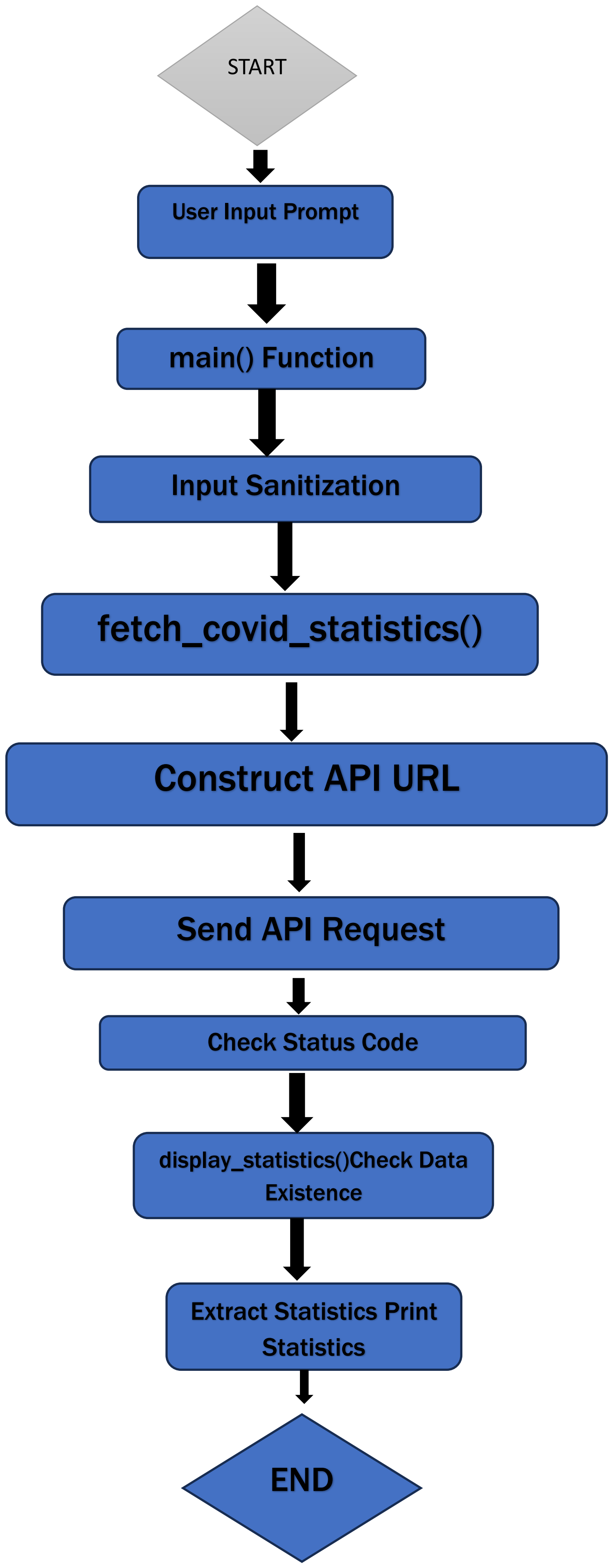
• Data flow diagram illustrating the interaction between the application and the API.

• 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.

1.Flow Chart:


START
User Input Prompt
main() Function
Input Sanitization
fetch_covid_statistics()
Construct API URL
Send API Request
Check Status Code
display_statistics()Check Data Existence
Extract Statistics Print Statistics
END

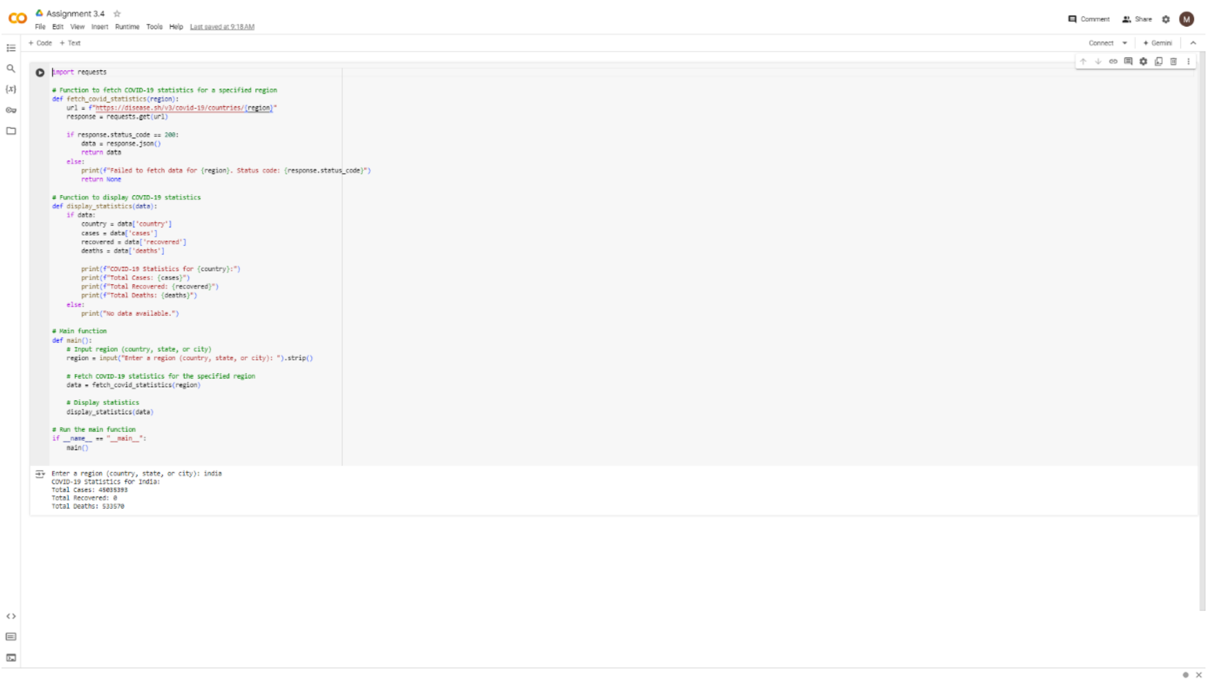
2.Implementation:

```python
import requests

# Function to fetch COVID-19 statistics for a specified region
def fetch_covid_statistics(region):
    url = f"https://disease.sh/v3/covid-19/countries/{region}"
    response = requests.get(url)

    if response.status_code == 200:
        data = response.json()
        return data
    else:
        print(f"Failed to fetch data for {region}. Status code:
{response.status_code}")
        return None

# Function to display COVID-19 statistics
def display_statistics(data):
    if data:
        country = data['country']
        cases = data['cases']
        recovered = data['recovered']
        deaths = data['deaths']

        print(f"COVID-19 Statistics for {country}:")
        print(f"Total Cases: {cases}")
        print(f"Total Recovered: {recovered}")
        print(f"Total Deaths: {deaths}")
    else:
        print("No data available.")

# Main function
def main():
    # Input region (country, state, or city)
    region = input("Enter a region (country, state, or city):
").strip()

    # Fetch COVID-19 statistics for the specified region
    data = fetch_covid_statistics(region)

    # Display statistics
    display_statistics(data)

# Run the main function
if __name__ == "__main__":
    main()
```

3.Sample Input & Output:

```
Enter a region (country, state, or city): india
COVID-19 Statistics for India:
Total Cases: 45035393
Total Recovered: 0
Total Deaths: 533570
```

4.User Input:


Assignment 3.4
File Edit View Insert Runtime Tools Help Last saved at 9:18 AM
+ Code + Text
Comment Share
Connect
Gemini

```python
import requests

# Function to fetch COVID-19 statistics for a specified region
def fetch_covid_statistics(region):
    url = f"https://disease.sh/v3/covid-19/countries/{region}"
    response = requests.get(url)

    if response.status_code == 200:
        data = response.json()
        return data
    else:
        print(f"Failed to fetch data for {region}. Status code: {response.status_code}")
        return None

# Function to display COVID-19 statistics
def display_statistics(data):
    if data:
        country = data['country']
        cases = data['cases']
        recovered = data['recovered']
        deaths = data['deaths']

        print(f"COVID-19 Statistics for {country}:")
        print(f"Total Cases: {cases}")
        print(f"Total Recovered: {recovered}")
        print(f"Total Deaths: {deaths}")
    else:
        print("No data available.")

# Main function
def main():
    # Input region (country, state, or city)
    region = input("Enter a region (country, state, or city): ").strip()

    # Fetch COVID-19 statistics for the specified region
    data = fetch_covid_statistics(region)

    # Display statistics
    display_statistics(data)

# Run the main function
if __name__ == "__main__":
    main()
```

```
Enter a region (country, state, or city): india
COVID-19 Statistics for India:
Total Cases: 45035393
Total Recovered: 0
Total Deaths: 533570
```

5.Documentation:

Stystem overview:

This Python script utilizes the `requests` library to fetch COVID-19 statistics from the Disease.sh API based on user-inputted regions (countries, states, or cities). Upon receiving the input, it constructs a URL to query the API for relevant data. If the API call is successful, it extracts and displays statistics such as total cases, recoveries, and deaths for the specified region. Error handling ensures that if the API call fails or the region is not found, appropriate messages are displayed to the user. This script provides a straightforward interface for retrieving and displaying current pandemic data on demand.

Interface:

This Python script serves as a simple interface for users to retrieve and display COVID-19 statistics by entering a region (country, state, or city). Upon execution, the program prompts the user to input the desired region. It then queries the Disease.sh API to fetch the latest statistics regarding total cases, recoveries, and deaths for that region. The fetched data is subsequently formatted and displayed in a clear manner. Error handling ensures that users are notified if the region entered is invalid or if there are issues with fetching data from the API, providing a seamless and informative interaction for tracking pandemic statistics.

Assumptions & Improvements:

This script assumes that users will input a valid region name (country, state, or city) that exists in the Disease.sh API database. It also assumes that the API will consistently return data in the expected JSON format when queried successfully. Furthermore, it assumes a stable internet connection for making API requests. First, implementing input validation to handle edge cases such as empty inputs, non-existent regions, or unexpected API responses would make the script more robust. Additionally, incorporating error handling for network issues or API rate limits could improve reliability. Caching previously fetched data locally could also reduce redundant API calls and improve performance, especially in scenarios with frequent data requests. Lastly, providing options for users to specify additional details or view historical data could enrich the functionality and utility of the script.