

## Assignment: Python Programming for DL

**Name:** Syed Sabira

**Register Number:** 192373044

**Department:** BE(Computer Science Engineering(Data Science))

**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.

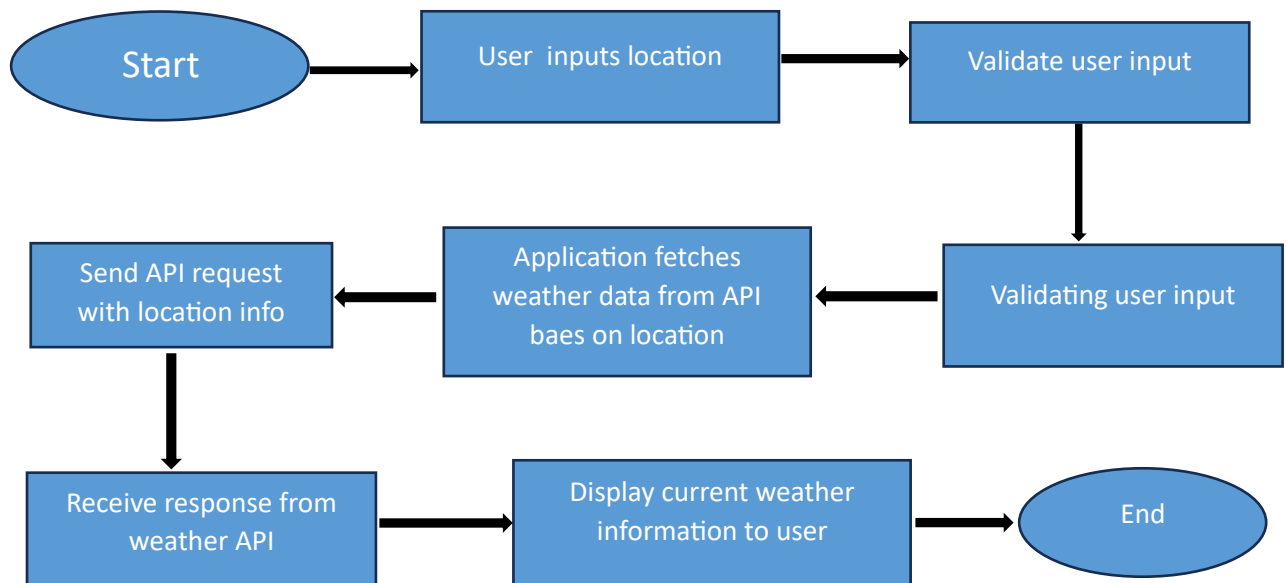
### Tasks:

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., Open Weather Map) 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:



## 2.IMPLEMENTATION:

```
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 the city name: Chennai

Enter location (city name or coordinates 'latitude,longitude'): Chennai

Weather Conditions:

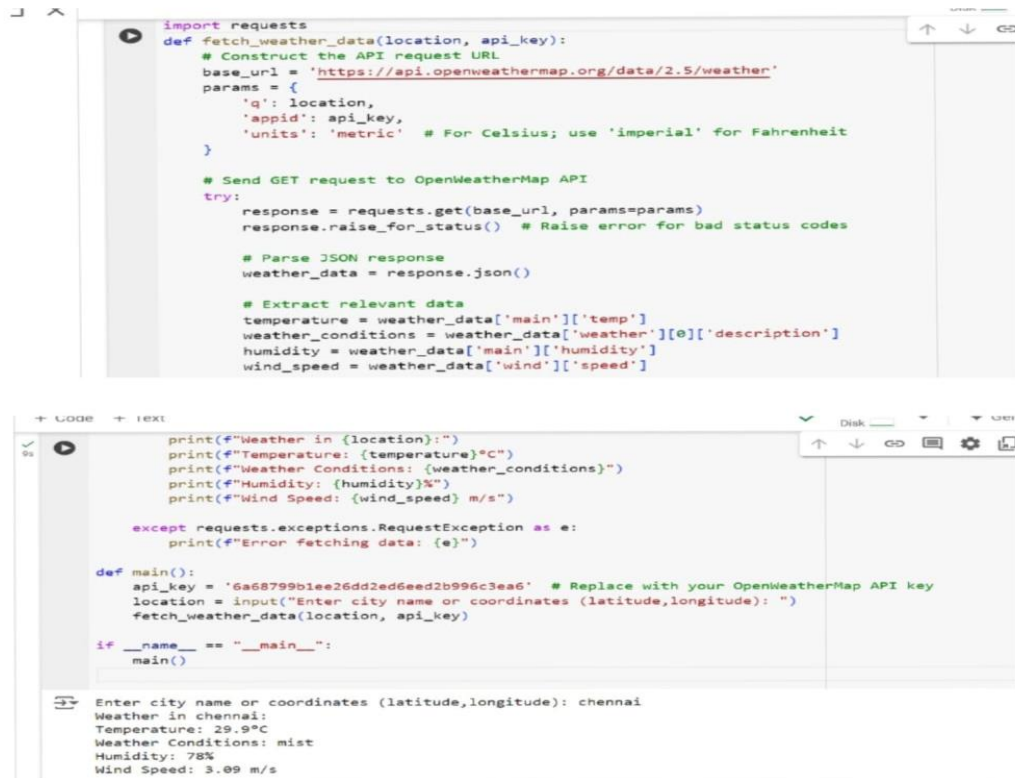
Temperature: 20.42 °C

Conditions: broken clouds

Humidity: 67%

Wind Speed: 4.12 m/s

## 4. User Input:



```
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']

    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()
```

```
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: 29.9°C  
Weather Conditions: mist  
Humidity: 78%  
Wind Speed: 3.09 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 Open Weather Map 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 Open Weather Map 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. Inventory Manager 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.

#### **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:

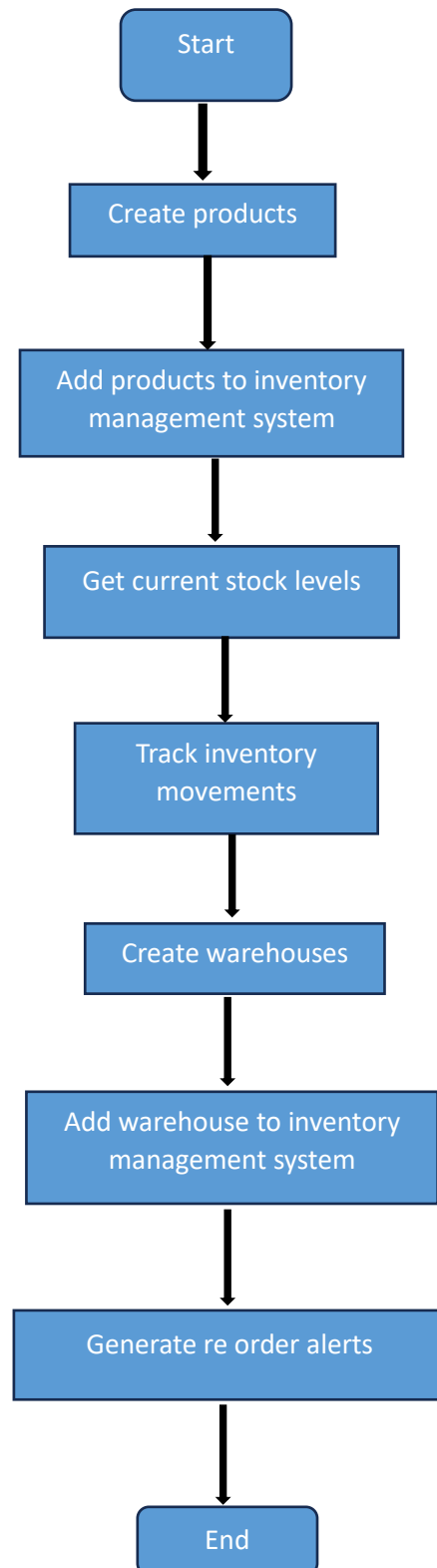
1. Model the inventory system: Define the structure of the inventory system, including products, warehouses, and current stock levels.
2. Implement an inventory tracking application: Develop a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain 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:





## 2.Implementation:

```
import pandas as pd
import random

class Product:
    def __init__(self, id, name, category, price,
reorder_threshold):
        self.id = id
        self.name = name
        self.category = category
        self.price = price
        self.reorder_threshold = reorder_threshold

class Warehouse:
    def __init__(self, name):
        self.name = name
        self.inventory = {}

    def add_product(self, product, quantity):
        if product.id in self.inventory:
            self.inventory[product.id] += quantity
        else:
            self.inventory[product.id] = quantity

    def get_stock_level(self, product_id):
        return self.inventory.get(product_id, 0)

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

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

    def add_warehouse(self, warehouse):
        self.warehouses[warehouse.name] = warehouse

    def track_inventory(self, warehouse_name, product_id,
quantity):
        if warehouse_name in self.warehouses:
            warehouse = self.warehouses[warehouse_name]
            if product_id in self.products:
```

```

        product = self.products[product_id]
        warehouse.add_product(product, quantity)
    else:
        print(f"Product with ID {product_id} not found.")
    else:
        print(f"Warehouse {warehouse_name} not found.")

def get_stock_level(self, warehouse_name, product_id):
    if warehouse_name in self.warehouses:
        warehouse = self.warehouses[warehouse_name]
        return warehouse.get_stock_level(product_id)
    else:
        print(f"Warehouse {warehouse_name} not found.")

def generate_reorder_alerts(self):
    alerts = []
    for warehouse_name, warehouse in self.warehouses.items():
        for product_id, current_stock in
warehouse.inventory.items():
            product = self.products[product_id]
            if current_stock < product.reorder_threshold:
                alerts.append(f"Alert: {product.name} in
{warehouse_name} needs reorder.")
    return alerts

def calculate_reorder_point(self, product_id, lead_time,
demand_forecast):
    average_daily_demand = sum(demand_forecast) /
len(demand_forecast)
    reorder_point = average_daily_demand * lead_time
    return reorder_point

def calculate_reorder_quantity(self, product_id,
reorder_point, lead_time):

    EOQ = 2 * reorder_point * (sum(demand_forecast) /
len(demand_forecast))
    safety_stock = 1.65 * (demand_forecast.std() * (lead_time
** 0.5))
    reorder_quantity = EOQ + safety_stock - current_inventory
    return reorder_quantity
if __name__ == "__main__":

    product1 = Product(1, "Keyboard", "Electronics", 29.99, 10)
    product2 = Product(2, "Mouse", "Electronics", 14.99, 15)
    warehouse1 = Warehouse("Main Warehouse")
    warehouse2 = Warehouse("Secondary Warehouse")
    inventory_system = InventoryManagementSystem()

```

```

inventory_system.add_product(product1)
inventory_system.add_product(product2)
inventory_system.add_warehouse(warehouse1)
inventory_system.add_warehouse(warehouse2)
inventory_system.track_inventory("Main Warehouse", 1, 50)
inventory_system.track_inventory("Main Warehouse", 2, 30)
inventory_system.track_inventory("Secondary Warehouse", 1,
20)

print("Current stock levels:")
print(f"Keyboard in Main Warehouse:
{inventory_system.get_stock_level('Main Warehouse', 1)} units")
print(f"Mouse in Main Warehouse:
{inventory_system.get_stock_level('Main Warehouse', 2)} units")
print(f"Keyboard in Secondary Warehouse:
{inventory_system.get_stock_level('Secondary Warehouse', 1)}
units")

alerts = inventory_system.generate_reorder_alerts()
print("\nReorder Alerts:")
for alert in alerts:
    print(alert)

```

### 3.Sample Input and Output:

Current stock levels:

Keyboard in Main Warehouse: 50 units

Mouse in Main Warehouse: 30 units

Keyboard in Secondary Warehouse: 20 units.

## 4. User Input:

```
import pandas as pd
import random

class Product:
    def __init__(self, id, name, category, price, reorder_threshold):
        self.id = id
        self.name = name
        self.category = category
        self.price = price
        self.reorder_threshold = reorder_threshold

class Warehouse:
    def __init__(self, name):
        self.name = name
        self.inventory = {}

    def add_product(self, product, quantity):
        if product.id in self.inventory:
            self.inventory[product.id] += quantity
        else:
            self.inventory[product.id] = quantity

    def get_stock_level(self, product_id):
        if product_id in self.inventory:
            return self.inventory[product_id]
        else:
            print(f"Warehouse {self.name} not found.")

    def generate_reorder_alerts(self):
        alerts = []
        for warehouse_name, warehouse in self.warehouses.items():
            for product_id, current_stock in warehouse.inventory.items():
                product = self.products[product_id]
                if current_stock < product.reorder_threshold:
                    alerts.append(f"Alert: {product.name} in {warehouse_name} needs reorder.")
        return alerts

    def calculate_reorder_point(self, product_id, lead_time, demand_forecast):
        EOQ = 2 * reorder_point * (sum(demand_forecast) / len(demand_forecast))
        safety_stock = 1.65 * (demand_forecast.std() * (lead_time ** 0.5))
        reorder_quantity = EOQ + safety_stock - current_inventory
        return reorder_quantity

if __name__ == "__main__":
    product1 = Product(1, "Keyboard", "Electronics", 29.99, 10)
    product2 = Product(2, "Mouse", "Electronics", 14.99, 15)
    warehouse1 = Warehouse("Main Warehouse")
    warehouse2 = Warehouse("Secondary Warehouse")
    inventory_system = InventoryManagementSystem()
    inventory_system.add_product(product1)
    inventory_system.add_product(product2)
    inventory_system.add_warehouse(warehouse1)
    inventory_system.add_warehouse(warehouse2)
    inventory_system.track_inventory("Main Warehouse", 1, 50)
    inventory_system.track_inventory("Main Warehouse", 2, 30)

    inventory_system.track_inventory("Secondary Warehouse", 1, 20)
    print("Current stock levels:")
    print(f"Keyboard in Main Warehouse: {inventory_system.get_stock_level('Main Warehouse', 1)} units")
    print(f"Mouse in Main Warehouse: {inventory_system.get_stock_level('Main Warehouse', 2)} units")
    print(f"Keyboard in Secondary Warehouse: {inventory_system.get_stock_level('Secondary Warehouse', 1)} units")

    alerts = inventory_system.generate_reorder_alerts()
    print("Reorder Alerts:")
    for alert in alerts:
        print(alert)

    print("Current stock levels:")
    print(f"Keyboard in Main Warehouse: 50 units")
    print(f"Mouse in Main Warehouse: 30 units")
    print(f"Keyboard in Secondary Warehouse: 20 units")

    print("Reorder Alerts:")
```

## 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.
- **Inventory Manager 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 Inventory Manager. 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 alert.

## 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.

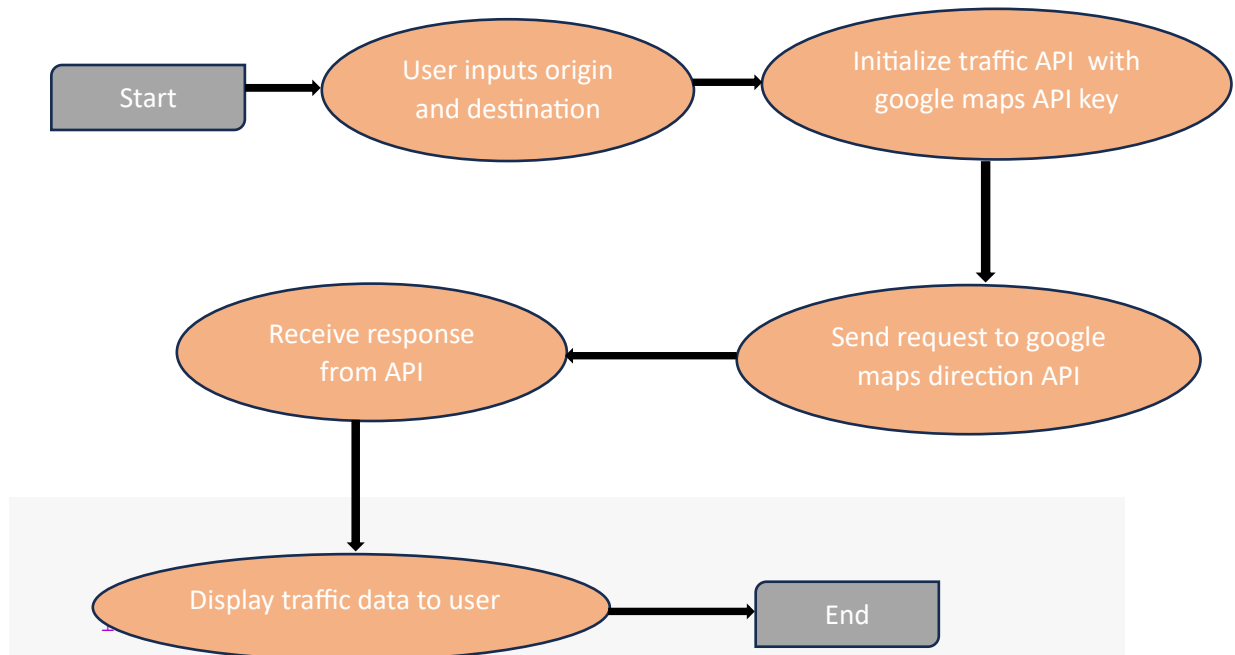
### Tasks:

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:



## 2.Implementation:

```
class TrafficAPI:
    def __init__(self, api_key):
        self.api_key = api_key
        self.base_url =
        "https://maps.googleapis.com/maps/api/directions/json"

    def get_traffic_data(self, origin, destination):
        params = {
            'origin': origin,
            'destination': destination,
            'key': self.api_key,
            'departure_time': 'now',
            'traffic_model': 'best_guess',
            'mode': 'driving'
        }
        try:
            response = requests.get(self.base_url, params=params)
            response.raise_for_status()

            traffic_data = response.json()
            return traffic_data

        except requests.exceptions.RequestException as e:
            print(f"Error fetching traffic data: {e}")
            return None
```

```

def display_traffic_data(traffic_data):
    if traffic_data:
        if traffic_data['status'] == 'OK':
            route = traffic_data['routes'][0]
            print("Route Summary:")
            print(f"Origin: {route['legs'][0]['start_address']}")
            print(f"Destination: {route['legs'][0]['end_address']}")
            print(f"Distance: {route['legs'][0]['distance']['text']}")
            print(f"Duration in current traffic: {route['legs'][0]['duration_in_traffic']['text']}")
            print("\nTraffic Conditions:")
            for step in route['legs'][0]['steps']:
                print(f"{step['html_instructions']}")
            if 'warnings' in route:
                print("\nIncidents or Delays:")
                for warning in route['warnings']:
                    print(f"- {warning}")
            else:
                print(f"Unable to fetch traffic data. Status: {traffic_data['status']}")
            else:
                print("No traffic data available.")

if __name__ == "__main__":
    api_key = 'YOUR_API_KEY_HERE'
    traffic_api = TrafficAPI(api_key)

    origin = input("Enter starting point (address or coordinates): ")
    destination = input("Enter destination (address or coordinates): ")

    traffic_data = traffic_api.get_traffic_data(origin, destination)

    if traffic_data:
        display_traffic_data(traffic_data)

```

### 3.Sample input & Output:

Enter starting point: new York, NY

Enter destination: Washington, DC

Route Summary:

Origin: New York, NY, USA

Destination: Washington, DC, USA

Distance: 225 mi

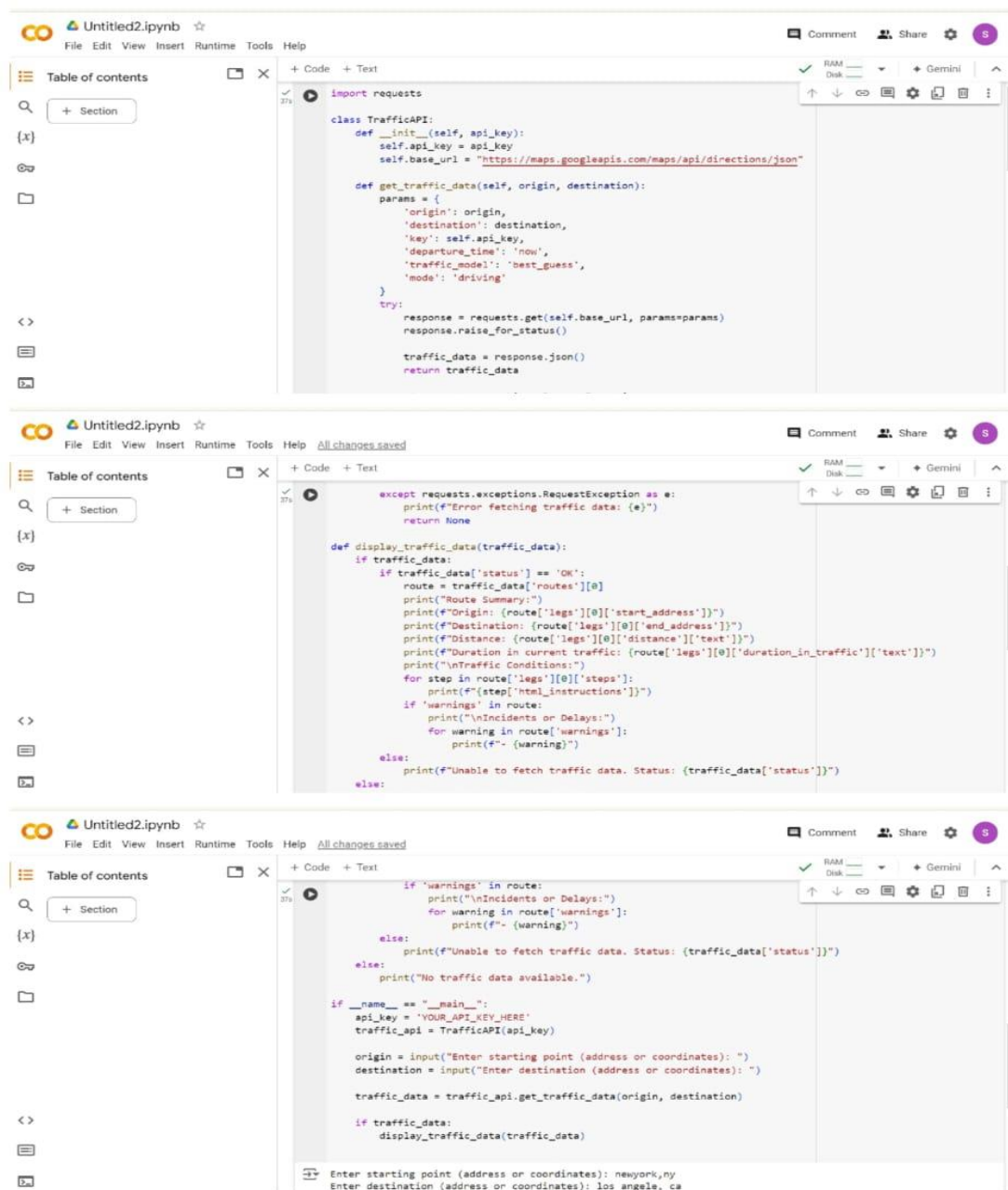
Duration in current traffic: 4 hours 30 mins

Traffic Conditions:

Continue on I-95 S. Take exit 2B for Interstate 395 S toward Washington.



## 4. User Input:



The image displays three sequential screenshots of a Jupyter Notebook titled 'Untitled2.ipynb', showing the development of a traffic API client. The interface includes a 'Table of contents' sidebar on the left and a 'Code' editor on the right.

**First Screenshot:** The code defines a `TrafficAPI` class. The `__init__` method takes an `api_key` and sets `self.base_url` to `"https://maps.googleapis.com/maps/api/directions/json"`. The `get_traffic_data` method takes `origin` and `destination` as arguments, constructs a `params` dictionary with `'origin'`, `'destination'`, `'key'` (using `self.api_key`), `'departure_time': 'now'`, `'traffic_model': 'best_guess'`, and `'mode': 'driving'`. It then makes a GET request to the base URL with these parameters, raises an exception on status, and returns the JSON data.

**Second Screenshot:** This screenshot adds an exception handling block to the `get_traffic_data` method. It catches `requests.exceptions.RequestException` and prints an error message. A `display_traffic_data` method is also added, which checks the `'status'` of the traffic data. If it's 'OK', it prints route summary, origin, destination, distance, and duration. It also iterates through steps to print HTML instructions. If there are warnings or incidents, it prints them. Otherwise, it prints the status.

**Third Screenshot:** This screenshot adds a main execution block. It prompts the user for an `api_key` (placeholder: 'YOUR\_API\_KEY\_HERE'). It then uses `input()` to get the `origin` and `destination`. The `TrafficAPI` class is instantiated, and `get_traffic_data` is called with the user input. Finally, `display_traffic_data` is called with the returned data. The output shows the user entering 'newyork,ny' for the starting point and 'los angele, ca' for the destination.

## 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 userfriendly 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

## 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.

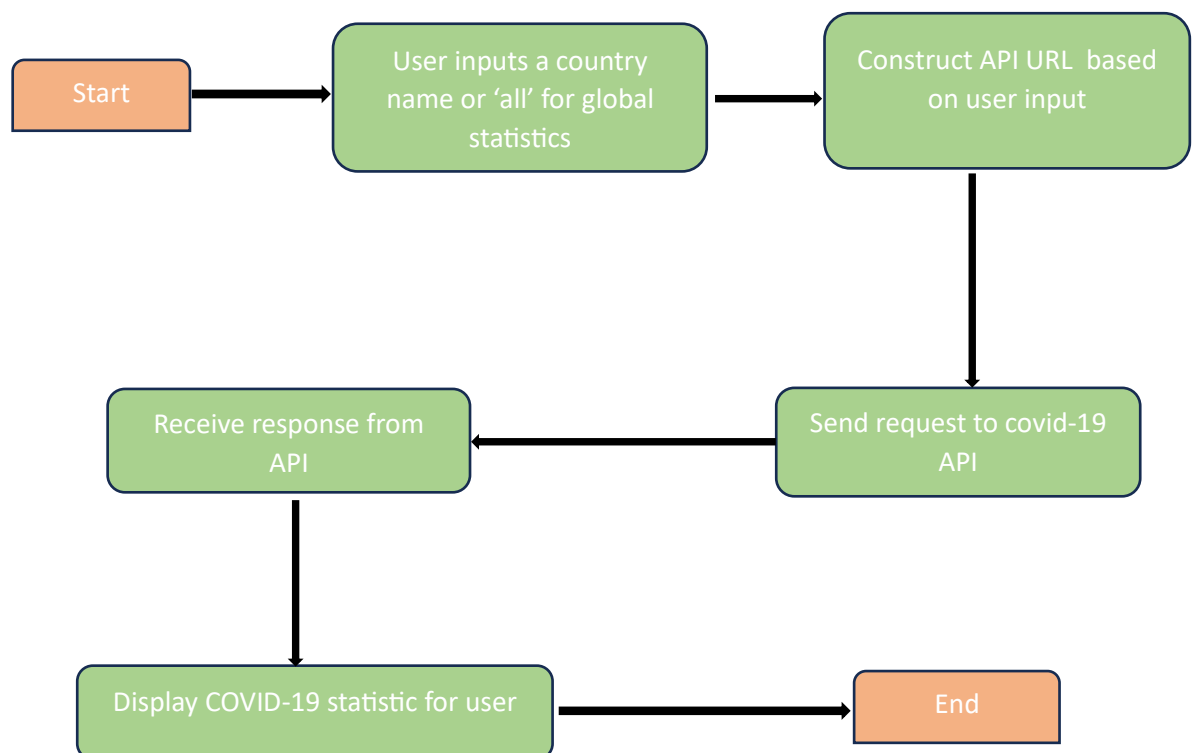
### Tasks:

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:



## 2.Implementation:

```
import requests

def fetch_covid_stats(region):
    base_url = f"https://disease.sh/v3/covid-19/countries/{region}"

    try:
        response = requests.get(base_url)
        response.raise_for_status() # Raise an exception for 4xx/5xx status codes

        data = response.json()

        if "message" in data:
            print(f"Error: {data['message']}")
        else:
            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}")

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

# Example usage:
if __name__ == "__main__":
    region = input("Enter a country name or 'all' for global
statistics: ")

    fetch_covid_stats(region)

```

### 3.Sample input & Output:

Enter a country name or 'all' for global statistics: india

COVID-19 Statistics for India:

Total Cases: 45035393

Total Recovered: 0

Total Deaths: 533570

## 4. User Input:

The image displays two sequential screenshots of a Google Colab notebook titled 'Untitled2.ipynb'. The browser tabs at the top include 'Google Classroom', 'Assignment - III', 'Untitled2.ipynb', 'api.openweathermap', 'Introducing ChatGPT', and 'ChatGPT'. The address bar shows a Google Drive link.

**Top Screenshot:** The notebook shows a code cell with the following Python code:

```
import requests

def fetch_covid_stats(region):
    base_url = f"https://disease.sh/v3/covid-19/countries/{region}"

    try:
        response = requests.get(base_url)
        response.raise_for_status()

        data = response.json()

        if "message" in data:
            print(f"Error: {data['message']}")
        else:
            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}")
```

The code is executed, and the output shows it completed at 12:07 PM.

**Bottom Screenshot:** The notebook shows the same code cell after further modifications. The code now includes an exception handler and a main block for user input:

```
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}")

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

# Example usage:
if __name__ == "__main__":
    region = input("Enter a country name or 'all' for global statistics: ")

    fetch_covid_stats(region)
```

The code is executed, and the output shows the user input 'india' and the resulting COVID-19 statistics for India:

```
Enter a country name or 'all' for global statistics: india
COVID-19 Statistics for India:
Total Cases: 45035393
Total Recovered: 0
Total Deaths: 533570
```

The code is executed, and the output shows it completed at 12:07 PM.

## 5.documentation:

### **System 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.