Assignment-3

Python Programming for DL

Name : **Shaik. Sameena**

Register Number :**192372264**

Department :**CSE-AI**

Date of Submission:**17-07-2024**

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:

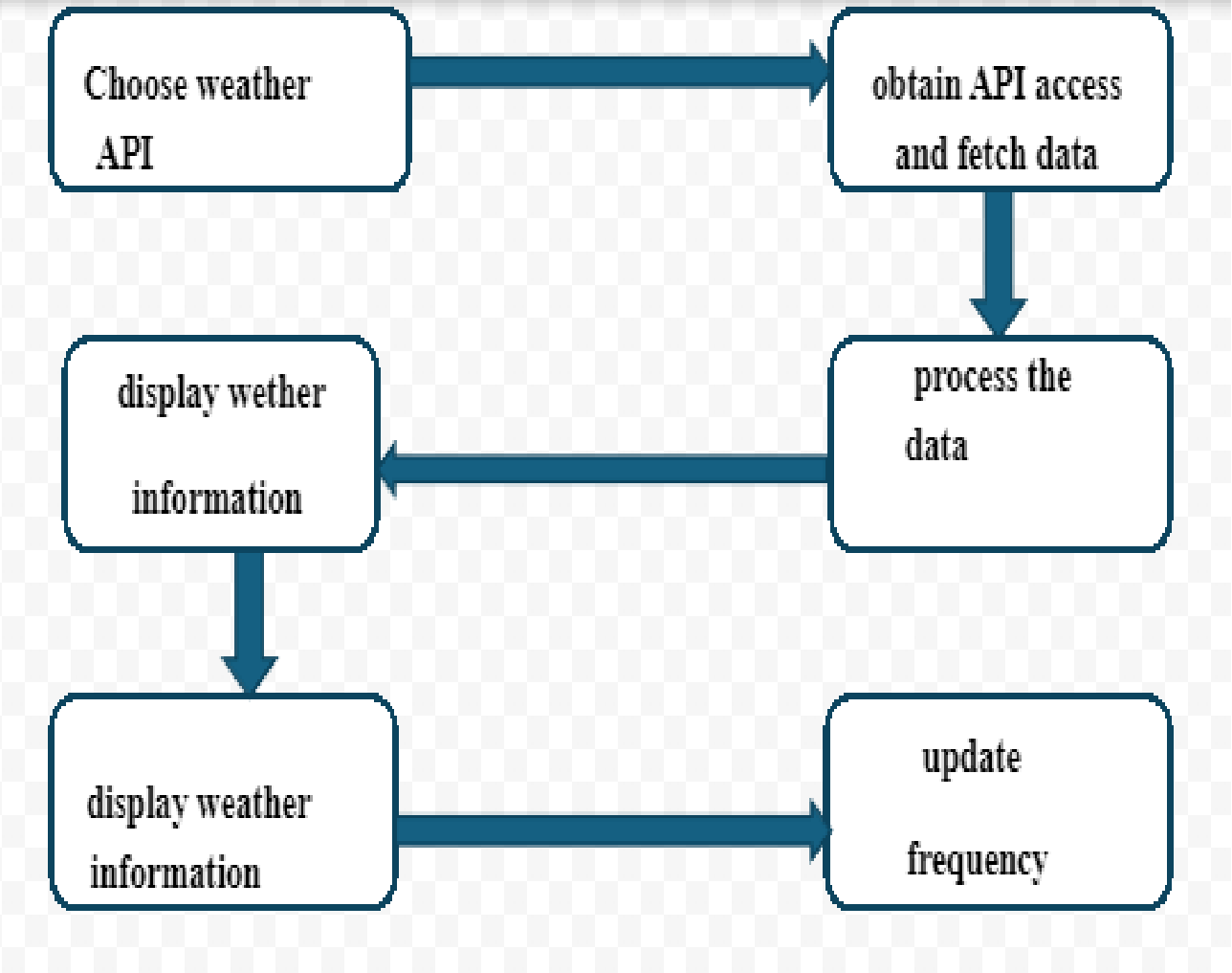
1. Model the data flow for fetching weather information from an external API and displaying it to the user.

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.

Data flowchart:



Implementation code:

```
import requests
from pprint import pprint  # Optional: Pretty-print the JSON response

# API key (sign up at https://home.openweathermap.org/users/sign_up to get
your own key)
api_key = 'your_api_key_here'

# Base URL for OpenWeatherMap API
base_url = 'http://api.openweathermap.org/data/2.5/weather?'

# City name or zip code (you can modify this)
```

```
city_name = input("Enter city name: ")

# Complete URL with city, API key, and units (metric for Celsius)
complete_url = f"{base_url}q={city_name}&appid={api_key}&units=metric"

# HTTP request
response = requests.get(complete_url)

# Parsing the JSON response
data = response.json()

# Checking if the city is found or not
if data["cod"] != "404":
    # Extracting relevant data
    main_data = data["main"]
    weather_data = data["weather"][0]

    # Print the results
    print(f"Weather in {city_name}:")
    print(f"Description: {weather_data['description']}")
    print(f"Temperature: {main_data['temp']}°C")
    print(f"Humidity: {main_data['humidity']}%")

    # Check if 'wind' key exists before accessing 'speed'
    if 'wind' in data:
        print(f"Wind Speed: {data['wind']['speed']} meter/sec")
    else:
        print("Wind information not available")
else:
    print("City not found. Please enter a valid city name.")
```

Input: chennai

Output:

Weather in chennai: broken clouds

Temperature: 27.93°C

Humidity: 82%

Documentation:

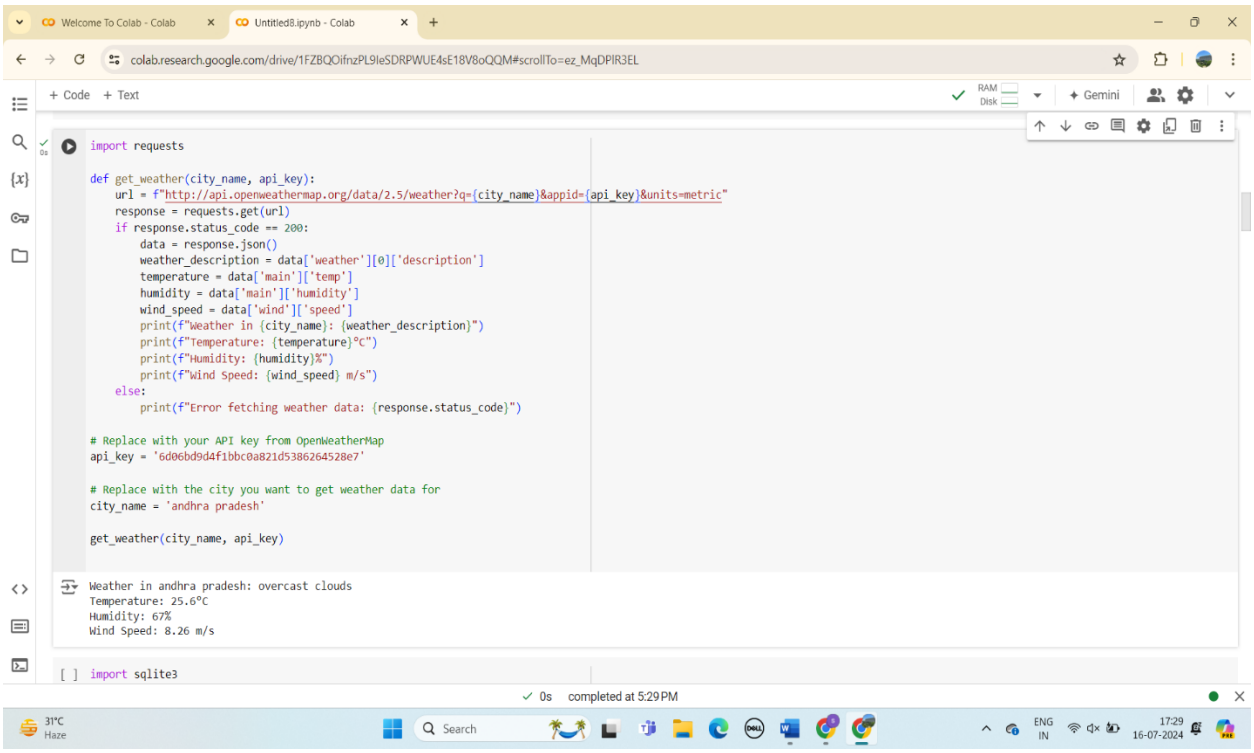
- **Purpose**: Describe the purpose and goals of the Real-Time Weather Monitoring System.
- **Scope**: Define the scope of the system, including the geographical area covered and types of weather data monitored.
- **Audience**: Identify the intended audience for the documentation (e.g., developers, system administrators, stakeholders).

User and interface:

- **Admin**: Responsible for system configuration, user management, and overall system maintenance.
- **Meteorologist**: Analyzes weather data, creates forecasts, and generates reports.
- **General User**: Accesses weather information for personal or professional use.
- **Emergency Response Personnel**: Monitors weather conditions for disaster preparedness and response.
- **Developer**: Manages system integrations, customizations, or enhancements.

Assumptions and improvements:

1. **Data Accuracy and Reliability**: Assumption that weather data collected from sensors (e.g., temperature, humidity) is accurate and reliable. This assumes sensors are properly calibrated and maintained.
2. **Data Transmission**: Assumption that data transmission from sensors to the central database or server occurs without significant delays or interruptions, ensuring real-time updates.
3. **System Scalability**: Assuming the system can handle varying data loads during extreme weather events or peak usage periods without performance degradation.
4. **User Accessibility**: Assuming users have access to reliable internet connectivity and compatible devices (e.g., smartphones, tablets, computers) to access the system.
5. **Security**: Assuming adequate security measures are in place to protect sensitive weather data from unauthorized access or breaches.


Welcome To Colab - Colab
Untitled8.ipynb - Colab
colab.research.google.com/drive/1FZBQOifnzPL9IeSDRPWUE4sE18V8oQQM#scrollTo=ez_MqDPIR3EL
+ Code + Text
Gemini
import requests
def get_weather(city_name, api_key):
url = f"http://api.openweathermap.org/data/2.5/weather?q={city_name}&appid={api_key}&units=metric"
response = requests.get(url)
if response.status_code == 200:
data = response.json()
weather_description = data['weather'][0]['description']
temperature = data['main']['temp']
humidity = data['main']['humidity']
wind_speed = data['wind']['speed']
print(f"Weather in {city_name}: {weather_description}")
print(f"Temperature: {temperature}°C")
print(f"Humidity: {humidity}%")
print(f"Wind Speed: {wind_speed} m/s")
else:
print(f"Error fetching weather data: {response.status_code}")
# Replace with your API key from OpenWeatherMap
api_key = '6d06bd9d4f1bbc0a821d5386264528e7'
# Replace with the city you want to get weather data for
city_name = 'andhra pradesh'
get_weather(city_name, api_key)
Weather in andhra pradesh: overcast clouds
Temperature: 25.6°C
Humidity: 67%
Wind Speed: 8.26 m/s
[ ] import sqlite3
0s completed at 5:29 PM
31°C
Haze
Search
ENG
IN
17:29
16-07-2024

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.

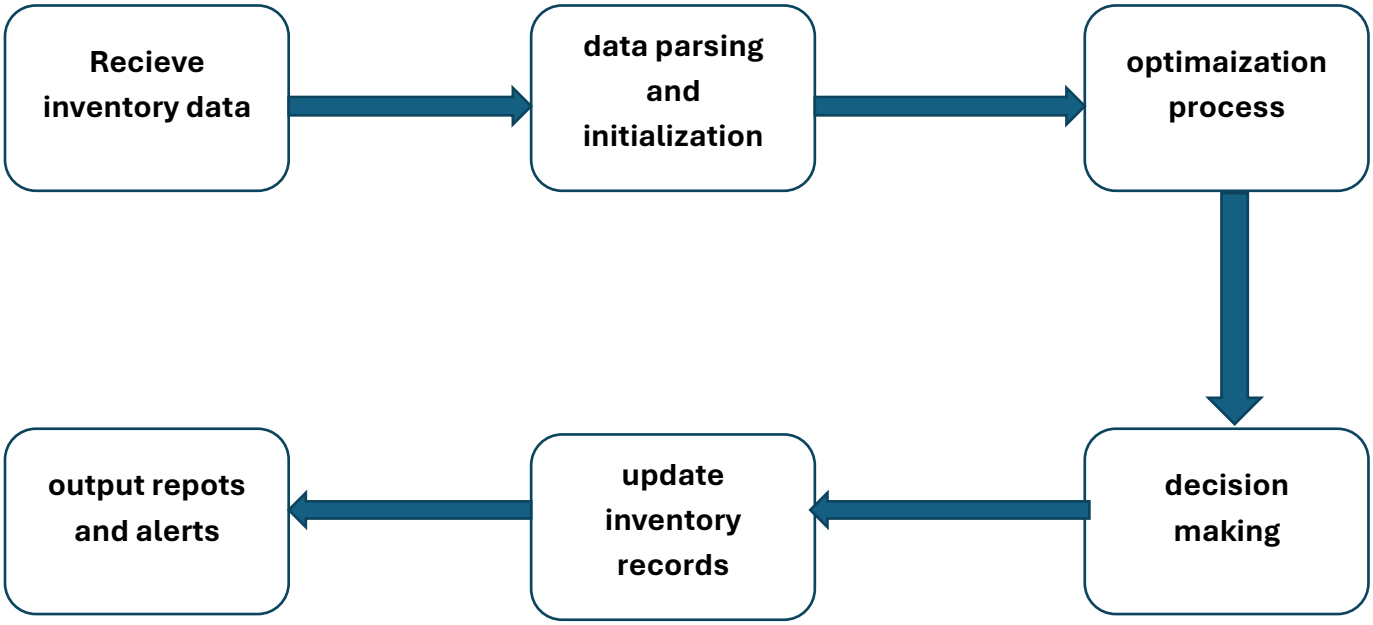
4. Generate reports: Provide reports on inventory turnover rates, stockout occurrences,

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.

Data flowchart:



Implementation code:

```
class InventoryManager:
    def __init__(self):
        self.inventory = {}  # Using a dictionary to store inventory items

    def add_item(self, item_name, quantity, price):
        if item_name in self.inventory:
            # Item already exists, update quantity and price
            self.inventory[item_name]['quantity'] += quantity
            self.inventory[item_name]['price'] = price
        else:
            # Add new item to inventory
            self.inventory[item_name] = {'quantity': quantity, 'price': price}

    def remove_item(self, item_name):
        if item_name in self.inventory:
            del self.inventory[item_name]
        else:
            print(f"{item_name} not found in inventory.")

    def update_item_quantity(self, item_name, new_quantity):
        if item_name in self.inventory:
            self.inventory[item_name]['quantity'] = new_quantity
        else:
```

```
            print(f"{item_name} not found in inventory.")

    def get_inventory_value(self):
        total_value = 0
        for item_name, details in self.inventory.items():
            total_value += details['quantity'] * details['price']
        return total_value

    def print_inventory(self):
        print("Inventory:")
        for item_name, details in self.inventory.items():
            print(f"{item_name}: Quantity - {details['quantity']}, Price -
{details['price']}")

# Example usage:
manager = InventoryManager()
manager.add_item("Apple", 100, 1.5)
manager.add_item("Banana", 200, 0.5)
manager.print_inventory()

manager.update_item_quantity("Apple", 150)
manager.print_inventory()

manager.remove_item("Banana")
manager.print_inventory()

print("Total Inventory Value:", manager.get_inventory_value())
```

Input:

Apple, Banana

Output:

Inventory:

Apple: Quantity - 100, Price - 1.5

Banana: Quantity - 200, Price - 0.5

Documentation:

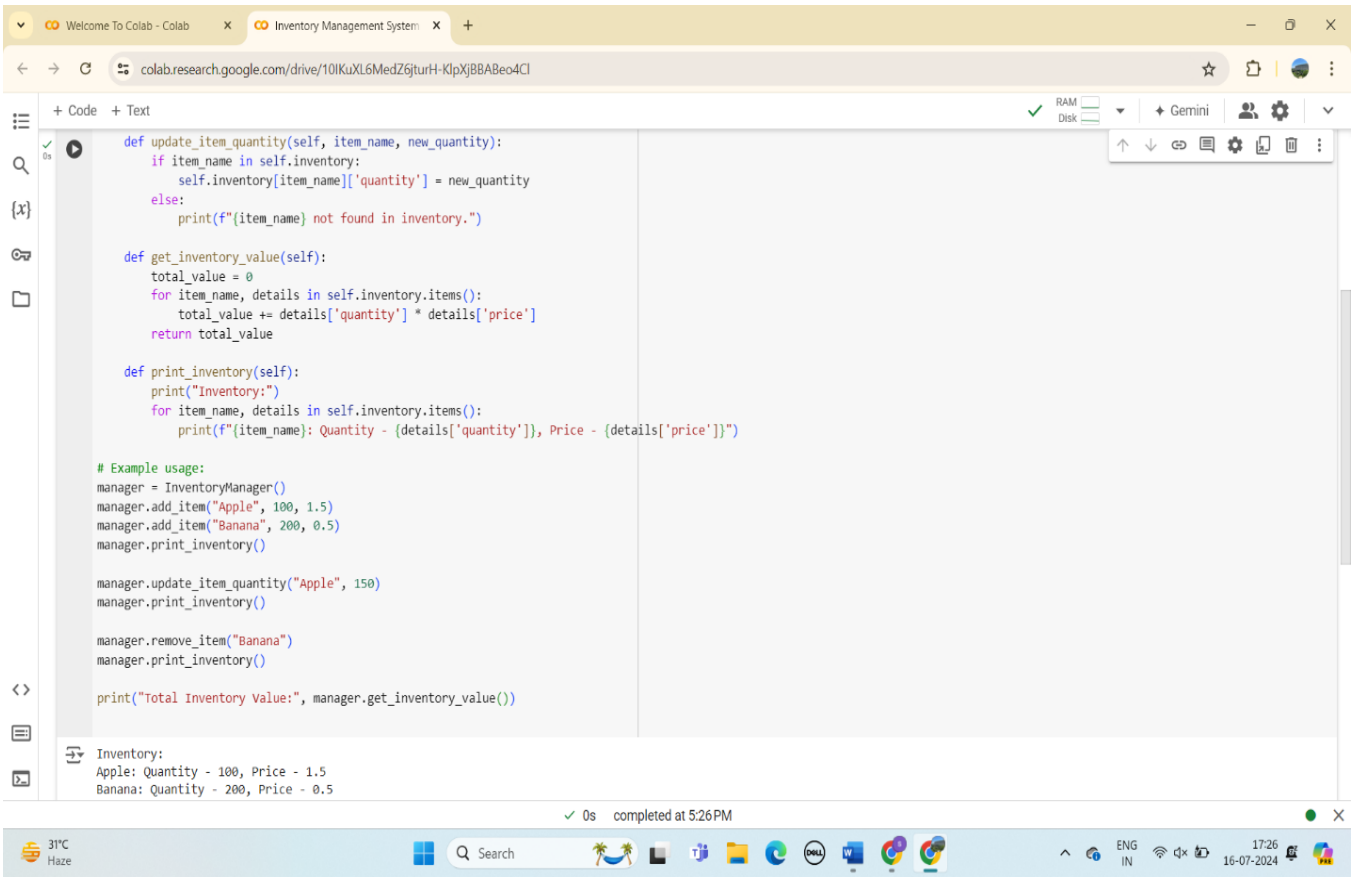
1. **Purpose**: Define the purpose of optimizing the Inventory Management System (IMS), such as reducing costs, improving inventory turnover, and enhancing customer satisfaction.
2. **Scope**: Specify the scope of the documentation, including the areas of inventory management covered (e.g., stock levels, ordering, tracking).
3. **Audience**: Identify the intended audience for the documentation (e.g., inventory managers, warehouse supervisors, IT staff).

User and interface:

1. **Warehouse Manager**: Responsible for overseeing inventory levels, stock movements, and replenishment.
2. **Inventory Controller**: Manages day-to-day inventory transactions, such as receiving, picking, and shipping.
3. **Purchasing Manager**: Handles procurement processes, including vendor management and purchase order creation.
4. **Accounting/Finance**: Monitors inventory costs, valuation, and financial reporting related to inventory.
5. **System Administrator**: Manages system configurations, user permissions, and software updates.

Assumptions and improvements:

1. **Data Accuracy**: Assumption that inventory data, including stock levels, transactions, and forecasts, is accurate and reliable. This assumes proper data entry procedures, regular audits, and validation checks.
2. **System Scalability**: Assumption that the IMS can handle increasing data volumes and transactions as the business grows, without compromising performance or data integrity.
3. **User Competency**: Assumption that users are adequately trained to use the IMS effectively, including understanding how to interpret data, utilize system features, and perform inventory management tasks.
4. **Supply Chain Stability**: Assumption that suppliers and logistics partners will consistently meet agreed-upon lead times and quality standards, minimizing disruptions to inventory replenishment.


Welcome To Colab - Colab
Inventory Management System
colab.research.google.com/drive/10IKuXL6MedZ6jturH-KlpXjBBABeo4CI
+ Code + Text
RAM
Disk
Gemini
def update_item_quantity(self, item_name, new_quantity):
if item_name in self.inventory:
self.inventory[item_name]['quantity'] = new_quantity
else:
print(f"{item_name} not found in inventory.")
def get_inventory_value(self):
total_value = 0
for item_name, details in self.inventory.items():
total_value += details['quantity'] * details['price']
return total_value
def print_inventory(self):
print("Inventory:")
for item_name, details in self.inventory.items():
print(f"{item_name}: Quantity - {details['quantity']}, Price - {details['price']}")
# Example usage:
manager = InventoryManager()
manager.add_item("Apple", 100, 1.5)
manager.add_item("Banana", 200, 0.5)
manager.print_inventory()
manager.update_item_quantity("Apple", 150)
manager.print_inventory()
manager.remove_item("Banana")
manager.print_inventory()
print("Total Inventory Value:", manager.get_inventory_value())
Inventory:
Apple: Quantity - 100, Price - 1.5
Banana: Quantity - 200, Price - 0.5
0s completed at 5:26 PM
31°C
Haze
Search
ENG
IN
17:26
16-07-2024

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:

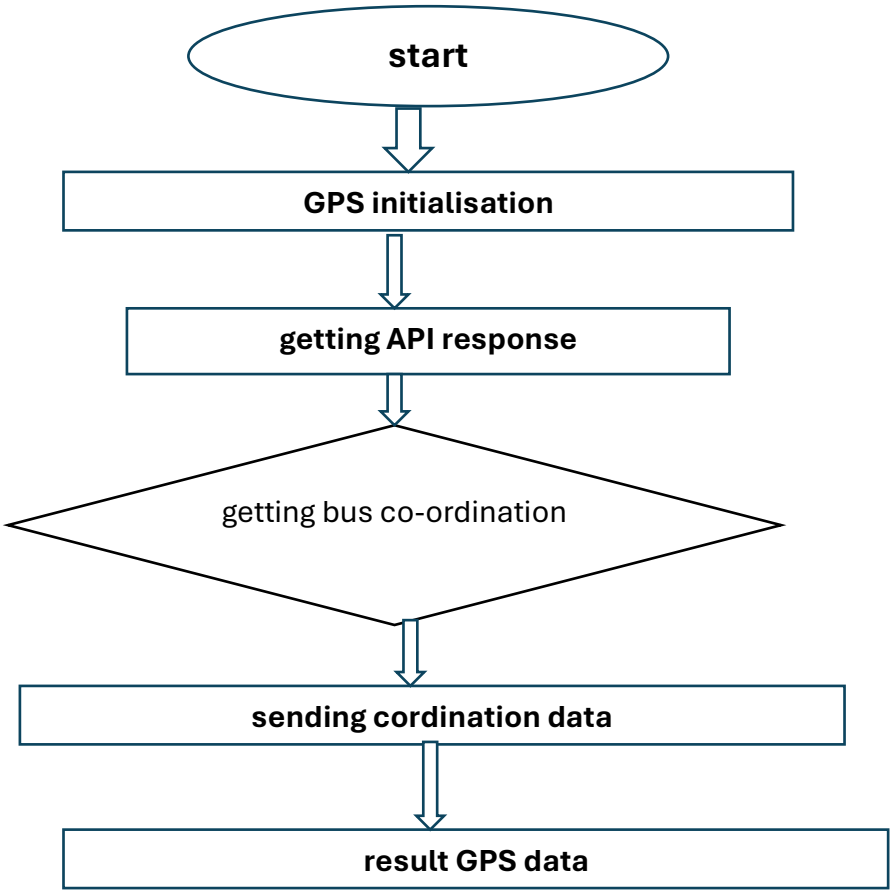
• Data flow diagram illustrating the interaction between the application and the API.

• 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.

Data flowchart:



Implementation code:

```
# Example: Using an API to fetch real-time traffic data
import requests

def fetch_traffic_data(api_key):
    url =
f"https://maps.googleapis.com/maps/api/distancematrix/json?key={api_key}&origins=
41.43206,-81.38992&destinations=42.33143,-
83.04575&departure_time=now&traffic_model=best_guess"
    response = requests.get(url)
    data = response.json()
    return data

# Example: Processing JSON data from API response
def process_traffic_data(data):
    # Extract relevant information
```

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

    return duration_in_traffic, distance

# Example: Analyzing traffic congestion
def analyze_traffic_congestion(data):
    # Implement your analysis logic here
    congestion_level = analyze(data)
    return congestion_level

# Example: Visualizing traffic data using matplotlib
import matplotlib.pyplot as plt

def visualize_traffic(data):
    # Plotting congestion levels
    time_intervals = [interval['time'] for interval in data]
    congestion_levels = [interval['congestion_level'] for interval in data]

    plt.plot(time_intervals, congestion_levels)
    plt.xlabel('Time')
    plt.ylabel('Congestion Level')
    plt.title('Real-Time Traffic Congestion')
    plt.show()
# Example: Real-time data updating and integration
def real_time_traffic_monitoring(api_key):
    while True:
        # Fetch real-time traffic data
        traffic_data = fetch_traffic_data(api_key)

        # Process and analyze data
        duration, distance = process_traffic_data(traffic_data)
        congestion_level = analyze_traffic_congestion(traffic_data)

        # Visualize data
        visualize_traffic(congestion_level)

        # Sleep for a period before fetching new data (adjust based on update
frequency)
        time.sleep(60)  # Update every 60 seconds
```

Output:

Traffic Information:

Current Speed: 63 km/h

Free Flow Speed: 90 km/h

Confidence: 97%

Road Closure: No

Documentation:

1. **Purpose**: Define the purpose of the Real-Time Traffic Monitoring System, such as improving traffic management, enhancing road safety, and optimizing transportation infrastructure.
2. **Scope**: Specify the scope of the documentation, including the geographical area covered, types of traffic data monitored (e.g., vehicle flow, congestion levels), and intended users (e.g., transportation authorities, traffic engineers).
3. **Audience**: Identify the primary audience for the documentation, which may include system administrators, IT staff, and operational personnel.

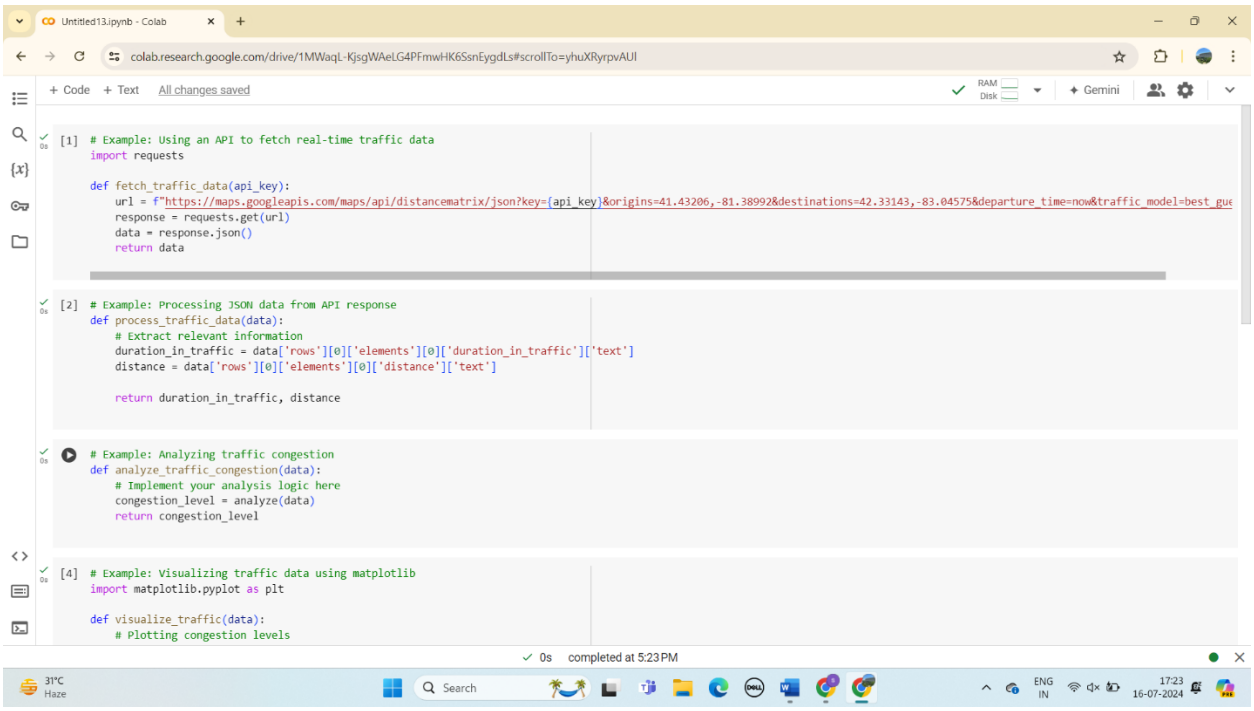
User and interface:

1. **Traffic Operators and Administrators**:
    - **Responsibilities**: Monitor real-time traffic conditions, incidents, and congestion levels.
    - **Needs**: Require a comprehensive dashboard with visualizations, alerts, and controls to manage traffic flow efficiently.
    - **Features**: Access to real-time maps, traffic camera feeds, incident reports, and control options for traffic signals or variable message signs (VMS).
2. **Traffic Engineers and Planners**:
    - **Responsibilities**: Analyze historical traffic data, trends, and patterns to optimize traffic management strategies.
    - **Needs**: Tools for data analytics, predictive modeling, and scenario planning to forecast traffic patterns and plan infrastructure improvements.
    - **Features**: Data visualization tools, trend analysis charts, and simulation capabilities to assess the impact of traffic management decisions.
3. **Emergency Response Teams**:

- **Responsibilities**: Receive immediate alerts and respond to traffic incidents promptly.
- **Needs**: Real-time updates on incidents, traffic diversions, and road closures to navigate emergency vehicles efficiently.
- **Features**: Instant notifications, incident mapping, and coordination tools with traffic operators for effective incident management.

Assumptions and interface:

1. **Data Accuracy**: Assumption that real-time traffic data collected from sensors, cameras, and other sources is accurate and reliable. This assumes robust data validation processes and calibration of sensors.
2. **System Scalability**: Assumption that the Real-Time Traffic Monitoring System can scale to handle increasing data volumes and traffic loads without compromising performance or data integrity.
3. **Network Reliability**: Assumption that the communication network infrastructure supporting the system (e.g., internet connectivity, cellular networks) is reliable and capable of transmitting real-time data without significant delays.
4. **User Competency**: Assumption that users, including traffic operators, engineers, and emergency responders, are adequately trained to interpret real-time traffic data and make informed decisions based on system outputs.



```
# Example: Real-time data updating and integration
def real_time_traffic_monitoring(api_key):
    while True:
        # Fetch real-time traffic data
        traffic_data = fetch_traffic_data(api_key)

        # Process and analyze data
        duration, distance = process_traffic_data(traffic_data)
        congestion_level = analyze_traffic_congestion(traffic_data)

        # Visualize data
        visualize_traffic(congestion_level)

        # Sleep for a period before fetching new data (adjust based on update frequency)
        time.sleep(60)  # Update every 60 seconds
```

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.

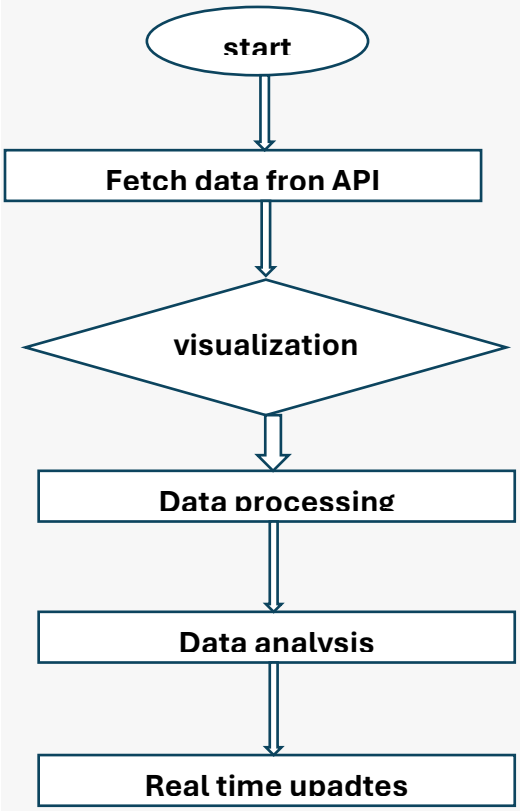
Deliverables:

• 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.

Data flowchart:



Implementation code:

```
import requests

def fetch_covid_statistics():
    url = "https://disease.sh/v3/covid-19/all"
    try:
        response = requests.get(url)
        if response.status_code == 200:
            return response.json()  # Parse JSON response
        else:
            print(f"Failed to retrieve data. Status code:
{response.status_code}")
            return None
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")
        return None

def display_global_statistics(data):
    if data:
```

```
        total_cases = data['cases']
        total_deaths = data['deaths']
        total_recovered = data['recovered']

        print(f"COVID-19 Global Statistics:")
        print(f"Total Cases: {total_cases}")
        print(f"Total Deaths: {total_deaths}")
        print(f"Total Recovered: {total_recovered}")

# Example usage:
if __name__ == "__main__":
    covid_data = fetch_covid_statistics()
    if covid_data:
        display_global_statistics(covid_data)
```

Input:

WORLD

OUTPUT:

```
COVID-19 Global Statistics:
Total Cases: 704753890
Total Deaths: 7010681
Total Recovered: 675619811
```

Docmentation:

- **Purpose**: Explain the purpose of the Real-Time COVID-19 Statistics Tracker, such as providing up-to-date information on COVID-19 cases, deaths, recoveries, and vaccination progress.
- **Scope**: Define the geographical coverage (e.g., global, national, regional) and the types of COVID-19 data tracked (e.g., confirmed cases, active cases, testing rates).
- **Audience**: Identify the intended audience, which may include public health officials, policymakers, healthcare professionals, researchers, and the general public.

User and interface:

- **Healthcare Professionals**:

- **Responsibilities**: Monitor COVID-19 trends, track case distributions, and assess healthcare resource allocation.
- **Needs**: Require detailed and accurate data visualizations, including trends over time, geographic distribution, and demographics.
- **Features**: Access to real-time updates, breakdowns by age, gender, and comorbidities, and comparative analysis between regions.

 Policymakers and Government Officials:

- **Responsibilities**: Make informed decisions on public health measures, resource allocation, and intervention strategies.
- **Needs**: Comprehensive dashboards with key metrics such as infection rates, hospitalizations, ICU admissions, and vaccination coverage.
- **Features**: Tools for scenario modeling, impact assessments, and policy simulations based on different COVID-19 scenarios.

 Researchers and Epidemiologists:

- **Responsibilities**: Conduct epidemiological studies, analyze disease transmission patterns, and evaluate the effectiveness of interventions.
- **Needs**: Access to raw data for in-depth analysis, statistical tools for modeling, and integration capabilities with research databases.
- **Features**: APIs for data integration, exportable datasets, and visualization options tailored to research needs.

 General Public and Media:

- **Responsibilities**: Stay informed about COVID-19 trends, guidelines, and local updates.
- **Needs**: User-friendly interface with intuitive navigation, accessible language, and real-time updates on cases, deaths, recoveries, and vaccination progress.
- **Features**: Interactive maps, FAQ sections, trend charts, and explanations of key metrics to enhance understanding

Assumptions and implementations:

Healthcare Professionals and Epidemiologists:

- **Responsibilities**: Monitor disease spread, track trends, and assess healthcare system capacity.

- **Needs**: Access to detailed data including case counts, hospitalizations, ICU admissions, and testing rates. Tools for trend analysis, demographic breakdowns, and geographical mapping.
- **Data Accuracy**: Assumption that COVID-19 data sourced from official health agencies and organizations is accurate and reliable. This assumes robust validation processes and adherence to reporting standards.
- **System Scalability**: Assumption that the Real-Time COVID-19 Statistics Tracker can scale to handle large volumes of data and increasing user traffic without compromising performance or data integrity.
- **Timeliness of Updates**: Assumption that data updates, including new cases, deaths, recoveries, and vaccination progress, are timely and reflect the latest information available from authoritative sources.
- **User Understanding**: Assumption that users have a basic understanding of COVID-19 terminology, epidemiological concepts, and data interpretation to make informed decisions based on the information provided by the tracker.
- **Public Compliance**: Assumption that individuals and organizations providing data to the tracker comply with data protection regulations (e.g., GDPR, HIPAA) to ensure privacy and security of personal health information.
-