DAREDDY MEGHANA

192372276

CSE-AI

PYTHON API PROGRAMS DOCUMENTATION

DATE:16/07/2024

1.Real-Time Weather Monitoring System

Scenario:

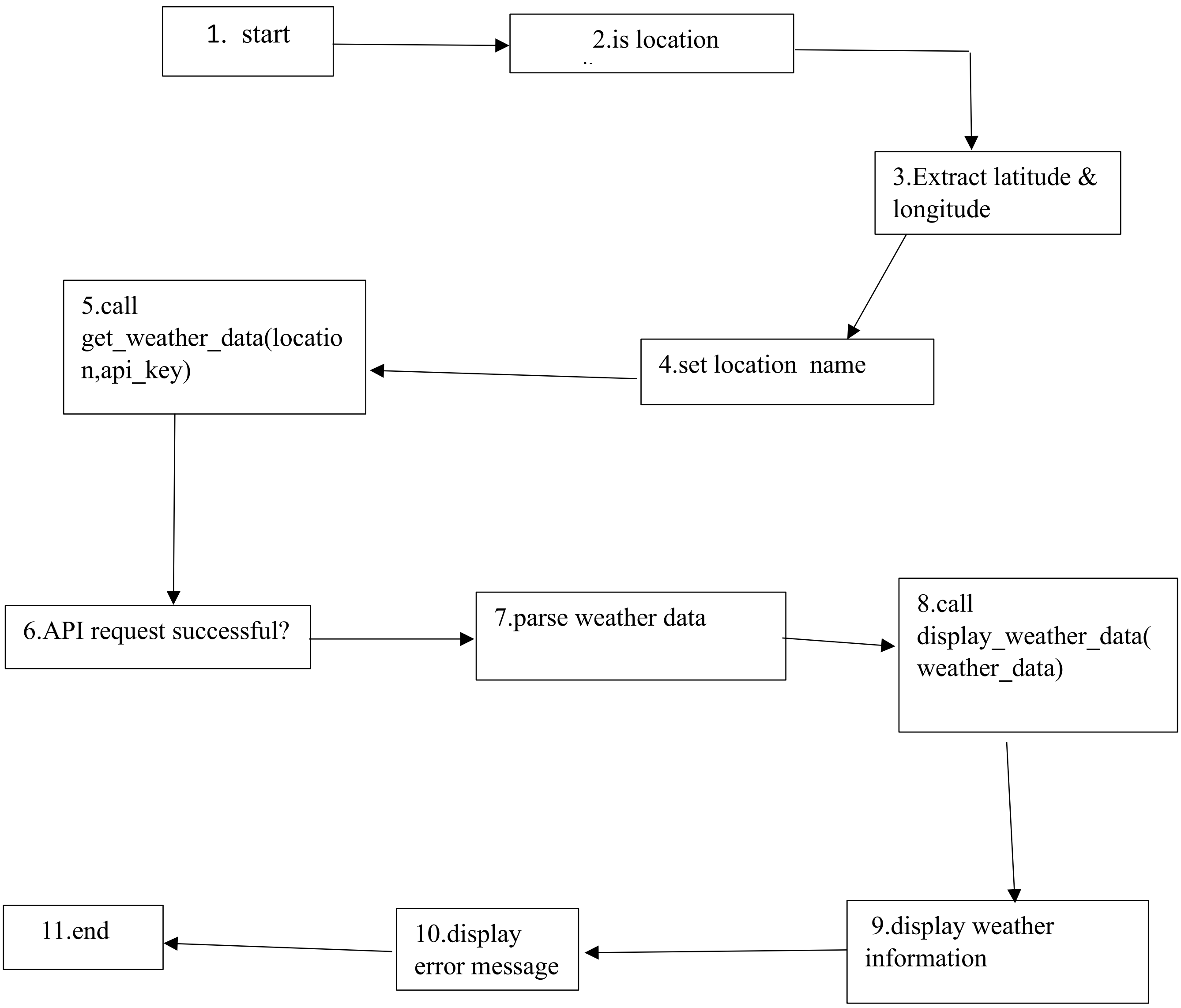
You are developing a real-time weather monitoring system for a weather forecasting company.
The system needs to fetch and display weather data for a specified location.

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., 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.

Data flow diagram:



Implementation:

```
import requests

def get_weather_data(location, api_key):
    base_url =
"https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid"
    params = {
        'q': location,
        'appid': api_key,
        'units': 'metric'
```

```
    }
    response = requests.get(base_url, params=params)
    return response.json()

def display_weather_data(weather_data):
    try:
        location = weather_data['name']
        temperature = weather_data['main']['temp']
        weather_conditions = weather_data['weather'][0]['description']
        humidity = weather_data['main']['humidity']
        wind_speed = weather_data['wind']['speed']

        print(f"Location: {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 KeyError:
        print("Error in fetching weather data. Please check the location input.")

def main():
    api_key = '9eef1b6f1d45139187997c1dc7cae216'
    location = input("Enter the city name or coordinates (latitude,longitude): ")
    weather_data = get_weather_data(location, api_key)
    display_weather_data(weather_data)

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

Displaying data:

Input:

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

Output:

```
Location: Raipur
Temperature: 33.08°C
Humidity: 66%
Wind Speed: 2.06 m/s
```

```python
        print(f"Location: {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 KeyError:
        print("Error in fetching weather data. Please check the location input.")

def main():
    api_key = '9eef1b6f1d45139187997c1dc7cae216'
    location = input("Enter the city name or coordinates (latitude,longitude): ")
    weather_data = get_weather_data(location, api_key)
    display_weather_data(weather_data)

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

```
Enter the city name or coordinates (latitude,longitude): raipur
Location: Raipur
Temperature: 33.08°C
Weather Conditions: haze
Humidity: 66%
Wind Speed: 2.06 m/s
```

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).

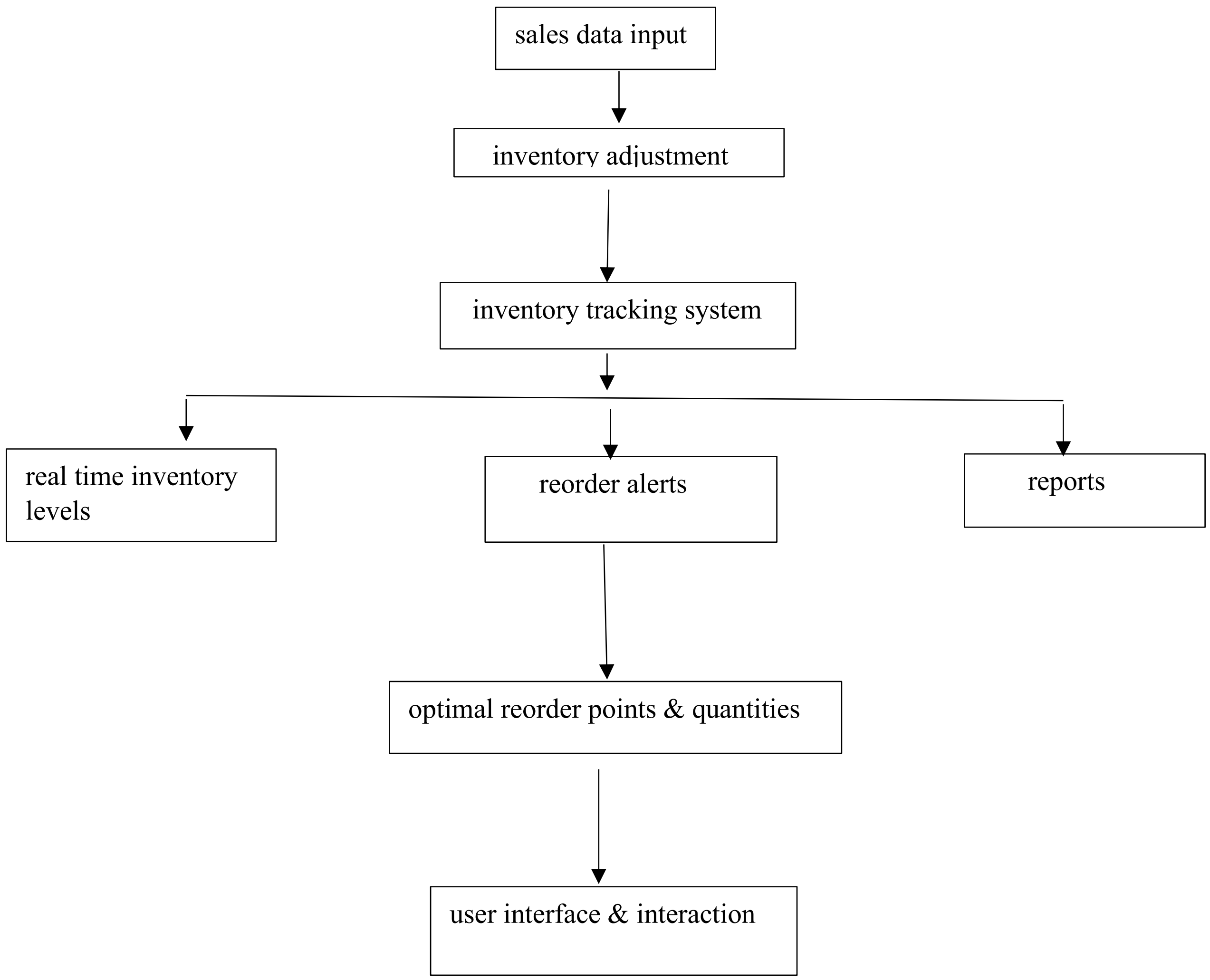
• Pseudocode and Implementation: Provide pseudocode and actual code demonstrating
how inventory levels are tracked, reorder points are calculated, and reports are
generated.

• 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.

Data flow diagram:



Implementation:

```
import math
from datetime import datetime, timedelta

class Product:
    def __init__(self, product_id, name, reorder_level, cost_price):
        self.product_id = product_id
        self.name = name
        self.reorder_level = reorder_level
        self.cost_price = cost_price
        self.stock = 0
        self.sales_history = []

    def update_stock(self, quantity):
        self.stock += quantity
```

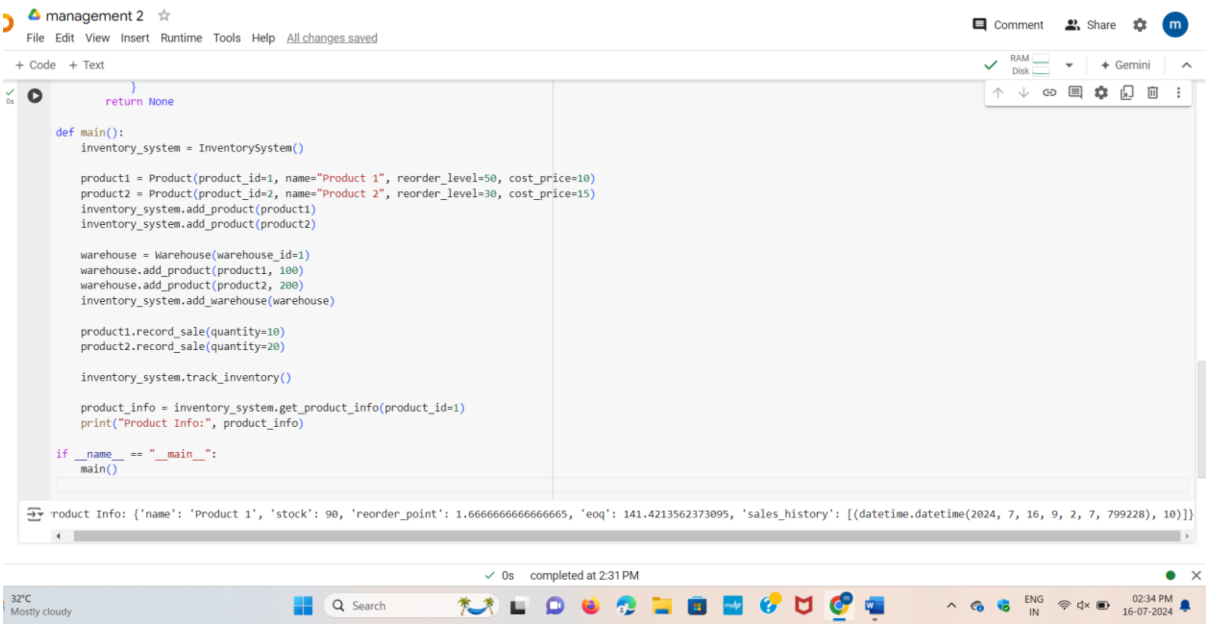
```
    def record_sale(self, quantity, sale_date=None):
        if sale_date is None:
            sale_date = datetime.now()
        self.sales_history.append((sale_date, quantity))
        self.stock -= quantity

    def avg_daily_demand(self, days=30):
        end_date = datetime.now()
        start_date = end_date - timedelta(days=days)
        total_sales = sum(q for date, q in self.sales_history if start_date <= date <= end_date)
        return total_sales / days

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

    def add_product(self, product, quantity):
        if product.product_id not in self.inventory:
            self.inventory[product.product_id] = 0
        self.inventory[product.product_id] += quantity
        product.update_stock(quantity)

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

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

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

    def track_inventory(self):
        for product in self.products.values():
            if product.stock < product.reorder_level:
                self.alert_reorder(product)

    def alert_reorder(self, product):
        print(f"Reorder alert for product: {product.name}")

    def reorder_point(self, product, lead_time_days):
        return product.avg_daily_demand() * lead_time_days

    def eoq(self, demand, order_cost, holding_cost):
        return math.sqrt((2 * demand * order_cost) / holding_cost)
```

```
    def get_product_info(self, product_id):
        if product_id in self.products:
            product = self.products[product_id]
            reorder_point = self.reorder_point(product, lead_time_days=5)
            eoq = self.eoq(demand=1000, order_cost=50, holding_cost=5)
            return {
                'name': product.name,
                'stock': product.stock,
                'reorder_point': reorder_point,
                'eoq': eoq,
                'sales_history': product.sales_history
            }
        return None

def main():
    inventory_system = InventorySystem()

    product1 = Product(product_id=1, name="Product 1", reorder_level=50, cost_price=10)
    product2 = Product(product_id=2, name="Product 2", reorder_level=30, cost_price=15)
    inventory_system.add_product(product1)
    inventory_system.add_product(product2)

    warehouse = Warehouse(warehouse_id=1)
    warehouse.add_product(product1, 100)
    warehouse.add_product(product2, 200)
    inventory_system.add_warehouse(warehouse)

    product1.record_sale(quantity=10)
    product2.record_sale(quantity=20)

    inventory_system.track_inventory()

    product_info = inventory_system.get_product_info(product_id=1)
    print("Product Info:", product_info)

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

Displaying data:

Output:

Product Info: {'name': 'Product 1', 'stock': 90, 'reorder_point': 1.6666666666666665, 'eoq': 141.4213562373095, 'sales_history': [(datetime.datetime(2024, 7, 16, 9, 2, 7, 799228), 10)]}



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:

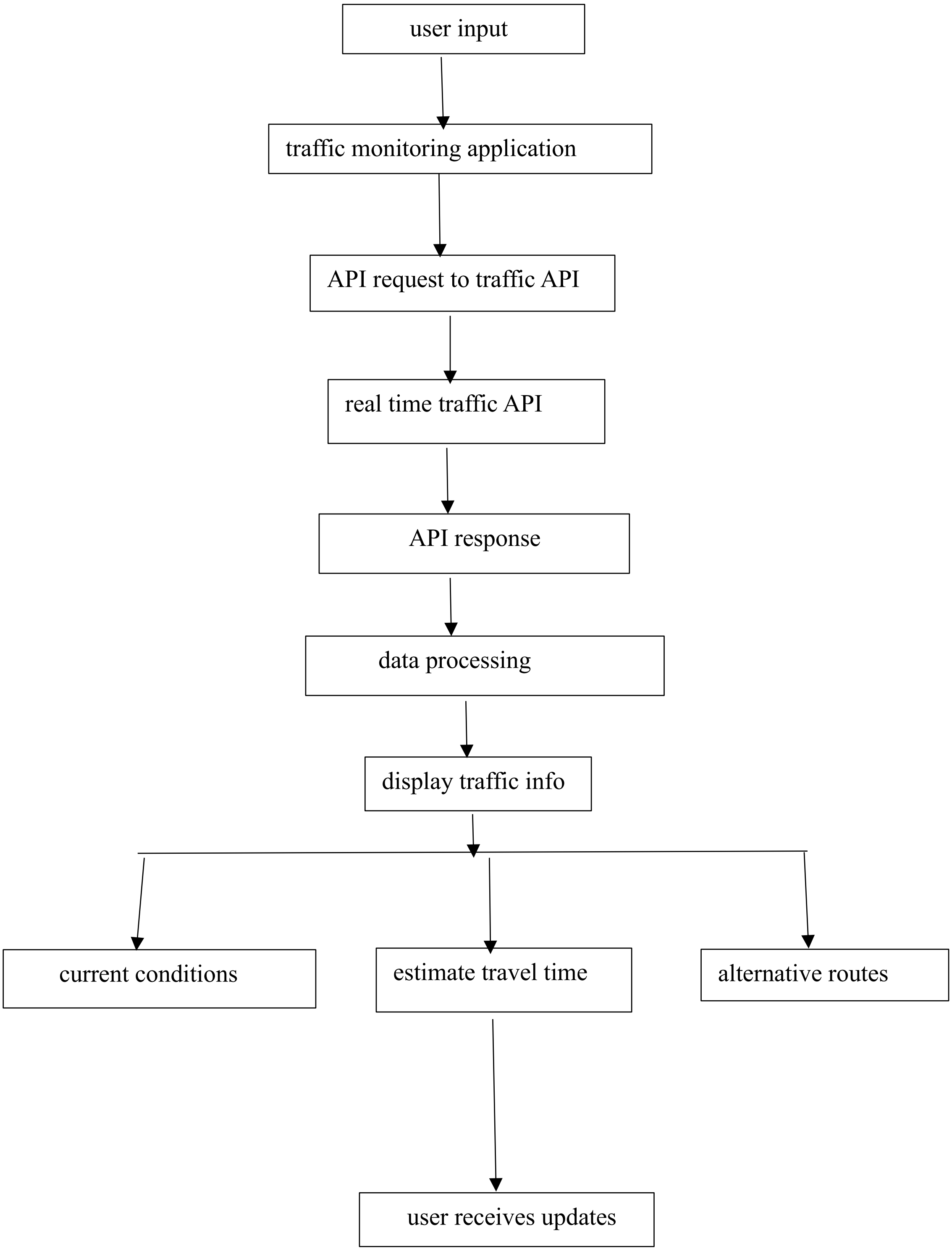
• Data flow diagram illustrating the interaction between the application and the API.

• 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 flow diagram:

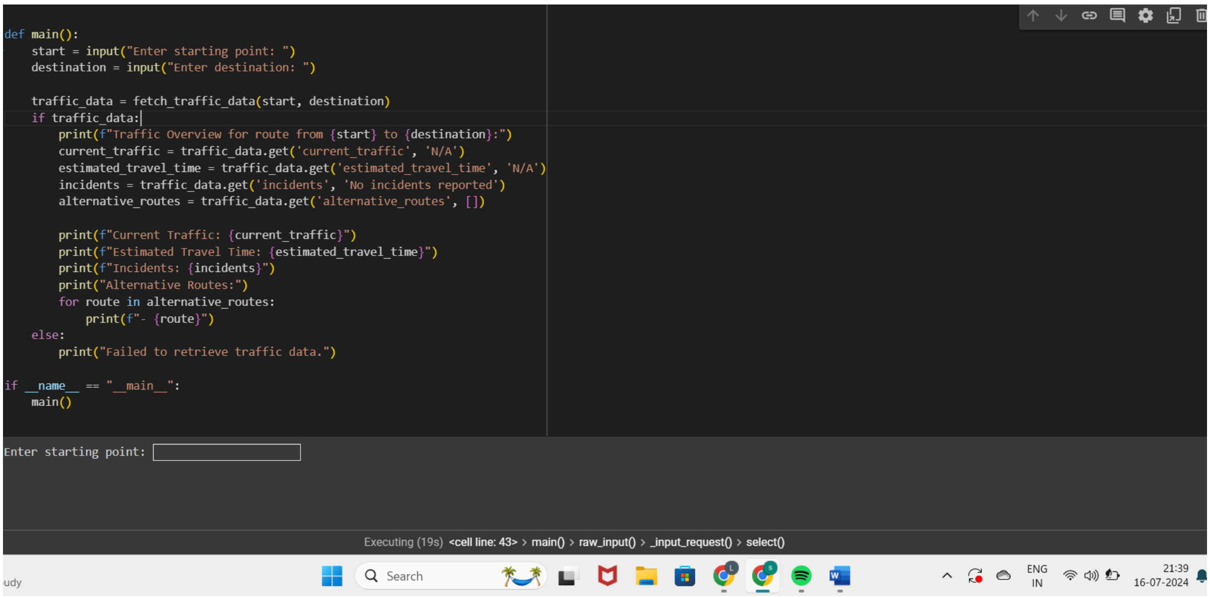


Implementation:

import requests

```
url = "https://mock-api.com/traffic"
def fetch_traffic_data(start, destination):
    params = {       'origin': start,
        'destination': destination
    }
    response = requests.get(url, params=params)
    if response.status_code == 200:
        try:
            data = response.json()
            return data
        except ValueError:
            print("Error: Unable to parse JSON response.")
            return None
    else:
        print(f"Error fetching data: {response.status_code} - {response.text}")
        return None

def main():
    start = input("Enter starting point: ")
    destination = input("Enter destination: ")

    traffic_data = fetch_traffic_data(start, destination)
    if traffic_data:
        print(f"Traffic Overview for route from {start} to {destination}:")
        current_traffic = traffic_data.get('current_traffic', 'N/A')
        estimated_travel_time = traffic_data.get('estimated_travel_time', 'N/A')
        incidents = traffic_data.get('incidents', 'No incidents reported')
        alternative_routes = traffic_data.get('alternative_routes', [])

        print(f"Current Traffic: {current_traffic}")
```

```
        print(f"Estimated Travel Time: {estimated_travel_time}")
        print(f"Incidents: {incidents}")
        print("Alternative Routes:")
        for route in alternative_routes:
            print(f"- {route}")
    else:
        print("Failed to retrieve traffic data.")

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



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 COVID19 data.

• Explanation of any assumptions made and potential improvements

Implementation:

```
    mport requests


API_ENDPOINT = "https://disease.sh/v3/covid-19"

def fetch_covid_stats(region):
    response = requests.get(f"{API_ENDPOINT}/all?region={region}")
    return response.json()

def display_data(data):
    print(f"Region: {data.get('country', 'N/A')}")
    print(f"Cases: {data.get('cases', 'N/A')}")
    print(f"Recoveries: {data.get('recovered', 'N/A')}")
    print(f"Deaths: {data.get('deaths', 'N/A')}")

def main():
    region = input("Enter the region (country, state, or city): ")
    covid_data = fetch_covid_stats(region)
    display_data(covid_data)

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

Displaying data:

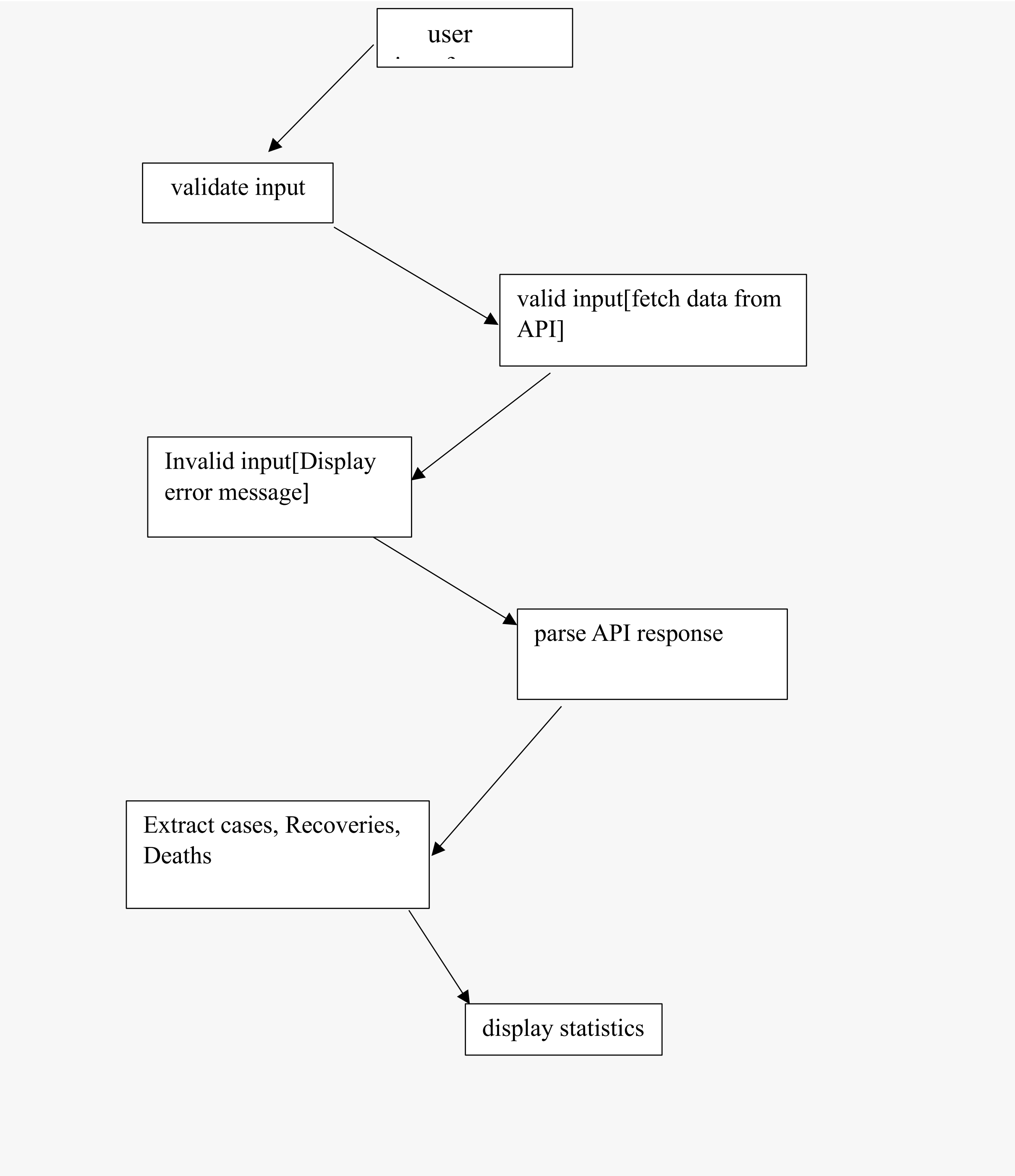
Input:

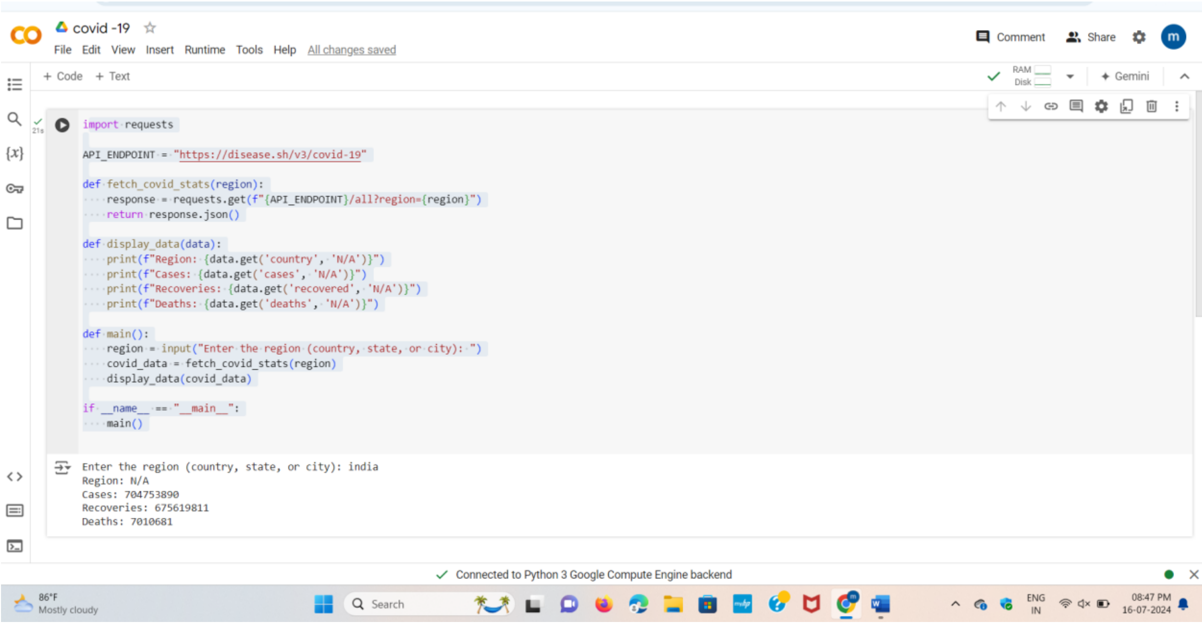
Enter the region (country, state, or city): india

Output:

Region: N/A
Cases: 704753890
Recoveries: 675619811
Deaths: 7010681

Data flow diagram:




covid -19
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Gemini
import requests
API_ENDPOINT = "https://disease.sh/v3/covid-19"
def fetch_covid_stats(region):
    response = requests.get(f"{API_ENDPOINT}/all?region={region}")
    return response.json()
def display_data(data):
    print(f"Region: {data.get('country', 'N/A')}")
    print(f"Cases: {data.get('cases', 'N/A')}")
    print(f"Recoveries: {data.get('recovered', 'N/A')}")
    print(f"Deaths: {data.get('deaths', 'N/A')}")
def main():
    region = input("Enter the region (country, state, or city): ")
    covid_data = fetch_covid_stats(region)
    display_data(covid_data)
if __name__ == "__main__":
    main()
Enter the region (country, state, or city): india
Region: N/A
Cases: 704753890
Recoveries: 675619811
Deaths: 7010681
Connected to Python 3 Google Compute Engine backend
86°F
Mostly cloudy
Search
ENG
IN
08:47 PM
16-07-2024