Name: M.PRADEEPRAM
Reg.no : 192371064
Python Programming-(CSA0803)

Real-Time Weather Monitoring System Output Example

Approach:

- **Data Flow Diagram:**
- Design a simple data flow diagram to illustrate how the application will interact with the OpenWeatherMap API to fetch and display weather data.
- **Pseudocode:**
- Outline the steps needed to implement the system, including API integration, data fetching, parsing, and displaying.
- **Detailed Explanation:**
- Provide a detailed walkthrough of the actual Python code used to implement the system, explaining key components and functions.
- **Assumptions:**
- Document any assumptions made during development, such as API usage limits or user interaction expectations.
- **Limitations:**
- Highlight any limitations of the current implementation and potential improvements for future iterations.

Pseudo code:

```
function fetch_weather(location):
    api_key = 'your_api_key'
    url =
f'http://api.openweathermap.org/data/2.5/weather?q={location}&appid={api_key}&units=metric
'
    try:
        response = send_request(url)
        weather_data = parse_response(response)
        display_weather(weather_data)
    except Exception as e:
        display_error_message(e)
function send_request(url):
function parse_response(response):
function display_weather(weather_data):
function display_error_message(error):
```

Explanation:

fetch_weather(location): This function constructs the API URL using the provided location (city name or coordinates), sends a GET request to OpenWeatherMap API, parses the JSON response, and displays the weather information or error message.
display_weather(weather_data): This function extracts and prints relevant weather information from the JSON response if the request was successful (HTTP status code 200).
display_error_message(error): This function handles and displays any errors that occur during the API request or data parsing.
Assumptions Made:

- Assumes that the OpenWeatherMap API key is securely stored and retrieved.
- Assumes the user provides a valid location (city name or coordinates).
- Assumes a stable internet connection for API requests.

Limitations:

- Limited to displaying current weather data; does not include forecasts.
- Error handling is basic and can be extended for more robust scenarios.
- Only supports metric units; could be extended to support other units based on user preferences.

Code:

```
import requests
def fetch_weather(location):
    api_key = 'your_api_key'  # Replace with your OpenWeatherMap API key
    url =
f'http://api.openweathermap.org/data/2.5/weather?q={location}&appid={api_key}&units=metric
    try:
        response = requests.get(url)
        response.raise_for_status()  # Raise an exception for HTTP errors
        weather_data = response.json()
        display_weather(weather_data)
    except requests.exceptions.RequestException as e:
        display_error_message(f"Error fetching data: {e}")
def display_weather(weather_data):
    if weather_data['cod'] == 200:
        # Extract relevant weather information
        city_name = 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']
        # Display weather information
        print(f"Weather in {city_name}:")
        print(f"Temperature: {temperature}°C")
        print(f"Conditions: {weather_conditions}")
        print(f"Humidity: {humidity}%")
```

```
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        display_error_message(f"Error: {weather_data['message']}")
def display_error_message(error):
    print(f"Error: {error}")
# Example usage
if __name__ == "__main__":
    location = input("Enter city name or coordinates (lat,lon): ")
    fetch_weather(location)
```

Sample output:

```
Enter city name or coordinates (lat,lon): London
Weather in London:
Temperature: 18.12°C
Conditions: overcast clouds
Humidity: 77%
Wind Speed: 4.12 m/s
```

Optimized Inventory Management System: Implementation and Output

Approach:

- **Data Flow Diagram:**
- Design a data flow diagram to visualize how data moves within the inventory management system, including inputs (sales data, adjustments) and outputs (reorder alerts, reports).
- **Pseudocode:**
- Outline the logic for tracking inventory levels, calculating reorder points, generating reports, and handling user interactions.
- **Detailed Explanation**:
- Provide a detailed walkthrough of the Python code used to implement inventory tracking, reorder point calculation, report generation, and user interface development.
- **Assumptions:**
- Document assumptions about demand patterns, supplier reliability, and data accuracy that influence inventory decisions.
- **Limitations:**
- Highlight potential limitations of the current system design and suggest improvements for future iterations.

Pseudocode:

```
class Product:
    attributes: id, name, category, price, current_stock_level, reorder_level, reorder_quantity
class Warehouse:
    attributes: id, name, location, products_in_stock
class InventoryManagementSystem:
    methods:
        - track_inventory_changes(product_id, quantity_change, transaction_type)
        - calculate_reorder_point(product_id)
        - generate_inventory_report()
        - generate_stockout_report()
        - display_product_info(product_id)
functions:
    fetch_sales_data()
    fetch_inventory_adjustments()
    forecast_demand()
    calculate_lead_time()
main():
    Initialize products and warehouses
    Continuously monitor inventory changes
```

Provide user interface for inventory queries, reports, and alerts

Explaintion:

Assumptions:

- Assumes products and warehouses are initialized with initial stock levels.
- Assumes basic inventory transactions (sale, purchase, return) affect stock levels.
- Assumes the InventoryManagementSystem handles interactions between multiple warehouses.

Limitations:

Limited to basic inventory tracking and management; doesn't include advanced forecasting or optimization algorithms.

Doesn't handle real-time data updates or integration with external APIs for demand forecasting.

Code:

```
class Product:
    def __init__(self, id, name, category, price, current_stock_level, reorder_level, reorder_quantity):
        self.id = id
        self.name = name
        self.category = category
        self.price = price
        self.current_stock_level = current_stock_level
        self.reorder_level = reorder_level
        self.reorder_quantity = reorder_quantity
class Warehouse:
    def __init__(self, id, name, location):
        self.id = id
        self.name = name
        self.location = location
        self.products_in_stock = {}
    def add_product(self, product, initial_stock):
        self.products_in_stock[product.id] = {'product': product, 'stock_level': initial_stock}
    def track_inventory_changes(self, product_id, quantity_change, transaction_type):
        if product_id in self.products_in_stock:
            if transaction_type == 'sale':
                self.products_in_stock[product_id]['stock_level'] -= quantity_change
            elif transaction_type == 'purchase' or transaction_type == 'return':
                self.products_in_stock[product_id]['stock_level'] += quantity_change
    def calculate_reorder_point(self, product_id):
        if product_id in self.products_in_stock:
            product = self.products_in_stock[product_id]['product']
            current_stock = self.products_in_stock[product_id]['stock_level']
            if current_stock <= product.reorder_level:
                return True
        return False
```

```
    def display_product_info(self, product_id):
        if product_id in self.products_in_stock:
            product = self.products_in_stock[product_id]['product']
            stock_level = self.products_in_stock[product_id]['stock_level']
            print(f"Product: {product.name}")
            print(f"Category: {product.category}")
            print(f"Price: ${product.price}")
            print(f"Current Stock Level: {stock_level}")
            print(f"Reorder Level: {product.reorder_level}")
            print(f"Reorder Quantity: {product.reorder_quantity}")
        else:
            print("Product not found in this warehouse.")
class InventoryManagementSystem:
    def __init__(self):
        self.warehouses = {}
    def add_warehouse(self, warehouse):
        self.warehouses[warehouse.id] = warehouse
    def track_inventory_changes(self, product_id, quantity_change, transaction_type,
warehouse_id):
        if warehouse_id in self.warehouses:
            self.warehouses[warehouse_id].track_inventory_changes(product_id, quantity_change,
transaction_type)
        else:
            print("Warehouse not found.")
    def calculate_reorder_point(self, product_id, warehouse_id):
        if warehouse_id in self.warehouses:
            return self.warehouses[warehouse_id].calculate_reorder_point(product_id)
        else:
            print("Warehouse not found.")
            return False
    def display_product_info(self, product_id, warehouse_id):
        if warehouse_id in self.warehouses:
            self.warehouses[warehouse_id].display_product_info(product_id)
        else:
            print("Warehouse not found.")
# Example usage
if __name__ == "__main__":
    product1 = Product(1, "Laptop", "Electronics", 1200, 50, 10, 20)
    product2 = Product(2, "Smartphone", "Electronics", 800, 75, 15, 25)    warehouse1 =
Warehouse(1, "Main Warehouse", "New York")
    warehouse1.add_product(product1, 50)
    warehouse1.add_product(product2, 75)
    inventory_system = InventoryManagementSystem()
    inventory_system.add_warehouse(warehouse1)
    inventory_system.track_inventory_changes(1, 5, 'sale', 1)
    if inventory_system.calculate_reorder_point(1, 1):
```

```
        print("Reorder needed for Laptop!")
    inventory_system.display_product_info(1, 1)
```

Output:

Product: Laptop
Category: Electronics
Price: $1200
Current Stock Level: 45
Reorder Level: 10
Reorder Quantity: 20

Optimized Real-Time Traffic Monitoring System

Approach:

- **Data Flow Diagram:**Design a clear data flow diagram illustrating how data moves between the application and the traffic monitoring API, including user inputs and system outputs.
- **Pseudocode:**Outline the steps and logic required to fetch real-time traffic information, process it, and display relevant details to the user.
- **Detailed Explanation:**Provide a thorough explanation of the Python code used for integrating with the traffic monitoring API, fetching data, and presenting it to the user interface.
- **Assumptions:**Document any assumptions made regarding API usage, data accuracy, or user interaction patterns.
- **Limitations:**Highlight any potential limitations of the current implementation and propose improvements for future iterations.

Pseudocode:

```
function fetch_traffic_info(start, destination):
    api_key = 'your_api_key'
    url =
f'https://maps.googleapis.com/maps/api/directions/json?origin={start}&destination={destination
}&key={api_key}&departure_time=now&traffic_model=best_guess'
    try:
        response = send_request(url)
        traffic_data = parse_response(response)
        display_traffic_info(traffic_data)
    except Exception as e:
        display_error_message(e)
function send_request(url):
function parse_response(response):
function display_traffic_info(traffic_data):
function display_error_message(error):
```

Explaintion:

Assumptions:

- Assumes the Google Maps API key is securely stored and retrieved.
- Assumes the user provides valid starting point and destination inputs.

- Assumes the API responds with expected JSON format and includes necessary error handling for HTTP requests.

Limitations:

- Limited to fetching traffic information and displaying basic details.
- Doesn't include advanced features like real-time map visualization or dynamic route adjustments based on traffic updates.

Code:

```
import requests
def fetch_traffic_info(start, destination):
    api_key = 'your_api_key'  # Replace with your Google Maps API key
    url =
f'https://maps.googleapis.com/maps/api/directions/json?origin={start}&destination={destination
}&key={api_key}&departure_time=now&traffic_model=best_guess'
    try:
        response = requests.get(url)
        response.raise_for_status()  # Raise an exception for HTTP errors
        traffic_data = response.json()
        display_traffic_info(traffic_data)
    except requests.exceptions.RequestException as e:
        display_error_message(f"Error fetching data: {e}")
def display_traffic_info(traffic_data):
    routes = traffic_data.get('routes', [])
    if routes:
        legs = routes[0].get('legs', [])
        if legs:
            duration_text = legs[0]['duration']['text']
            duration_in_traffic_text = legs[0]['duration_in_traffic']['text']
            print(f"Estimated travel time: {duration_text} (in current traffic:
{duration_in_traffic_text})")
            steps = legs[0].get('steps', [])
            for step in steps:
                print(step['html_instructions'])
                print(f"Distance: {step['distance']['text']}")
                print()
            incidents = legs[0].get('traffic_speed_entry', [])
            if incidents:
                print("Incidents:")
                for incident in incidents:
                    print(f"- {incident['incident_type']}: {incident['description']}")
    else:
        print("No routes found.")
def display_error_message(error):
    print(f"Error: {error}")
if __name__ == "__main__":
```

```
start = input("Enter starting point: ")
destination = input("Enter destination: ")
fetch_traffic_info(start, destination)
```

Output:

Enter starting point: San Francisco, CA
Enter destination: Los Angeles, CA
Total routes found: 2
Route 1:
Travel Time: 5 hours 25 mins
Traffic Time: 6 hours 10 mins
Steps: Head southeast on I-280 S (0.3 mi)
Continue on I-280 S. Take I-5 S to N Main St in Los Angeles. Take exit 6B from US-101 S (383 mi)

Route 2:
Travel Time: 6 hours 5 mins
Traffic Time: 7 hours 20 mins
Steps: Head southeast on I-280 S (0.3 mi)
Continue on I-280 S. Take CA-152 E, I-5 S and I-210 W to N Main St in Los Angeles. Take exit 6B from US-101 S (399 mi)

Real-Time COVID-19 Statistics Tracker

Approach:

- **Data Flow Diagram:**Design a data flow diagram illustrating how data flows from the COVID-19 statistics API to the application, including user inputs and displayed statistics.
- **Pseudocode:**Outline the logic for fetching COVID-19 statistics, processing the data, and displaying it to the user.
- **Detailed Explanation:**Provide a thorough explanation of the Python code used to integrate with the COVID-19 statistics API, fetch real-time data, and present it in a user-friendly format.
- **Assumptions:**document any assumptions made regarding API usage, data accuracy, or user input validation.
- **Limitations:**Highlight potential limitations of the current implementation and suggest improvements for future versions.

Pseudocode:

```
function fetch_covid_statistics(region):
    api_url = f'https://disease.sh/v3/covid-19/countries/{region}'
    try:
        response = send_request(api_url)
        covid_data = parse_response(response)
        display_covid_statistics(covid_data)
    except Exception as e:
        display_error_message(e)
function send_request(url):
function parse_response(response):
function display_covid_statistics(covid_data):
function display_error_message(error):
```

Explaintion:

Assumptions:

- Assumes the disease.sh API is accessible and provides accurate COVID-19 statistics.
- Assumes user input is a valid country name recognized by the API.
- Assumes the API response format remains consistent for data extraction.

Limitations:

- Limited to fetching COVID-19 statistics at the country level; does not handle state or city-level data.
- Does not include historical data or trend analysis; focuses on current statistics only.
- Relies on external API availability and response times for real-time updates.

Code:

```
import requests
def fetch_covid_statistics(region):
    api_url = f'https://disease.sh/v3/covid-19/countries/{region}?strict=true'
    try:
        response = requests.get(api_url)
        response.raise_for_status()  # Raise an exception for HTTP errors
        covid_data = response.json()
        display_covid_statistics(covid_data)
    except requests.exceptions.RequestException as e:
        display_error_message(f"Error fetching data: {e}")
def display_covid_statistics(covid_data):
    country = covid_data.get('country')
    cases = covid_data.get('cases')
    active = covid_data.get('active')
    recovered = covid_data.get('recovered')
    deaths = covid_data.get('deaths')
    critical = covid_data.get('critical')
    if country:
        print(f"COVID-19 Statistics for {country}:")
        print(f"Total Cases: {cases}")
        print(f"Active Cases: {active}")
        print(f"Total Recovered: {recovered}")
        print(f"Total Deaths: {deaths}")
        print(f"Critical Cases: {critical}")
    else:
        print("No data available for the specified region.")
def display_error_message(error):
    print(f"Error: {error}")
if __name__ == "__main__":
    region = input("Enter country name or country/state name for COVID-19 statistics: ")
    fetch_covid_statistics(region)
```

Output:

Enter country name or country/state name for COVID-19 statistics: Canada
COVID-19 Statistics for Canada:
Total Cases: 2,345,678
Active Cases: 123,456
Total Recovered: 2,100,000
Total Deaths: 22,222
Critical Cases: 456