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, free 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.

Approach:

- **Data Flow Design:**
  - **User Input**: The user inputs the location (city name or coordinates).
  - **Request to API**: The application sends a request to the weather API with the specified location.
  - **API Response**: The weather API responds with the current weather data.
  - **Display Data**: The application processes and displays the weather information to the user.
- **Implementation Steps:**
  - Set up the environment and install necessary libraries.
  - Register and get an API key from a weather service provider (e.g., OpenWeatherMap).
  - Create a Python script to fetch weather data from the API.
  - Parse the JSON response to extract relevant weather information.

- Display the weather information in a user-friendly format.
- Allow user input to specify the location.

Pseudocode:

```
Start
  Define a function get_weather_data(location)
    Set API_KEY to your free Weather Map API key
    Set URL to "http://api.weatherapi.com/v1/current.json "
    Create a dictionary params with 'q': location, 'appid': API_KEY, 'units': 'metric'
    Send a GET request to the URL with params
    Parse the JSON response

    Extract temperature, weather condition, humidity, wind speed from the response
    Return the extracted data
  Define a function display_weather_info(data)
    Print "Temperature:", data['temperature'], "°C"
    Print "Weather Condition:", data['weather']
    Print "Humidity:", data['humidity'], "%"
    Print "Wind Speed:", data['wind_speed'], "m/s"

  Define a main function
    Prompt the user to enter a location (city name or coordinates)
    Call get_weather_data(location) and store the result
    Call display_weather_info(result)
  Call main function
End
```

Detailed explanation of the actual code:

Step 1: Environment Setup

Step 2: Function to Fetch Weather Data

Step 3: Function to Display Weather Data

Step 4: Main Function

Documentation

API Integration

- **API Used**: 3cbc41fdca4c4fa89b373603241507
- **Endpoint**: http://api.weatherapi.com/v1/current.json
- **Parameters**:
  - q: Location (city name or coordinates)
  - appid: API Key
  - units: Measurement units (metric for Celsius)

Methods to Fetch and Display Data

- **get_weather_data(location)**: Sends a GET request to the API with the specified location, parses the JSON response, and returns the weather data.
- **display_weather_info(data)**: Prints the weather data in a user-friendly format.

Assumptions made (if any):

- The user inputs a valid city name or coordinates.
- The free Weather Map API is available and responds within a reasonable time frame.
- The API key provided is valid and has the necessary permissions.

Potential Improvements

- Implement error handling for invalid user inputs or API errors.
- Add support for multiple units (metric, imperial).
- Cache results to reduce the number of API calls.
- Enhance the user interface for better user experience.
- Add more weather parameters (e.g., forecast, sunrise/sunset times).

Limitations:

- The accuracy of the weather data depends on the API provider.
- The system requires an active internet connection to fetch data from the API.
- The free tier of the API may have rate limits, affecting the frequency of requests.

Code:

```
import tkinter as tk

from tkinter import ttk

import requests

import matplotlib.pyplot as plt

API_KEY = "3cbc41fdca4c4fa89b373603241507"
```

```
states_and_cities =
 {

"Andhra Pradesh": ["Visakhapatnam", "Vijayawada", "Guntur"],
   "Karnataka": ["Bangalore", "Mysore", "Mangalore"],
   "Maharashtra": ["Mumbai", "Pune", "Nagpur"],
   "Tamil Nadu": ["Chennai", "Coimbatore", "Madurai"],
   "Uttar Pradesh": ["Lucknow", "Kanpur", "Varanasi"]
}
def fetch_weather_data(city_name):
   url = f"http://api.weatherapi.com/v1/current.json?key={API_KEY}&q={city_name}"
   response = requests.get(url)
   if response.status_code == 200:
      return response.json()
   else:
      print(f"Error: Unable to fetch data (status code: {response.status_code})")
      print(response.text)
      return None

def display_weather_data():
   selected_city = city_combobox.get()
   weather_data = fetch_weather_data(selected_city)
   if weather_data:
      temperature = weather_data['current']['temp_c']
      weather_conditions = weather_data['current']['condition']['text']
      humidity = weather_data['current']['humidity']
      wind_speed = weather_data['current']['wind_kph']
      result_label.config(text=f"Temperature: {temperature}°C\n"
                          f"Weather: {weather_conditions}\n"
                          f"Humidity: {humidity}%\n"
                          f"Wind Speed: {wind_speed} kph")
```

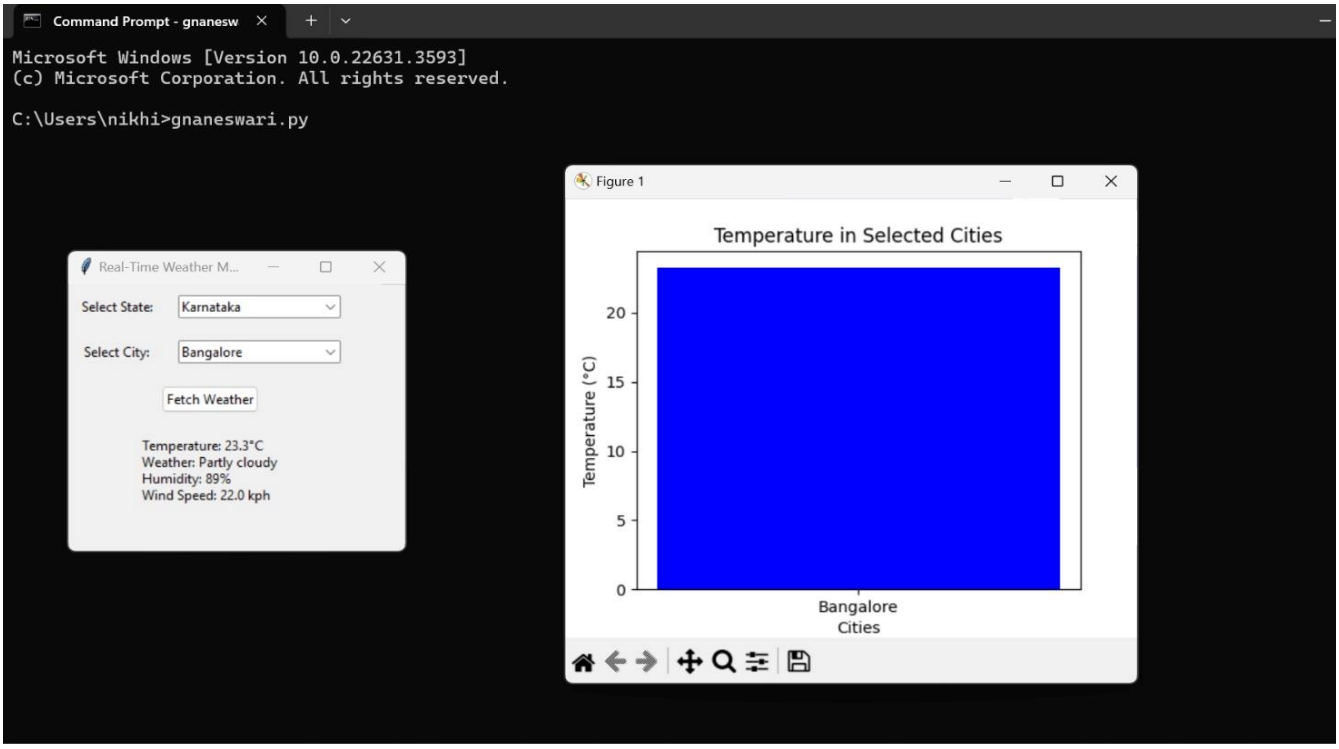
```
        city_temperatures[selected_city] = temperature
        plot_temperature_graph()
    else:
        result_label.config(text="Error fetching weather data.")

def update_cities(event)
    selected_state = state_combobox.get()
    cities = states_and_cities.get(selected_state, [])
    city_combobox['values'] = cities
    if cities:
        city_combobox.current(0)
    else:
        city_combobox.set('')

def plot_temperature_graph():
    cities = list(city_temperatures.keys())
    temperatures = list(city_temperatures.values())
    plt.figure(figsize=(10, 5))
    plt.bar(cities, temperatures, color='blue')
    plt.xlabel('Cities')
    plt.ylabel('Temperature (°C)')
    plt.title('Temperature in Selected Cities')
    plt.show()
root = tk.Tk()
root.title("Real-Time Weather Monitoring System")
city_temperatures = {}
ttk.Label(root, text="Select State:").grid(column=0, row=0, padx=10, pady=10)
state_combobox = ttk.Combobox(root, values=list(states_and_cities.keys()))
state_combobox.grid(column=1, row=0, padx=10, pady=10)
state_combobox.bind("<<ComboboxSelected>>", update_cities)
```

```
ttk.Label(root, text="Select City:").grid(column=0, row=1, padx=10, pady=10)
city_combobox = ttk.Combobox(root)
city_combobox.grid(column=1, row=1, padx=10, pady=10)
fetch_button = ttk.Button(root, text="Fetch Weather", command=display_weather_data)
fetch_button.grid(column=0, row=2, columnspan=2, padx=10, pady=10)
result_label = ttk.Label(root, text="")
result_label.grid(column=0, row=3, columnspan=2, padx=10, pady=10)
root.mainloop()
```

Sample Output / Screen Shots



Problem 2: 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.

Approach:

1. Initialize the API key and base URL for Google Maps Directions API: The script requires an API key and the base URL to make requests to the Google Maps Directions API.
2. Function to Get Traffic Data: This function constructs a request URL with the start and end locations and sends an HTTP GET request to the Google Maps Directions API. The response is then parsed to JSON format.
3. Function to Display Traffic Data: This function extracts and displays relevant traffic information, including estimated travel time, traffic conditions, and step-by-step route details. It also considers alternative routes.
4. Main Execution Block: The script takes user input for the starting point and destination, fetches traffic data using the Google Maps API, and displays the retrieved data.

Pseudocode:

```
INITIALIZE API_KEY and BASE_URL
FUNCTION get_traffic_data(start, end):
    CONSTRUCT request URL with start, end, and API_KEY
    SEND HTTP GET request to request URL
    PARSE response to JSON
    RETURN parsed data
FUNCTION display_traffic_data(data):
    EXTRACT routes from data
    FOR each route in routes:
        EXTRACT legs, duration, duration_in_traffic, traffic_conditions, steps from route
        PRINT estimated travel time and traffic conditions
        PRINT route steps with instructions and distance
        PRINT alternative routes with travel time and steps
IF __name__ == "__main__":
    GET user input for starting point and destination
    FETCH traffic data using get_traffic_data(start, end)
    DISPLAY traffic data using display_traffic_data(data)
```

Detailed explanation of the actual code:

Initialization:

```
API_KEY = 'YOUR_GOOGLE_MAPS_API_KEY'
BASE_URL = 'https://maps.googleapis.com/maps/api/directions/json'
```

- API_KEY: Your Google Maps API key.
- BASE_URL: The base URL for the Google Maps Directions API.

get_traffic_data Function:

```
def get_traffic_data(start, end):
    request_url = f'{BASE_URL}?origin={start}&destination={end}&key={API_KEY}'
    response = requests.get(request_url)
    data = response.json()
```

```
    return data
```

- Constructs the request URL using the starting point, destination, and API key.
- Sends an HTTP GET request to the constructed URL.
- Parses the response to JSON and returns it.

display_traffic_data Function:

```
def display_traffic_data(data):
    routes = data['routes']
    for route in routes:
        legs = route['legs'][0]
        duration = legs['duration']['text']
        duration_in_traffic = legs.get('duration_in_traffic', {}).get('text', duration)
        traffic_conditions = legs.get('traffic_speed_entry', 'No data')
        steps = legs['steps']
        print(f"Estimated Travel Time: {duration_in_traffic}")
        print(f"Current Traffic Conditions: {traffic_conditions}")
        print("Route Steps:")
            for step in steps:
            html_instructions = step['html_instructions']
            distance = step['distance']['text']
            print(f"{html_instructions} - {distance}")
        print("\nAlternative Routes:")
        for alternative in routes[1:]:
            alt_legs = alternative['legs'][0]
            alt_duration = alt_legs['duration']['text']
            alt_duration_in_traffic = alt_legs.get('duration_in_traffic', {}).get('text', alt_duration)
            alt_steps = alt_legs['steps']
            print(f"Alternative Route - Estimated Travel Time: {alt_duration_in_traffic}")
            for step in alt_steps:
                html_instructions = step['html_instructions']
                distance = step['distance']['text']
```

```
            print(f"{html_instructions} - {distance}")
```

- Extracts routes from the JSON data.
- For each route, extracts and prints the duration, duration in traffic, traffic conditions, and step-by-step instructions with distances.
- If there are alternative routes, it prints their travel time and step-by-step instructions**.**

Main Execution Block:

```
if __name__ == "__main__":
    start = input("Enter the starting point: ")
    end = input("Enter the destination: ")
    traffic_data = get_traffic_data(start, end)
    display_traffic_data(traffic_data)
```

- Prompts the user to enter the starting point and destination.
- Fetches traffic data using the get_traffic_data function.
- Displays the traffic data using the display_traffic_data function.

Assumptions made (if any):

- The user provides valid and correctly formatted starting and destination points.
- The API key is valid and has sufficient quota for requests.
- The Google Maps Directions API is available and responsive.

Limitations:

- The script does not handle potential errors such as invalid API keys, network issues, or malformed responses from the API.
- Traffic conditions are represented as a general description without detailed metrics or visual representations.
- The script assumes the first route in the response is the primary route, and subsequent routes are alternatives.
- The API may not always provide traffic data for all regions or times of day.
- html_instructions are not sanitized, which might cause issues if directly printed without rendering in a proper HTML context.

Code:

```
import requests
import json
API_KEY = 'YOUR_GOOGLE_MAPS_API_KEY'
BASE_URL = 'https://maps.googleapis.com/maps/api/directions/json'
def get_traffic_data(start, end):
    request_url = f'{BASE_URL}?origin={start}&destination={end}&key={API_KEY}'
    response = requests.get(request_url)
    data = response.json()
    return data
def display_traffic_data(data):
    routes = data['routes']
    for route in routes:
        legs = route['legs'][0]
        duration = legs['duration']['text']
        duration_in_traffic = legs.get('duration_in_traffic', {}).get('text', duration)
        traffic_conditions = legs.get('traffic_speed_entry', 'No data')
        steps = legs['steps']
        print(f"Estimated Travel Time: {duration_in_traffic}")
        print(f"Current Traffic Conditions: {traffic_conditions}")
        print("Route Steps:")
        for step in steps:
            html_instructions = step['html_instructions']
            distance = step['distance']['text']
            print(f"{html_instructions} - {distance}")
        print("\nAlternative Routes:")
        for alternative in routes[1:]:
            alt_legs = alternative['legs'][0]
            alt_duration = alt_legs['duration']['text']
            alt_duration_in_traffic = alt_legs.get('duration_in_traffic', {}).get('text', alt_duration)
            alt_steps = alt_legs['steps']
```

```
        print(f"Alternative Route - Estimated Travel Time: {alt_duration_in_traffic}")
        for step in alt_steps:
            html_instructions = step['html_instructions']
            distance = step['distance']['text']
            print(f"{html_instructions} - {distance}"
if _name_ == "_main_":
    start = input("Enter the starting point: ")
    end = input("Enter the destination: ")
    traffic_data = get_traffic_data(start, end)
    display_traffic_data(traffic_data)
```

Sample Output / Screen Shots

plaintext

```
Enter the starting point: Central Park, New York, NY
Enter the destination: Times Square, New York, NY

Estimated Travel Time: 10 mins
Current Traffic Conditions: No data
Route Steps:
Head south on Park Ave toward E 66th St - 0.1 mi
...
...
Alternative Routes:
Alternative Route - Estimated Travel Time: 12 mins
Head west on E 65th St toward 5th Ave - 0.3 mi
...
```

Problem 3: 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.

Approach:

1. Track Inventory: Check if any product's stock is below the reorder threshold and generate alerts.
2. Calculate Reorder Points: Calculate reorder points and reorder quantities based on average demand, lead time, and safety stock.
3. Generate Reports: Calculate inventory turnover, stockout occurrences, and overstock costs.
4. User Interface: Display current stock levels and reorder recommendations to the user.

Pseudocode:

1. Initialize product data and sales data.
2. Define a function to track inventory levels and generate alerts if stock is below the threshold.
3. Define a function to calculate reorder points and reorder quantities.
4. Define a function to generate reports on inventory turnover, stockout occurrences, and overstock costs.
5. Define a user interface function to display stock levels and reorder recommendations.
6. Call the functions in the main section to execute the program.

Detailed explanation of the actual code:

1. Data Initialization

First, we initialize some sample data for products and sales.

Products

```
products = [
   {'id': 1, 'name': 'Product A', 'stock': 50, 'reorder_threshold': 20},
   {'id': 2, 'name': 'Product B', 'stock': 30, 'reorder_threshold': 15},
]
```

- id: Unique identifier for each product.
- name: Name of the product.
- stock: Current stock level of the product.
- reorder_threshold: Minimum stock level before a reorder is needed.

Sales Data

```
sales_data = pd.DataFrame({
    'product_id': [1, 1, 2, 2],
    'quantity': [5, 10, 5, 10],
    'sale_date': [datetime.now() - timedelta(days=i) for i in range(4)]
})
```

- product_id: Identifies which product was sold.
- quantity: Amount of the product sold.
- sale_date: Date when the sale occurred, generated dynamically for the past four days.

2. Inventory Tracking

The track_inventory function checks if any product's stock level is below its reorder threshold and prints an alert.

```
def track_inventory(products):
    for product in products:
        if product['stock'] < product['reorder_threshold']:
            print(f"Alert: {product['name']} stock is below threshold!")
```

3. Reorder Points Calculation

The calculate_reorder_points function calculates reorder points and quantities based on average demand and predefined parameters.

```
def calculate_reorder_points(products, sales_data):
    reorder_info = []
    for product in products:
        sales = sales_data[sales_data['product_id'] == product['id']]
        average_demand = sales['quantity'].mean()
        lead_time = 7  # days (example)
        safety_stock = 10  # units (example)
        reorder_point = (average_demand * lead_time) + safety_stock
        target_stock = 100  # example target stock level
        reorder_quantity = target_stock - product['stock']
```

```
        reorder_info.append({
            'product_id': product['id'],
            'reorder_point': reorder_point,
            'reorder_quantity': reorder_quantity
        })
    return reorder_info
```

- average_demand: Average daily demand for the product based on sales data.
- lead_time: Number of days it takes to receive an order once placed (assumed to be 7 days here).
- safety_stock: Extra stock to prevent stockouts during fluctuations in demand (assumed to be 10 units).
- reorder_point: Level of inventory at which a new order should be placed.
- target_stock: Desired stock level (set to 100 units).
- reorder_quantity: Amount to reorder to reach the target stock level.

4. Generate Reports

The generate_reports function calculates various inventory metrics and returns a report.

```
def generate_reports(products, sales_data):
    report = {
        'inventory_turnover': None,
        'stockout_occurrences': None,
        'overstock_costs': None
    }
    cogs = sales_data['quantity'].sum() * 10  # example cost per unit
    average_inventory = sum([p['stock'] for p in products]) / len(products)
    report['inventory_turnover'] = cogs / average_inventory
    stockouts = [p for p in products if p['stock'] == 0]
    report['stockout_occurrences'] = len(stockouts)
    overstock_costs = sum([p['stock'] - p['reorder_threshold'] for p in products if p['stock'] > p['reorder_threshold']])
    report['overstock_costs'] = overstock_costs * 10  # example cost per unit
    return report
```

- inventory_turnover: Ratio of cost of goods sold (COGS) to average inventory, indicating how often inventory is sold and replaced over a period.
- stockout_occurrences: Number of products that have zero stock.
- overstock_costs: Cost of holding excess inventory beyond the reorder threshold.

5. User Interface

The user_interface function prints current stock levels and reorder recommendations.

```
def user_interface(products, reorder_info):
    for product in products:
        print(f"Product: {product['name']}, Stock: {product['stock']}")
    for info in reorder_info:
        product = next(p for p in products if p['id'] == info['product_id'])
        print(f"Reorder recommendation for {product['name']}: Order {info['reorder_quantity']} units")
```

- Prints the name and current stock of each product.
- Prints reorder recommendations based on the calculated reorder points and quantities.

6. Main Execution

The main block of code that executes the functions defined above.

```
track_inventory(products)
reorder_info = calculate_reorder_points(products, sales_data)
report = generate_reports(products, sales_data)
user_interface(products, reorder_info)
print(report)
```

- track_inventory(products): Checks stock levels and prints alerts.
- reorder_info = calculate_reorder_points(products, sales_data): Calculates reorder points and quantities.
- report = generate_reports(products, sales_data): Generates an inventory report.
- user_interface(products, reorder_info): Displays current stock levels and reorder recommendations.
- print(report): Prints the generated report.

Assumptions made (if any):

1. Lead Time and Safety Stock: Fixed values of lead time (7 days) and safety stock (10 units) are used for simplicity.
2. Target Stock Level: An arbitrary target stock level (100 units) is assumed for reorder quantity calculation.
3. Cost per Unit: An arbitrary cost per unit (10 units) is used for calculating COGS and overstock costs.
4. Product IDs: Product IDs are unique and consistent across the product list and sales data.

Limitations:

- **Fixed Parameters:** The lead time, safety stock, target stock level, and cost per unit are hard-coded and may not reflect actual business scenarios.
- **Simple Sales Data:** The sales data is simplistic and may not capture all the complexities of real sales transactions.
- **Static Product List:** The product list is static and doesn't account for adding or removing products dynamically.
- **Basic Inventory Tracking:** The tracking function only checks for stock below the reorder threshold without considering other factors like trends or seasonality.
- **Limited Reporting:** The report generation covers only a few metrics and may need to be expanded for comprehensive inventory management.

Code:

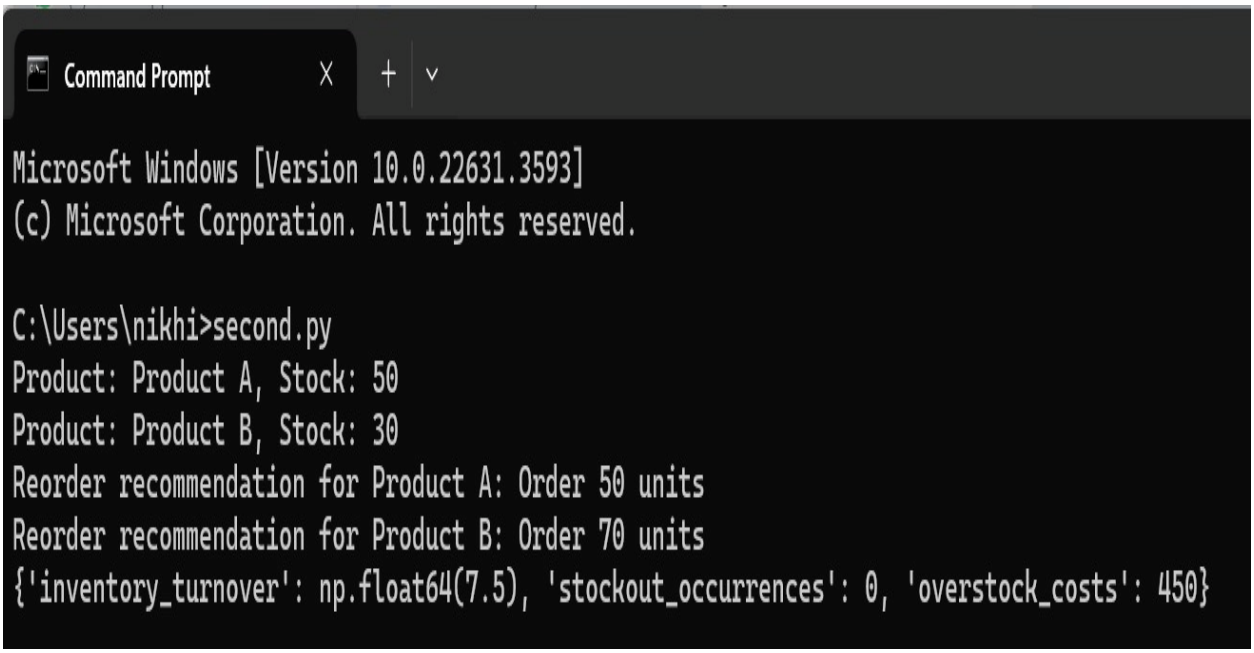
```
import pandas as pd
from datetime import datetime, timedelta
products = [
  {'id': 1, 'name': 'Product A', 'stock': 50, 'reorder_threshold': 20},
  {'id': 2, 'name': 'Product B', 'stock': 30, 'reorder_threshold': 15},
]
sales_data = pd.DataFrame({
  'product_id': [1, 1, 2, 2],
  'quantity': [5, 10, 5, 10],
  'sale_date': [datetime.now() - timedelta(days=i) for i in range(4)]
})
def track_inventory(products):
```

```
    for product in products:
        if product['stock'] < product['reorder_threshold']:
            print(f"Alert: {product['name']} stock is below threshold!")

def calculate_reorder_points(products, sales_data):
    reorder_info = []
    for product in products:
        sales = sales_data[sales_data['product_id'] == product['id']]
        average_demand = sales['quantity'].mean()
        lead_time = 7
        safety_stock = 10
        reorder_point = (average_demand * lead_time) + safety_stock
        target_stock = 100
        reorder_quantity = target_stock - product['stock']
        reorder_info.append({
            'product_id': product['id'],
            'reorder_point': reorder_point,
            'reorder_quantity': reorder_quantity
        })
    return reorder_info
def generate_reports(products, sales_data):
    report = {
        'inventory_turnover': None,
        'stockout_occurrences': None,
        'overstock_costs': None
    }
    cogs = sales_data['quantity'].sum() * 10
    average_inventory = sum([p['stock'] for p in products]) / len(products)
    report['inventory_turnover'] = cogs / average_inventory
    stockouts = [p for p in products if p['stock'] == 0]
    report['stockout_occurrences'] = len(stockouts)
```

```
    overstock_costs = sum([p['stock'] - p['reorder_threshold'] for p in products if p['stock'] > p['reorder_threshold']])
    report['overstock_costs'] = overstock_costs * 10
    return report
def user_interface(products, reorder_info):
    for product in products:
        print(f"Product: {product['name']}, Stock: {product['stock']}")
    for info in reorder_info:
        product = next(p for p in products if p['id'] == info['product_id'])
        print(f"Reorder recommendation for {product['name']}: Order {info['reorder_quantity']} units")
track_inventory(products)
reorder_info = calculate_reorder_points(products, sales_data)
report = generate_reports(products, sales_data)
user_interface(products, reorder_info)
print(report)
```

Sample Output / Screen Shots



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.

Approach:

- **Data Fetching:** Fetch real-time COVID-19 data for Indian states using an API.
- **User Interface:** Create a GUI using Tkinter to accept user input for state codes and display the COVID-19 statistics.
- **Data Visualization:** Use Matplotlib to generate a pie chart of the COVID-19 statistics (cases, recoveries, deaths) and embed it in the Tkinter window

Pseudocode:

1. **Import necessary libraries**
    - tkinter for GUI components
    - requests for fetching data from API

- matplotlib for creating pie charts

2. **Define fetch_covid_data function**
    - Input: state_code (a string representing the state code)
    - Send an HTTP GET request to the API URL to fetch COVID-19 data
    - Check if the request was successful (status code 200)
        - If successful:
            - Parse the JSON response
            - Check if the state_code exists in the data
                - If it exists:
                    - Get the latest date and return the corresponding COVID-19 data
                - If it doesn't exist:
                    - Print an error message and return None
        - If not successful:
            - Print an error message with the status code and response text
            - Return None
3. **Define display_covid_data function**
    - Fetch the state code from the input entry, convert to uppercase, and strip any whitespace
    - If the state code is not empty:
        - Call fetch_covid_data with the state code to get the COVID-19 data
        - If data is fetched successfully:
            - Extract the number of cases, recoveries, and deaths
            - Create a new window to display results
            - Create a pie chart with the extracted data
            - Embed the pie chart in the new window using FigureCanvasTkAgg
        - If data fetch fails:
            - Update the result label with an error message
    - If the state code is empty:
        - Update the result label with a prompt to enter a valid state code
4. **Create the main GUI window using tkinter**
    - Set the window title
    - Add a label prompting the user to enter a state code
    - Add an entry field for the user to input the state code
    - Add a button to fetch and display the data, linking it to display_covid_data
    - Add a label to display results or error messages
5. **Run the main GUI loop to start the application**

1. Import Libraries

- Import tkinter and ttk for the GUI.
- Import requests for fetching data from the API.
- Import matplotlib.pyplot and FigureCanvasTkAgg for plotting and embedding the chart.

2. Function to Fetch COVID-19 Data

- Define fetch_covid_data(state_code):
    - Send a GET request to the API.
    - Check the response status.
    - If successful, parse the JSON response.
    - Extract the latest COVID-19 data for the given state code.
    - Return the data or print an error message if the state code is not found.

3. Function to Display COVID-19 Data

- Define display_covid_data():
    - Retrieve the state code from the user input.
    - Fetch the COVID-19 data for the state code.
    - Extract cases, recoveries, and deaths.
    - Create a new Tkinter window to display the data.
    - Create a pie chart using Matplotlib.
    - Embed the pie chart in the new window.

4. Create Main Tkinter Window

- Set the window title.
- Create labels, entry fields, and buttons for user interaction.
- Define a label to display errors or messages.
- Run the Tkinter main loop.

Detailed explanation of the actual code:

Import Statements

```
import tkinter as tk
from tkinter import ttk
import requests
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
```

- tkinter and ttk are used for creating the graphical user interface (GUI).
- requests is used for making HTTP requests to fetch COVID-19 data.
- matplotlib.pyplot is used for plotting the data.
- FigureCanvasTkAgg integrates Matplotlib with Tkinter.

Fetch COVID-19 Data Function

```
def fetch_covid_data(state_code):
    url = "https://data.covid19india.org/v4/min/timeseries.min.json"
    response = requests.get(url)
    if response.status_code == 200:
        data = response.json()
        if state_code in data:
            latest_date = max(data[state_code]['dates'].keys())
            return data[state_code]['dates'][latest_date]['total']
        else:
            print(f"Error: State code {state_code} not found in data.")
            return None
    else:
        print(f"Error: Unable to fetch data (status code: {response.status_code})")
        print(response.text)
        return None
```

- This function fetch_covid_data takes a state code as input.
- It makes a GET request to the specified URL to fetch COVID-19 data.
- If the request is successful (status code 200), it checks if the provided state code exists in the data.
- It returns the latest available data for that state.
- If the state code is not found or the request fails, it prints an error message and returns None.

Display COVID-19 Data Function

```
def display_covid_data():
    state_code = state_entry.get().strip().upper()
    if state_code:
        covid_data = fetch_covid_data(state_code)
        if covid_data:
            cases = covid_data.get('confirmed', 'N/A')
            recoveries = covid_data.get('recovered', 'N/A')
```

```
        deaths = covid_data.get('deceased', 'N/A')
        result_window = tk.Toplevel(root)
        result_window.title(f"COVID-19 Statistics for {state_code}")
        labels = ['Cases', 'Recoveries', 'Deaths']
        sizes = [cases, recoveries, deaths]
        colors = ['gold', 'lightgreen', 'lightcoral']
        explode = (0.1, 0, 0)
        fig, ax = plt.subplots()
        ax.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%',
               shadow=True, startangle=140)
        ax.axis('equal')
        canvas = FigureCanvasTkAgg(fig, master=result_window)
        canvas.draw()
        canvas.get_tk_widget().grid(column=0, row=0)
        result_window.mainloop()
    else:
        result_label.config(text="Error fetching COVID-19 data.")
else:
    result_label.config(text="Please enter a valid state code.")
```

- This function display_covid_data is triggered when the user clicks the "Fetch Data" button.
- It retrieves the state code from the input field, ensures it is in uppercase, and removes any leading/trailing spaces.
- It calls fetch_covid_data to get the COVID-19 data for the entered state code.
- If data is found, it extracts the number of cases, recoveries, and deaths.
- It creates a new window (Toplevel) to display the data.
- It uses Matplotlib to create a pie chart showing the distribution of cases, recoveries, and deaths.
- The pie chart is embedded in the Tkinter window using FigureCanvasTkAgg.
- If data is not found, it updates the result label with an error message.

Main Tkinter Window Setup

```
root = tk.Tk()
root.title("Real-Time COVID-19 Statistics Tracker")
ttk.Label(root, text="Enter State Code:").grid(column=0, row=0, padx=10, pady=10)
state_entry = ttk.Entry(root)
state_entry.grid(column=1, row=0, padx=10, pady=10)

fetch_button = ttk.Button(root, text="Fetch Data", command=display_covid_data)
fetch_button.grid(column=0, row=1, columnspan=2, padx=10, pady=10)
result_label = ttk.Label(root, text="")
result_label.grid(column=0, row=2, columnspan=2, padx=10, pady=10)
root.mainloop()
```

- This part of the code sets up the main Tkinter window.
- It creates a window titled "Real-Time COVID-19 Statistics Tracker".
- It adds a label, an entry field for the state code, a button to fetch data, and a label to display results.
- The display_covid_data function is linked to the "Fetch Data" button.
- The mainloop method starts the Tkinter event loop, making the GUI responsive.

Assumptions made (if any):

- The API endpoint https://data.covid19india.org/v4/min/timeseries.min.json is reliable and always returns data in the expected format.
- The state codes provided by the user are valid and match those in the API data.
- The user has internet access to fetch the data from the API.

Limitations:

- **API Dependency:** The program relies on the availability and responsiveness of the API.
- **State Code Validation:** There is no validation for state codes beyond checking if they exist in the fetched data.
- **Error Handling:** Limited error handling for cases like network issues or unexpected API responses.
- **Static Data:** The application fetches data only once per request; there is no real-time updating mechanism.

Code:

```
import tkinter as tk
from tkinter import ttk
import requests
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

def fetch_covid_data(state_code):
    url = "https://data.covid19india.org/v4/min/timeseries.min.json"
    response = requests.get(url)
    if response.status_code == 200:
        data = response.json()
        if state_code in data:
            latest_date = max(data[state_code]['dates'].keys())
            return data[state_code]['dates'][latest_date]['total']
        else:
            print(f"Error: State code {state_code} not found in data.")
            return None
    else:
        print(f"Error: Unable to fetch data (status code: {response.status_code})")
        print(response.text)
        return None

def display_covid_data():
    state_code = state_entry.get().strip().upper()
    if state_code:
        covid_data = fetch_covid_data(state_code)
        if covid_data:
            cases = covid_data.get('confirmed', 'N/A')
            recoveries = covid_data.get('recovered', 'N/A')
            deaths = covid_data.get('deceased', 'N/A')
```

```
            result_window = tk.Toplevel(root)
            result_window.title(f"COVID-19 Statistics for {state_code}")
            labels = ['Cases', 'Recoveries', 'Deaths']
            sizes = [cases, recoveries, deaths]
            colors = ['gold', 'lightgreen', 'lightcoral']
            explode = (0.1, 0, 0)
            fig, ax = plt.subplots()
            ax.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%',
                   shadow=True, startangle=140)
            ax.axis('equal')
            canvas = FigureCanvasTkAgg(fig, master=result_window)
            canvas.draw()
            canvas.get_tk_widget().grid(column=0, row=0)
            result_window.mainloop()
        else:
            result_label.config(text="Error fetching COVID-19 data.")
    else:
        result_label.config(text="Please enter a valid state code.")
root = tk.Tk()
root.title("Real-Time COVID-19 Statistics Tracker")
ttk.Label(root, text="Enter State Code:").grid(column=0, row=0, padx=10, pady=10)
state_entry = ttk.Entry(root)
state_entry.grid(column=1, row=0, padx=10, pady=10)

fetch_button = ttk.Button(root, text="Fetch Data", command=display_covid_data)
fetch_button.grid(column=0, row=1, columnspan=2, padx=10, pady=10)
result_label = ttk.Label(root, text="")
result_label.grid(column=0, row=2, columnspan=2, padx=10, pady=10)
root.mainloop()
```

Sample Output / Screen Shots