**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., 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.**

**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:**

**The project aims to create a graphical user interface (GUI) application that allows users to monitor real-time weather data for any city in the world. Users can select a country from a dropdown list and then input a city name to retrieve weather information such as temperature, humidity, wind speed, and forecast data. The program also includes the functionality to display weather data graphically using plots.**

**Pseudocode:**

1. **Initialize the Application:**
   * **Import necessary libraries (tkinter, requests, PIL, matplotlib).**
   * **Define API keys and endpoint URLs for weather and geographical data.**
   * **Create the main window (tkinter root) and set up the initial layout (labels, dropdowns, and entry fields).**
2. **Fetch Country Data:**
   * **Make an API request to retrieve a list of countries and their corresponding codes.**
   * **Populate the country dropdown list with these values.**
3. **Handle Country Selection:**
   * **When a country is selected, fetch the list of cities in that country using an API.**
   * **Allow the user to input a city name.**
4. **Fetch and Display Weather Data:**
   * **When the user enters a city name and clicks the "Get Weather" button:**
     + **Make an API request to fetch current weather data for the selected city.**
     + **Display the weather information in the GUI.**
     + **Update the background color based on the weather condition.**
     + **Fetch and display the weather icon.**
5. **Plot Temperature Data:**
   * **Fetch a 5-day weather forecast for the selected city.**
   * **Plot the temperature data using matplotlib and display it in a separate window.**
6. **Handle Errors:**
   * **Implement error handling for invalid city names, missing API keys, or network issues.**
   * **Display appropriate error messages to the user.**

**Detailed explanation of the actual code:**

1. **Initialization:**
   * **tkinter is used to create the GUI, including input fields, buttons, and labels.**
   * **requests is used to interact with the OpenWeatherMap API and GeoDB Cities API for retrieving weather and geographical data.**
   * **PIL (Pillow) is used to handle weather icons.**
   * **matplotlib is used to plot temperature data for the forecast.**
2. **Fetching Country Data:**
   * **An API call to the GeoDB Cities API retrieves a list of all available countries and their corresponding ISO codes.**
   * **This data is stored in a dictionary and used to populate the country dropdown.**
3. **Handling Country Selection and City Input:**
   * **When the user selects a country, another API call retrieves cities in that country.**
   * **The user can enter a city name, which is then validated before making an API call to fetch weather data.**
4. **Fetching and Displaying Weather Data:**
   * **The weather data is fetched using the OpenWeatherMap API by providing the city name and country code.**
   * **The response data includes temperature, humidity, wind speed, weather conditions, and more.**
   * **The GUI displays this information, and the background color dynamically changes based on the weather condition (e.g., blue for clear skies, gray for clouds).**
5. **Plotting Temperature Data:**
   * **A 5-day forecast is fetched from the OpenWeatherMap API.**
   * **Temperature data is extracted and plotted using matplotlib.**
   * **The plot is displayed in a new window, showing temperature trends over the next five days.**
6. **Error Handling:**
   * **The code includes error handling for common issues such as invalid API keys, non-existent cities, and network errors.**
   * **Users are notified of issues through dialog boxes.**

**Assumptions made (if any):**

* **The user has a stable internet connection, as the program relies on real-time data fetched from APIs.**
* **The tkinter, requests, PIL, and matplotlib libraries are installed and accessible.**
* **The API keys for OpenWeatherMap and GeoDB Cities are valid and have the necessary permissions.**
* **The user is familiar with the concept of selecting countries and cities for weather data retrieval.**

**Limitations:**

* **API Limitations: The application is dependent on third-party APIs (OpenWeatherMap, GeoDB Cities), which may have usage limits or downtimes.**
* **City Availability: Not all cities may be available in the GeoDB Cities API, especially smaller towns or non-standard locations.**
* **Performance: Fetching and processing data for large numbers of cities or plotting extensive datasets may result in performance issues.**
* **UI Complexity: The user interface is relatively simple and may not scale well with additional features or larger datasets.**
* **Error Handling: While common errors are handled, there may be edge cases (e.g., temporary API outages) that are not fully addressed.**
* **Timezones: The application assumes UTC for time-related data, which may not always match the user's local timezone.**

**Code:**

**import tkinter as tk**

**from tkinter import ttk, messagebox**

**import requests**

**from datetime import datetime, timezone**

**from PIL import Image, ImageTk**

**import matplotlib.pyplot as plt**

**from io import BytesIO**

**# Function to get current weather data from OpenWeatherMap API**

**def get\_weather(city\_name, country\_code, api\_key):**

**base\_url = "http://api.openweathermap.org/data/2.5/weather?"**

**complete\_url = f"{base\_url}q={city\_name},{country\_code}&appid={api\_key}&units=metric"**

**try:**

**response = requests.get(complete\_url)**

**response.raise\_for\_status() # Raises an HTTPError for bad responses**

**data = response.json()**

**main = data['main']**

**wind = data['wind']**

**weather = data['weather'][0]**

**sys = data['sys']**

**visibility = data.get('visibility', 'N/A') / 1000 # Convert to kilometers**

**clouds = data['clouds']['all']**

**# Determine weather condition for dynamic background**

**condition = weather['main'].lower()**

**update\_background(condition)**

**# Build the weather report string**

**weather\_report = (**

**f"City: {city\_name}, {sys['country']}\n"**

**f"Temperature: {main['temp']}°C\n"**

**f"Feels Like: {main['feels\_like']}°C\n"**

**f"Min Temperature: {main['temp\_min']}°C\n"**

**f"Max Temperature: {main['temp\_max']}°C\n"**

**f"Humidity: {main['humidity']}%\n"**

**f"Pressure: {main['pressure']} hPa\n"**

**f"Wind Speed: {wind['speed']} m/s\n"**

**f"Wind Direction: {wind['deg']}°\n"**

**f"Weather: {weather['main']} ({weather['description']})\n"**

**f"Visibility: {visibility} km\n"**

**f"Cloudiness: {clouds}%\n"**

**f"Sunrise: {format\_time\_with\_timezone(sys['sunrise'])}\n"**

**f"Sunset: {format\_time\_with\_timezone(sys['sunset'])}"**

**)**

**return weather\_report, weather['icon']**

**except requests.exceptions.HTTPError as http\_err:**

**if response.status\_code == 404:**

**return "City Not Found.", None**

**elif response.status\_code == 401:**

**return "Invalid API Key.", None**

**else:**

**return f"HTTP error occurred: {http\_err}", None**

**except Exception as err:**

**return f"An error occurred: {err}", None**

**# Function to get 5-day forecast data from OpenWeatherMap API**

**def get\_forecast(city\_name, country\_code, api\_key):**

**base\_url = "http://api.openweathermap.org/data/2.5/forecast?"**

**complete\_url = f"{base\_url}q={city\_name},{country\_code}&appid={api\_key}&units=metric"**

**try:**

**response = requests.get(complete\_url)**

**response.raise\_for\_status()**

**data = response.json()**

**# Extract temperature and time data for plotting**

**temps = [entry['main']['temp'] for entry in data['list']]**

**times = [entry['dt'] for entry in data['list']]**

**# Convert Unix timestamps to formatted time strings**

**formatted\_times = [datetime.fromtimestamp(t, timezone.utc).strftime('%Y-%m-%d %H:%M') for t in times]**

**return temps, formatted\_times**

**except requests.exceptions.HTTPError as http\_err:**

**print(f"HTTP error occurred: {http\_err}")**

**return [], []**

**except Exception as err:**

**print(f"An error occurred: {err}")**

**return [], []**

**# Function to format the time from Unix format using timezone-aware datetime objects**

**def format\_time\_with\_timezone(unix\_time):**

**# Convert the unix timestamp to a timezone-aware datetime object**

**utc\_time = datetime.fromtimestamp(unix\_time, timezone.utc)**

**# Format the time in the desired format**

**return utc\_time.strftime('%Y-%m-%d %H:%M:%S')**

**# Function to show the weather icon in the GUI**

**def show\_icon(icon\_code):**

**try:**

**icon\_url = f"http://openweathermap.org/img/wn/{icon\_code}@2x.png"**

**icon\_image = Image.open(requests.get(icon\_url, stream=True).raw)**

**icon\_photo = ImageTk.PhotoImage(icon\_image)**

**icon\_label.config(image=icon\_photo)**

**icon\_label.image = icon\_photo**

**except Exception as e:**

**print(f"Error loading icon: {e}")**

**# Function to update the background color based on weather condition**

**def update\_background(condition):**

**if 'clear' in condition:**

**root.config(bg='#87CEEB') # Clear sky - light blue**

**elif 'cloud' in condition:**

**root.config(bg='#B0C4DE') # Cloudy - light steel blue**

**elif 'rain' in condition or 'drizzle' in condition:**

**root.config(bg='#778899') # Rainy - light slate gray**

**elif 'snow' in condition:**

**root.config(bg='#F0F8FF') # Snowy - Alice blue**

**else:**

**root.config(bg='#708090') # Default - slate gray**

**# Function to clear the inputs**

**def clear\_input():**

**city\_entry.delete(0, tk.END)**

**country\_entry.delete(0, tk.END)**

**root.config(bg=default\_bg)**

**icon\_label.config(image='')**

**# Function to show the weather report**

**def show\_weather():**

**city\_name = city\_entry.get()**

**country\_code = country\_entry.get().upper() # Convert to uppercase for standardization**

**if city\_name and country\_code:**

**weather\_report, icon\_code = get\_weather(city\_name, country\_code, api\_key)**

**if icon\_code:**

**show\_icon(icon\_code)**

**messagebox.showinfo("Weather Report", weather\_report)**

**else:**

**messagebox.showwarning("Input Error", "Please enter both a city and country code.")**

**# Function to plot temperature data**

**def plot\_temperature():**

**city\_name = city\_entry.get()**

**country\_code = country\_entry.get().upper()**

**if city\_name and country\_code:**

**temps, times = get\_forecast(city\_name, country\_code, api\_key)**

**if temps and times:**

**plt.figure(figsize=(10, 6))**

**plt.plot(times, temps, marker='o', linestyle='-', color='b')**

**plt.title(f"5-Day Temperature Forecast for {city\_name}, {country\_code}")**

**plt.xlabel('Date and Time')**

**plt.ylabel('Temperature (°C)')**

**plt.xticks(rotation=45)**

**plt.tight\_layout()**

**plt.show()**

**else:**

**messagebox.showerror("Error", "Could not retrieve forecast data.")**

**else:**

**messagebox.showwarning("Input Error", "Please enter both a city and country code.")**

**# API Key (replace 'your\_api\_key\_here' with your actual API key)**

**api\_key = "a2361ffa0c07dcf3b94f9d6197fd0213"**

**# Creating the main window**

**root = tk.Tk()**

**root.title("Weather Monitoring System")**

**root.geometry("400x550")**

**# Default background color**

**default\_bg = '#F5F5F5'**

**root.config(bg=default\_bg)**

**# Display current date and time**

**current\_time\_label = ttk.Label(root, text=f"Current Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}", background=default\_bg)**

**current\_time\_label.pack(pady=5)**

**# Label for the country entry box**

**country\_label = ttk.Label(root, text="Enter Country Code (e.g., US for United States):", background=default\_bg)**

**country\_label.pack(pady=10)**

**# Entry box for country code**

**country\_entry = ttk.Entry(root)**

**country\_entry.pack(pady=10)**

**# Label for the city entry box**

**city\_label = ttk.Label(root, text="Enter City Name:", background=default\_bg)**

**city\_label.pack(pady=10)**

**# Entry box for city name**

**city\_entry = ttk.Entry(root)**

**city\_entry.pack(pady=10)**

**# Button to fetch and display weather**

**weather\_button = ttk.Button(root, text="Show Weather", command=show\_weather)**

**weather\_button.pack(pady=10)**

**# Label to display weather icon**

**icon\_label = ttk.Label(root, background=default\_bg)**

**icon\_label.pack(pady=10)**

**# Button to plot temperature graph**

**plot\_button = ttk.Button(root, text="Plot Temperature", command=plot\_temperature)**

**plot\_button.pack(pady=10)**

**# Button to clear the input**

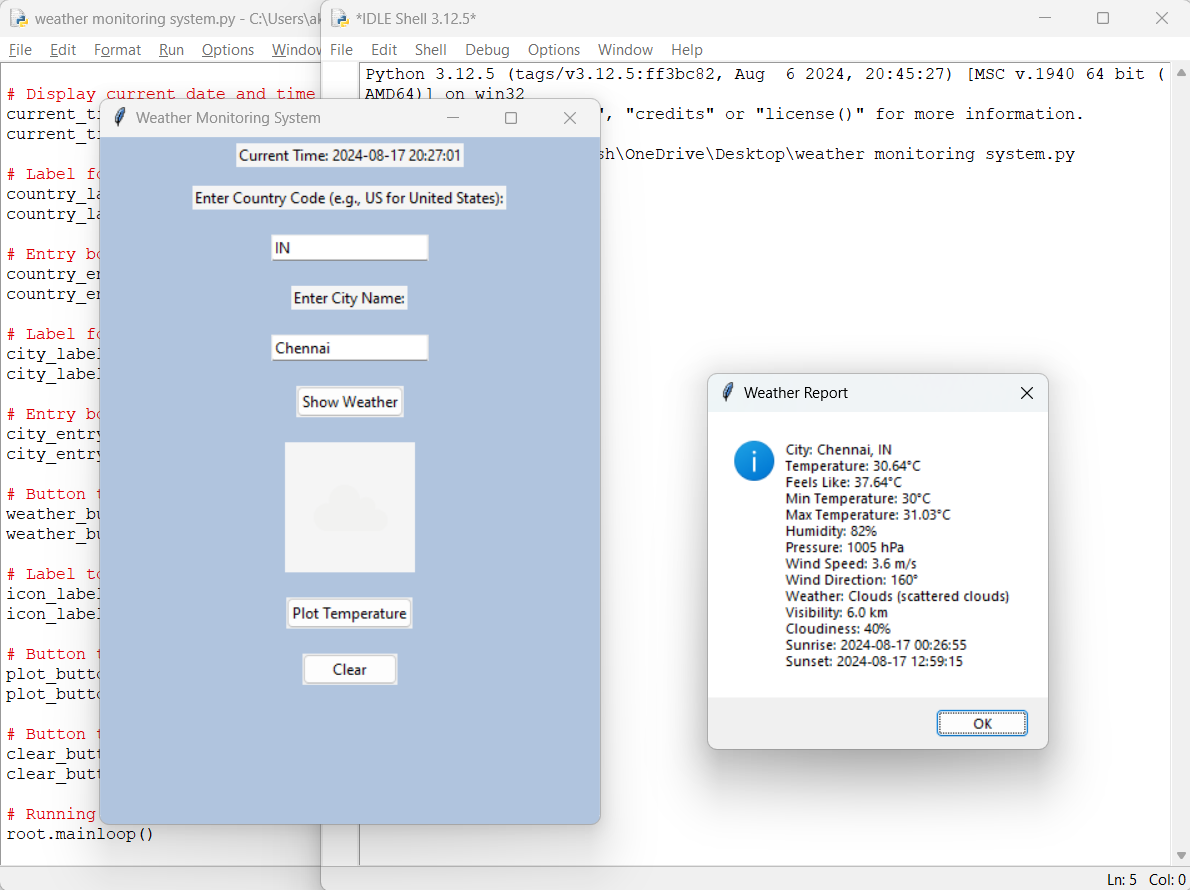
**clear\_button = ttk.Button(root, text="Clear", command=clear\_input)**

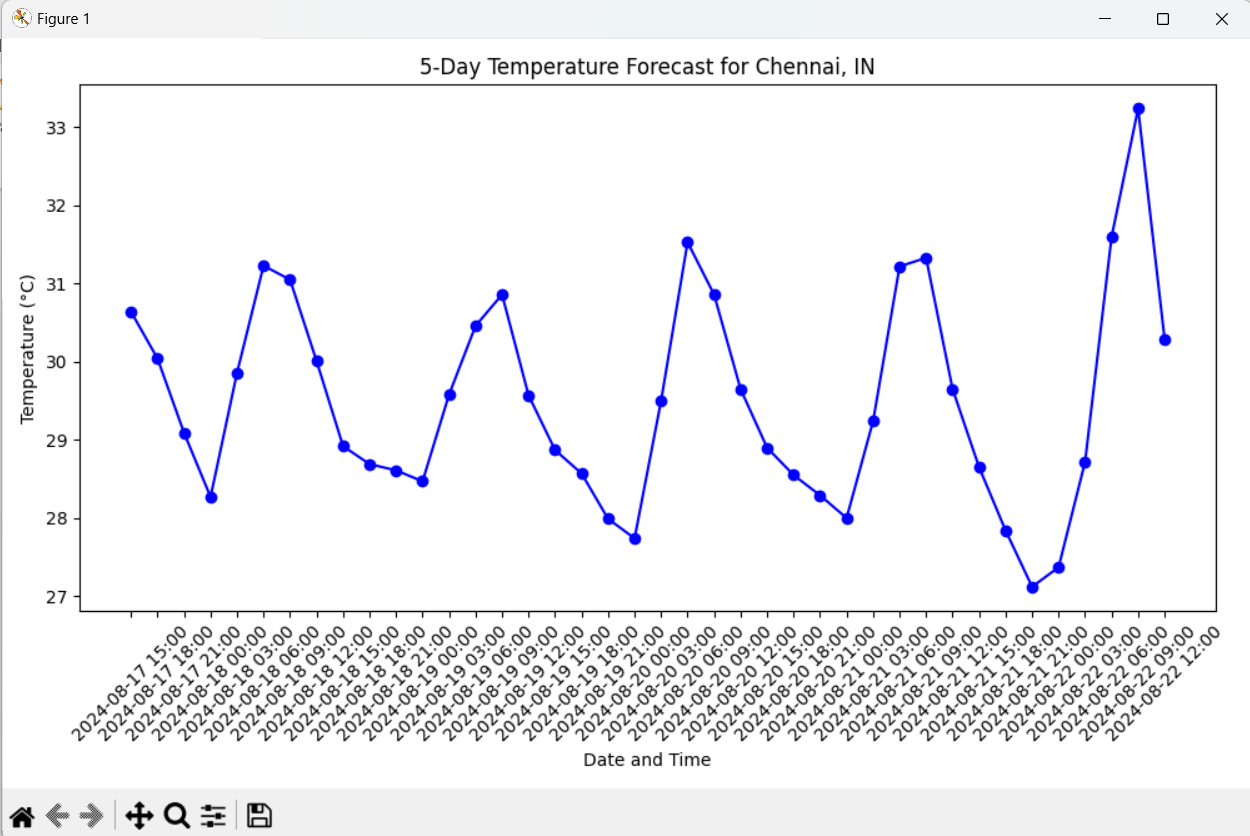
**clear\_button.pack(pady=10)**

**# Running the GUI application**

**root.mainloop()**

**Sample Output / Screen Shots**





**Problem 2: Inventory Management System Optimization**

**Scenario:**

You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.

**Tasks:**

1. **Model the inventory system**: Define the structure of the inventory system, including products, warehouses, and current stock levels.
2. **Implement an inventory tracking application**: Develop a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain threshold.
3. **Optimize inventory ordering**: Implement algorithms to calculate optimal reorder points and quantities based on historical sales data, lead times, and demand forecasts.
4. **Generate reports**: Provide reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.
5. **User interaction**: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

**Deliverables:**

* **Data Flow Diagram**: Illustrate how data flows within the inventory management system, from input (e.g., sales data, inventory adjustments) to output (e.g., reorder alerts, reports).
* **Pseudocode and Implementation**: Provide pseudocode and actual code demonstrating how inventory levels are tracked, reorder points are calculated, and reports are generated.
* **Documentation**: Explain the algorithms used for reorder optimization, how historical data influences decisions, and any assumptions made (e.g., constant lead times).
* **User Interface**: Develop a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts.
* **Assumptions and Improvements**: Discuss assumptions about demand patterns, supplier reliability, and potential improvements for the inventory management system's efficiency and accuracy.

**Approach:**

1. **Model the Inventory System:**
   * **Define classes for products, warehouses, and inventory management.**
   * **Structure includes products with attributes like ID, name, price, daily demand, and lead time.**
   * **Warehouses track stock levels and handle stock adjustments.**
2. **Implement Inventory Tracking Application:**
   * **Use Python with Tkinter for a graphical user interface.**
   * **Track real-time inventory levels and alert users when stock falls below a threshold.**
   * **Provide functionalities for adding/removing stock, checking stock levels, and generating reports.**
3. **Optimize Inventory Ordering:**
   * **Implement algorithms to calculate reorder points and quantities based on daily demand, lead times, and safety stock.**
   * **Use historical sales data and demand forecasts to refine these calculations.**
4. **Generate Reports:**
   * **Create reports on inventory turnover rates, stockout occurrences, and overstock costs.**
   * **Use JSON to format reports for easy readability.**
5. **User Interaction:**
   * **Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.**
   * **Provide options for checking stock levels, generating detailed reports, and retrieving product information.**

**Pseudocode:**

**Class Product:**

**Initialize product with ID, name, price, daily\_demand, lead\_time**

**Class Warehouse:**

**Initialize warehouse with location**

**Add stock**

**Remove stock**

**Get stock level**

**Check if below threshold**

**Class InventorySystem:**

**Initialize inventory system**

**Add product**

**Add warehouse**

**Add stock**

**Remove stock**

**Check stock levels and generate alerts**

**Generate report with stock levels, prices, and total value**

**Generate detailed report including reorder points and order quantities**

**Get product info based on ID or name**

**Function calculate\_reorder\_point(daily\_demand, lead\_time):**

**Return daily\_demand \* lead\_time**

**Function calculate\_order\_quantity(daily\_demand, lead\_time, safety\_stock):**

**Return reorder\_point + safety\_stock**

**Class InventoryApp:**

**Initialize GUI with buttons for checking stock, generating reports, and getting product info**

**Handle button actions**

**Check stock levels**

**Generate reports**

**Get product info**

**Exit application**

**Function main():**

**Create Tkinter root window**

**Create InventoryApp instance**

**Start Tkinter main loop**

**Detailed explanation of the actual code:**

1. **Classes Definition:**
   * **Product: Represents an inventory item with attributes such as ID, name, price, daily demand, and lead time.**
   * **Warehouse: Manages stock levels with methods to add or remove stock, check current stock levels, and determine if stock is below a threshold.**
   * **InventorySystem: Manages products and warehouses, including methods for adding and removing stock, checking stock levels, and generating reports.**
2. **Inventory Tracking Application:**
   * **GUI: Created using Tkinter. The interface includes buttons for checking stock levels, generating reports, and retrieving product information.**
   * **Button Actions: Each button triggers a method in InventoryApp to perform the corresponding action, such as displaying stock levels or generating reports.**
3. **Optimization Algorithms:**
   * **Reorder Point Calculation: calculate\_reorder\_point multiplies daily demand by lead time.**
   * **Order Quantity Calculation: calculate\_order\_quantity adds safety stock to the reorder point.**
4. **Report Generation:**
   * **Basic Report: Includes stock levels, prices, and total value of inventory.**
   * **Detailed Report: Adds reorder points, reorder quantities, turnover rates, and stockout occurrences.**
5. **User Interaction:**
   * **Users can interact with the system via a graphical interface, inputting product IDs or names to view detailed information and receive alerts.**

**Assumptions made (if any):**

1. **Constant Lead Times: Lead times are assumed to be constant and do not vary.**
2. **Safety Stock Level: A default safety stock level of 10 units is used for reorder calculations.**
3. **Simplified Turnover Rate and Stockouts: Turnover rate and stockout occurrences are simulated and do not reflect real data.**

**Limitations:**

1. **Simplified Demand Forecasting: The model does not account for variability or trends in demand.**
2. **Static Safety Stock: The safety stock level is fixed and does not adjust based on historical data or demand fluctuations.**
3. **No Real-Time Data Integration: The application does not integrate with real-time sales data or inventory systems.**
4. **Simulated Reports: Turnover rate and stockout occurrences are simulated, which may not accurately represent actual business conditions.**

**Code:**

**import json**

**import random**

**import tkinter as tk**

**from tkinter import messagebox, simpledialog**

**# Product class to represent an inventory item**

**class Product:**

**def \_\_init\_\_(self, product\_id, name, price, daily\_demand, lead\_time):**

**self.product\_id = product\_id**

**self.name = name**

**self.price = price**

**self.daily\_demand = daily\_demand**

**self.lead\_time = lead\_time**

**# Warehouse class to manage stock levels**

**class Warehouse:**

**def \_\_init\_\_(self, location):**

**self.location = location**

**self.stock = {}**

**def add\_stock(self, product, quantity):**

**self.stock[product.product\_id] = self.stock.get(product.product\_id, 0) + quantity**

**def remove\_stock(self, product, quantity):**

**if product.product\_id in self.stock and self.stock[product.product\_id] >= quantity:**

**self.stock[product.product\_id] -= quantity**

**else:**

**print(f"Not enough stock for {product.name}.")**

**def get\_stock\_level(self, product):**

**return self.stock.get(product.product\_id, 0)**

**def is\_below\_threshold(self, product, threshold):**

**return self.get\_stock\_level(product) < threshold**

**# Inventory system class to manage products and warehouses**

**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.location] = warehouse**

**def add\_stock(self, location, product\_id, quantity):**

**if location in self.warehouses and product\_id in self.products:**

**self.warehouses[location].add\_stock(self.products[product\_id], quantity)**

**def remove\_stock(self, location, product\_id, quantity):**

**if location in self.warehouses and product\_id in self.products:**

**self.warehouses[location].remove\_stock(self.products[product\_id], quantity)**

**def check\_stock\_levels(self, threshold):**

**alerts = []**

**for warehouse in self.warehouses.values():**

**for product\_id, quantity in warehouse.stock.items():**

**if quantity < threshold:**

**alerts.append(f"Low stock alert for {self.products[product\_id].name} in {warehouse.location}: {quantity} remaining.")**

**return alerts**

**def generate\_report(self):**

**report = {}**

**for warehouse in self.warehouses.values():**

**for product\_id, quantity in warehouse.stock.items():**

**product = self.products[product\_id]**

**report[product.name] = {**

**'Stock Level': quantity,**

**'Price': product.price,**

**'Total Value': quantity \* product.price**

**}**

**return report**

**def generate\_detailed\_report(self):**

**report = self.generate\_report()**

**for product in self.products.values():**

**reorder\_point = calculate\_reorder\_point(product.daily\_demand, product.lead\_time)**

**order\_quantity = calculate\_order\_quantity(product.daily\_demand, product.lead\_time, 10) # Safety stock of 10**

**report[product.name]['Reorder Point'] = reorder\_point**

**report[product.name]['Reorder Quantity'] = order\_quantity**

**turnover\_rate = random.uniform(0.1, 1.0) # Simulated turnover rate**

**stockout\_occurrences = random.randint(0, 5) # Simulated stockout occurrences**

**report['Turnover Rate'] = turnover\_rate**

**report['Stockout Occurrences'] = stockout\_occurrences**

**return report**

**def get\_product\_info(self, product\_id\_or\_name):**

**"""Retrieve product information based on ID or name."""**

**if product\_id\_or\_name.isdigit():**

**product\_id = int(product\_id\_or\_name)**

**product = self.products.get(product\_id)**

**if product:**

**stock\_level = self.warehouses["Main Warehouse"].get\_stock\_level(product)**

**reorder\_point = calculate\_reorder\_point(product.daily\_demand, product.lead\_time)**

**return f"Product: {product.name}\nStock Level: {stock\_level}\nReorder Point: {reorder\_point}"**

**else:**

**return "Product ID not found."**

**else:**

**for product in self.products.values():**

**if product.name.lower() == product\_id\_or\_name.lower():**

**stock\_level = self.warehouses["Main Warehouse"].get\_stock\_level(product)**

**reorder\_point = calculate\_reorder\_point(product.daily\_demand, product.lead\_time)**

**return f"Product: {product.name}\nStock Level: {stock\_level}\nReorder Point: {reorder\_point}"**

**return "Product name not found."**

**def calculate\_reorder\_point(daily\_demand, lead\_time):**

**return daily\_demand \* lead\_time**

**def calculate\_order\_quantity(daily\_demand, lead\_time, safety\_stock):**

**reorder\_point = calculate\_reorder\_point(daily\_demand, lead\_time)**

**return reorder\_point + safety\_stock**

**# GUI Application Class**

**class InventoryApp:**

**def \_\_init\_\_(self, master):**

**self.master = master**

**self.master.title("Inventory Management System")**

**self.inventory\_system = InventorySystem()**

**# Add products**

**product1 = Product(1, "Product A", 10.0, 5, 7) # Product ID, Name, Price, Daily Demand, Lead Time**

**product2 = Product(2, "Product B", 15.0, 3, 10)**

**self.inventory\_system.add\_product(product1)**

**self.inventory\_system.add\_product(product2)**

**# Add warehouse**

**warehouse = Warehouse("Main Warehouse")**

**self.inventory\_system.add\_warehouse(warehouse)**

**# Add stock**

**self.inventory\_system.add\_stock("Main Warehouse", 1, 100)**

**self.inventory\_system.add\_stock("Main Warehouse", 2, 50)**

**# Create buttons**

**self.check\_stock\_button = tk.Button(master, text="Check Stock Levels", command=self.check\_stock\_levels)**

**self.check\_stock\_button.pack(pady=10)**

**self.generate\_report\_button = tk.Button(master, text="Generate Inventory Report", command=self.generate\_report)**

**self.generate\_report\_button.pack(pady=10)**

**self.product\_info\_button = tk.Button(master, text="Get Product Info", command=self.get\_product\_info)**

**self.product\_info\_button.pack(pady=10)**

**self.exit\_button = tk.Button(master, text="Exit", command=self.exit\_application)**

**self.exit\_button.pack(pady=10)**

**# Override window close button behavior**

**self.master.protocol("WM\_DELETE\_WINDOW", self.exit\_application)**

**def check\_stock\_levels(self):**

**threshold = simpledialog.askinteger("Input", "Enter stock threshold:", minvalue=0)**

**if threshold is not None:**

**alerts = self.inventory\_system.check\_stock\_levels(threshold)**

**if alerts:**

**messagebox.showinfo("Stock Alerts", "\n".join(alerts))**

**else:**

**messagebox.showinfo("Stock Alerts", "All products are above the threshold.")**

**def generate\_report(self):**

**report = self.inventory\_system.generate\_detailed\_report()**

**report\_str = json.dumps(report, indent=4)**

**messagebox.showinfo("Inventory Report", report\_str)**

**def get\_product\_info(self):**

**product\_id\_or\_name = simpledialog.askstring("Input", "Enter Product ID or Name:")**

**if product\_id\_or\_name:**

**info = self.inventory\_system.get\_product\_info(product\_id\_or\_name)**

**messagebox.showinfo("Product Info", info)**

**def exit\_application(self):**

**self.master.destroy()**

**# Main function to run the GUI application**

**def main():**

**root = tk.Tk()**

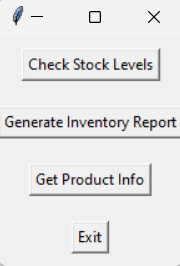
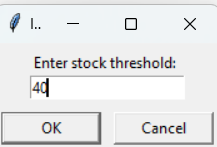
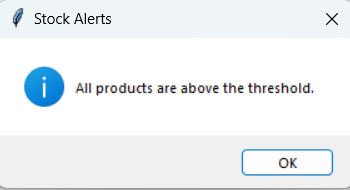
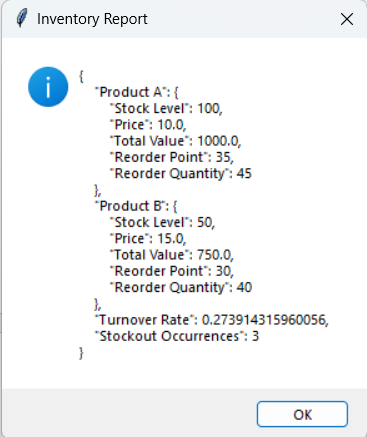
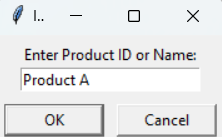
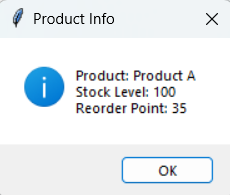
**app = InventoryApp(root)**

**root.mainloop()**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Sample Output / Screen Shots**

**Problem 3: Real-Time Traffic Monitoring System**

**Scenario:**

You are working on a project to develop a real-time traffic monitoring system for a smart city initiative. The system should provide real-time traffic updates and suggest alternative routes.

**Tasks:**

1. **Model the data flow for fetching real-time traffic information from an external API and displaying it to the user.**
2. **Implement a Python application that integrates with a traffic monitoring API (e.g., Google Maps Traffic API) to fetch real-time traffic data.**
3. **Display current traffic conditions, estimated travel time, and any incidents or delays.**
4. **Allow users to input a starting point and destination to receive traffic updates and alternative routes.**

**Deliverables:**

* Data flow diagram illustrating the interaction between the application and the API.
* Pseudocode and implementation of the traffic monitoring system.
* Documentation of the API integration and the methods used to fetch and display traffic data.
* Explanation of any assumptions made and potential improvements.

**Approach:**

1. **Model the Data Flow:**
   * **User Input: User provides a starting point and a destination.**
   * **Geocoding: Convert the user input addresses into latitude and longitude using the Geoapify Geocoding API.**
   * **Routing: Fetch real-time traffic and routing information using the Geoapify Routing API.**
   * **Display Results: Parse and display the traffic information, including estimated travel time, distance, and traffic incidents (if available).**
2. **Implement the Python Application:**
   * **Geocoding Functionality: Fetch coordinates for addresses.**
   * **Routing Functionality: Fetch routing data and parse it for traffic information.**
   * **GUI: Create an interface for user input and display results.**

**Pseudocode:**

**# Constants**

**API\_KEY = 'your\_geoapify\_api\_key'**

**GEOCODING\_URL = 'https://api.geoapify.com/v1/geocode/search'**

**ROUTING\_URL = 'https://api.geoapify.com/v1/routing'**

**# Function to get coordinates for an address**

**function get\_coordinates(address):**

**params = {'text': address, 'apiKey': API\_KEY}**

**response = request(GET, GEOCODING\_URL, params)**

**if response is successful:**

**data = parse(response)**

**if data has features:**

**coordinates = extract coordinates**

**return latitude, longitude**

**raise error**

**# Function to fetch traffic data**

**function fetch\_traffic\_data(origin, destination):**

**params = {'waypoints': format\_waypoints(origin, destination), 'mode': 'drive', 'apiKey': API\_KEY}**

**response = request(GET, ROUTING\_URL, params)**

**if response is successful:**

**return parse(response)**

**raise error**

**# Function to parse traffic data**

**function parse\_traffic\_data(data):**

**if data has features:**

**route = extract\_route\_properties(data)**

**travel\_time = calculate\_travel\_time(route)**

**distance = calculate\_distance(route)**

**return {'travel\_time': travel\_time, 'distance': distance, 'incidents': 'No incidents available'}**

**raise error**

**# Function to display traffic information**

**function display\_traffic\_info(info):**

**format and display information in GUI**

**# Main function to get and display traffic information**

**function get\_traffic\_info():**

**origin = get user input for starting point**

**destination = get user input for destination**

**origin\_coords = get\_coordinates(origin)**

**destination\_coords = get\_coordinates(destination)**

**traffic\_data = fetch\_traffic\_data(origin\_coords, destination\_coords)**

**traffic\_info = parse\_traffic\_data(traffic\_data)**

**display\_traffic\_info(traffic\_info)**

**# GUI setup**

**create GUI elements**

**set button to trigger get\_traffic\_info function**

**start GUI event loop**

**Detailed explanation of the actual code:**

1. **Import Statements:**
   * **tkinter for GUI components.**
   * **requests for making HTTP requests to the APIs.**
2. **Constants:**
   * **API\_KEY is used for authentication with the Geoapify API.**
   * **GEOCODING\_URL and ROUTING\_URL are endpoints for the Geoapify APIs.**
3. **get\_coordinates(address):**
   * **Makes a GET request to the Geocoding API to convert an address to latitude and longitude.**
   * **Returns coordinates if the address is found; raises an exception if not.**
4. **fetch\_traffic\_data(origin, destination):**
   * **Makes a GET request to the Routing API with the coordinates of origin and destination.**
   * **Returns the traffic data if successful; raises an exception if not.**
5. **parse\_traffic\_data(data):**
   * **Parses the JSON response from the Routing API.**
   * **Extracts travel time and distance; traffic incident information is not available from Geoapify and is set as a placeholder.**
6. **display\_traffic\_info(info):**
   * **Uses messagebox to show traffic information in a popup dialog.**
7. **get\_traffic\_info():**
   * **Retrieves user input, converts addresses to coordinates, fetches traffic data, parses it, and displays the results.**
8. **GUI Setup:**
   * **Creates labels, entry fields, and a button using tkinter.**
   * **The button triggers the get\_traffic\_info function when clicked.**
   * **Starts the GUI event loop to make the application responsive.**

**Assumptions made (if any):**

1. **API Key: The API key is valid and has sufficient quota for requests.**
2. **Data Availability: Geoapify APIs provide the required data (though detailed traffic incidents are not available).**
3. **User Input: Users provide valid addresses that can be geocoded.**

**Limitations:**

1. **Traffic Incidents: The Geoapify Routing API does not provide detailed traffic incidents or real-time updates, so this information is represented as a placeholder.**
2. **Error Handling: Basic error handling is implemented, but more robust handling might be needed for production use.**
3. **GUI: The GUI is minimal and may need additional features or enhancements for better user experience.**

**Code:**

**import tkinter as tk**

**from tkinter import messagebox**

**import requests**

**# Constants**

**API\_KEY = '87d451e4365441b79e31dc4cd63f532c' # Replace with your Geoapify API key**

**GEOCODING\_URL = 'https://api.geoapify.com/v1/geocode/search'**

**ROUTING\_URL = 'https://api.geoapify.com/v1/routing'**

**def get\_coordinates(address):**

**"""**

**Fetch the latitude and longitude for a given address using Geoapify Geocoding API.**

**"""**

**params = {**

**'text': address,**

**'apiKey': API\_KEY**

**}**

**response = requests.get(GEOCODING\_URL, params=params)**

**if response.status\_code == 200:**

**data = response.json()**

**if data['features']:**

**coordinates = data['features'][0]['geometry']['coordinates']**

**return coordinates[1], coordinates[0] # Return lat, lon**

**else:**

**raise Exception("Address not found")**

**else:**

**raise Exception(f"Error fetching coordinates: {response.status\_code}")**

**def fetch\_traffic\_data(origin, destination):**

**"""**

**Fetch routing data from Geoapify Routing API.**

**"""**

**params = {**

**'waypoints': f'{origin[0]},{origin[1]}|{destination[0]},{destination[1]}',**

**'mode': 'drive',**

**'apiKey': API\_KEY**

**}**

**response = requests.get(ROUTING\_URL, params=params)**

**if response.status\_code == 200:**

**return response.json()**

**else:**

**raise Exception(f"Error fetching data: {response.status\_code}")**

**def parse\_traffic\_data(data):**

**"""**

**Parse the routing data returned by the Geoapify Routing API.**

**"""**

**if 'features' not in data or not data['features']:**

**raise Exception("No routes found")**

**route = data['features'][0]['properties']**

**# Calculate estimated travel time and distance**

**travel\_time\_minutes = route['time'] // 60 # Time in minutes**

**distance\_km = route['distance'] / 1000 # Distance in kilometers**

**# Geoapify does not provide traffic incidents; this will show estimated time and distance only**

**return {**

**'travel\_time': f'{travel\_time\_minutes} minutes',**

**'distance': f'{distance\_km:.2f} km',**

**'incidents': "No detailed traffic incidents available" # Placeholder**

**}**

**def display\_traffic\_info(info):**

**"""**

**Display traffic information in the GUI.**

**"""**

**result = f"Estimated Travel Time: {info['travel\_time']}\n"**

**result += f"Distance: {info['distance']}\n"**

**result += f"Traffic Incidents: {info['incidents']}\n" # Placeholder text**

**messagebox.showinfo("Traffic Information", result)**

**def get\_traffic\_info():**

**"""**

**Get traffic information based on user input and display it.**

**"""**

**origin\_address = entry\_origin.get()**

**destination\_address = entry\_destination.get()**

**try:**

**origin\_coords = get\_coordinates(origin\_address)**

**destination\_coords = get\_coordinates(destination\_address)**

**data = fetch\_traffic\_data(origin\_coords, destination\_coords)**

**traffic\_info = parse\_traffic\_data(data)**

**display\_traffic\_info(traffic\_info)**

**except Exception as e:**

**messagebox.showerror("Error", f"An error occurred: {e}")**

**# GUI Setup**

**root = tk.Tk()**

**root.title("Real-Time Traffic Monitor")**

**# Input Labels and Fields**

**label\_origin = tk.Label(root, text="Starting Point (Address):")**

**label\_origin.grid(row=0, column=0, padx=10, pady=10)**

**entry\_origin = tk.Entry(root, width=40)**

**entry\_origin.grid(row=0, column=1, padx=10, pady=10)**

**label\_destination = tk.Label(root, text="Destination (Address):")**

**label\_destination.grid(row=1, column=0, padx=10, pady=10)**

**entry\_destination = tk.Entry(root, width=40)**

**entry\_destination.grid(row=1, column=1, padx=10, pady=10)**

**# Submit Button**

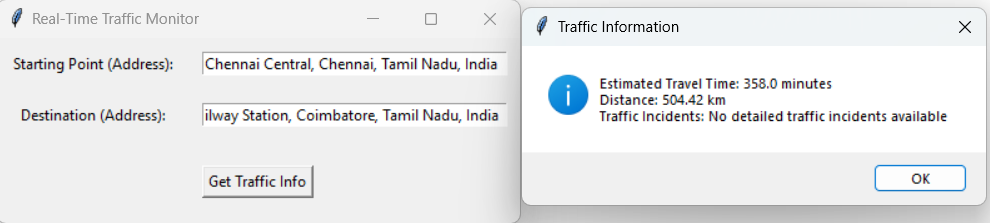
**btn\_get\_info = tk.Button(root, text="Get Traffic Info", command=get\_traffic\_info)**

**btn\_get\_info.grid(row=2, column=0, columnspan=2, pady=20)**

**# Start the GUI event loop**

**root.mainloop()**

**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:**

**Modeling Data Flow:**

* **Fetch COVID-19 Data: The application will make a request to an external API to fetch COVID-19 statistics.**
* **Process Data: The fetched data will be processed and formatted.**
* **Display Data: The formatted data will be displayed to the user in a graphical user interface (GUI).**
* **User Input: Users will input or select a region, and the application will fetch and display corresponding statistics.**

**Implementation Steps:**

* **API Integration: Use the requests library to fetch data from a COVID-19 statistics API.**
* **Data Processing: Extract and format the required data from the API response.**
* **GUI Development: Use the tkinter library to build the user interface that allows users to select a region and view COVID-19 statistics.**

**Pseudocode:**

1. **Initialize GUI**
   * **Create a window with a title.**
   * **Add a label instructing the user to select a country.**
   * **Create a combobox for country selection with search capability.**
   * **Add a button to fetch data.**
   * **Add a text area to display results.**
2. **Define Functions**
   * **get\_covid\_data(country\_code)**
     + **Make a GET request to the COVID-19 API with the country code.**
     + **Handle errors and return JSON response.**
   * **display\_covid\_data(data)**
     + **Format the JSON data into a readable string.**
     + **Return the formatted string.**
   * **fetch\_and\_display(event)**
     + **Get the selected country from the combobox.**
     + **Retrieve the country code from a predefined dictionary.**
     + **Fetch COVID-19 data using the country code.**
     + **Display the data in the text area.**
     + **Handle errors if data fetching fails.**
   * **update\_suggestions(event)**
     + **Update the combobox suggestions based on user input.**
     + **Filter the list of countries to show matches.**
3. **Run GUI Event Loop**
   * **Start the tkinter main event loop.**

**Detailed explanation of the actual code:**

1. **Imports:**
   * **requests for API requests.**
   * **tkinter and ttk for GUI elements.**
   * **pycountry for country codes.**
2. **Function Definitions:**
   * **get\_covid\_data(country\_code):**
     + **Constructs a request to the COVID-19 API using the provided country code.**
     + **Includes headers with the API key for authentication.**
     + **Handles responses and errors.**
   * **display\_covid\_data(data):**
     + **Formats the JSON response into a readable string format.**
     + **Handles cases where data might be returned in a list.**
   * **fetch\_and\_display(event):**
     + **Retrieves the selected country from the combobox.**
     + **Looks up the country code from the dictionary.**
     + **Fetches data from the API and displays it in the text area.**
   * **update\_suggestions(event):**
     + **Updates the combobox values as the user types to show matching countries.**
3. **Main GUI Setup:**
   * **Initializes the main window.**
   * **Creates and configures GUI components including labels, comboboxes, buttons, and text areas.**
   * **Binds actions to GUI events (e.g., button clicks and combobox selection).**

**Assumptions made (if any):**

* **The API key provided is valid and has sufficient usage limits.**
* **The country codes used are valid and supported by the API.**
* **Users will input valid country names that match those in the COUNTRY\_CODES dictionary.**
* **The API response structure is consistent with the expected format.**

**Limitations:**

* **API Key Restrictions: The application relies on a specific API key, which may have usage limits or expiration.**
* **Data Accuracy: The application depends on the external API for data accuracy and timeliness.**
* **Error Handling: Basic error handling is implemented, but additional handling may be needed for edge cases.**
* **User Experience: The combobox is limited by the available country names and may not handle all edge cases gracefully.**

**Code:**

**import requests**

**import tkinter as tk**

**from tkinter import messagebox**

**from tkinter import ttk**

**import pycountry**

**def get\_covid\_data(country\_code):**

**"""**

**Fetch COVID-19 data for a given country using its country code.**

**Parameters:**

**country\_code (str): The ISO 3166-1 alpha-2 code of the country.**

**Returns:**

**dict: The COVID-19 data for the specified country.**

**"""**

**url = "https://covid-19-data.p.rapidapi.com/country/code"**

**querystring = {"format": "json", "code": country\_code}**

**headers = {**

**"x-rapidapi-key": "87d6dba9c3mshcd34518bc372d98p11bfb7jsn76dc70a1bb62",**

**"x-rapidapi-host": "covid-19-data.p.rapidapi.com"**

**}**

**response = requests.get(url, headers=headers, params=querystring)**

**if response.status\_code == 200:**

**return response.json()**

**elif response.status\_code == 403:**

**raise Exception("Access forbidden: Check your API key and usage limits.")**

**else:**

**raise Exception(f"API request failed with status code {response.status\_code}")**

**def display\_covid\_data(data):**

**"""**

**Format the fetched COVID-19 data for display.**

**Parameters:**

**data (dict): The COVID-19 data for a country.**

**Returns:**

**str: The formatted COVID-19 data.**

**"""**

**if isinstance(data, list) and len(data) > 0:**

**data = data[0]**

**return (f"Country: {data['country']}\n"**

**f"Confirmed Cases: {data['confirmed']}\n"**

**f"Deaths: {data['deaths']}\n"**

**f"Recovered: {data['recovered']}\n"**

**f"Critical: {data['critical']}\n"**

**f"Last Update: {data['lastUpdate']}")**

**def fetch\_and\_display(event=None):**

**country\_name = combobox.get()**

**country\_code = COUNTRY\_CODES.get(country\_name)**

**if country\_code:**

**try:**

**covid\_data = get\_covid\_data(country\_code)**

**result = display\_covid\_data(covid\_data)**

**text\_output.config(state=tk.NORMAL)**

**text\_output.delete(1.0, tk.END)**

**text\_output.insert(tk.END, result)**

**text\_output.config(state=tk.DISABLED)**

**except Exception as e:**

**messagebox.showerror("Error", str(e))**

**else:**

**messagebox.showerror("Error", "Selected country code not found.")**

**def update\_suggestions(event=None):**

**typed = combobox.get().lower()**

**suggestions = [country for country in COUNTRY\_CODES.keys() if typed in country.lower()]**

**combobox['values'] = suggestions**

**# Generate a list of country codes and names using pycountry**

**COUNTRY\_CODES = {country.name: country.alpha\_2.lower() for country in pycountry.countries}**

**# Add common names and abbreviations**

**COMMON\_COUNTRIES = {**

**'United States': 'us',**

**'USA': 'us',**

**'US': 'us',**

**'United Kingdom': 'gb',**

**'UK': 'gb',**

**'Canada': 'ca',**

**'Germany': 'de',**

**'Italy': 'it',**

**# Add more common names and abbreviations as needed**

**}**

**# Combine the lists**

**COUNTRY\_CODES.update(COMMON\_COUNTRIES)**

**# Create the main window**

**root = tk.Tk()**

**root.title("COVID-19 Data Fetcher")**

**# Create and place the widgets**

**tk.Label(root, text="Select Country:").pack(pady=5)**

**# Create a combobox for country selection with search capability**

**combobox = ttk.Combobox(root, values=list(COUNTRY\_CODES.keys()), state="normal")**

**combobox.pack(pady=5)**

**combobox.bind("<KeyRelease>", update\_suggestions)**

**combobox.bind("<<ComboboxSelected>>", fetch\_and\_display)**

**tk.Button(root, text="Fetch Data", command=fetch\_and\_display).pack(pady=10)**

**text\_output = tk.Text(root, height=10, width=50, wrap=tk.WORD)**

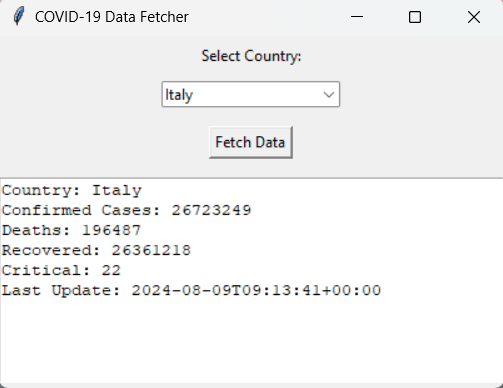
**text\_output.pack(pady=5)**

**text\_output.config(state=tk.DISABLED)**

**# Run the GUI event loop**

**root.mainloop()**

**Sample Output / Screen Shots**

****