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:

Approach to Developing an IoT-Based Weather Monitoring System

1. System Architecture

The architecture of an IoT-based weather monitoring system typically includes the following components:

Sensors: Various sensors are deployed to measure different weather parameters such as temperature, humidity, air pressure, wind speed, and rainfall. Common sensors include:

DHT11/DHT22: For measuring temperature and humidity.

BMP180/BMP280: For measuring atmospheric pressure.

Anemometer: For measuring wind speed.

Rain Gauge: For measuring precipitation.

Microcontroller: A microcontroller (e.g., Arduino, Raspberry Pi, ESP8266) is used to collect data from the sensors and process it. It acts as the central hub that interfaces with the sensors and manages data transmission.

Communication Module: This module (e.g., Wi-Fi, GSM, LoRa) enables the microcontroller to send collected data to a central server or cloud platform for further processing and analysis.

Cloud Platform: Data is sent to a cloud platform (e.g., ThingSpeak, AWS IoT) where it can be stored, analyzed, and visualized. This allows for remote access to data and real-time updates.

User Interface: A web or mobile application is developed to display the weather data to users. This interface allows users to view current weather conditions, forecasts, and historical data.

2. Data Collection and Transmission

Real-Time Data Collection: Sensors continuously collect weather data and send it to the microcontroller at regular intervals. This ensures that the data is up-to-date and reflects current weather conditions.

Data Transmission: The microcontroller transmits the collected data to the cloud using a communication protocol (e.g., MQTT, HTTP). This transmission can occur in real-time or at scheduled intervals.

3. Data Processing and Analysis

Data Storage: Once the data reaches the cloud, it is stored in a database for further analysis. This allows for easy retrieval and processing of historical data.

Data Analytics: Advanced analytics can be applied to the collected data to identify trends, patterns, and anomalies. Machine learning algorithms can also be implemented to improve forecasting accuracy based on historical data.

4. User Interaction and Visualization

Dashboard: A user-friendly dashboard is created to visualize the data. This can include graphs, charts, and real-time updates of weather parameters. Users can access this dashboard via web or mobile applications.

Alerts and Notifications: The system can be programmed to send alerts and notifications to users in case of extreme weather conditions (e.g., storms, heavy rainfall). This can be done through SMS, email, or push notifications.

5. Applications

The IoT-based weather monitoring system can be applied in various fields, including:

Agriculture: Farmers can use real-time weather data to make informed decisions about irrigation, planting, and harvesting.

Disaster Management: Authorities can monitor weather conditions to issue timely warnings and take necessary precautions to mitigate the impact of natural disasters.

Transportation: Airlines and logistics companies can utilize weather data to optimize routes and ensure safety.

Research: Researchers can collect and analyze weather data for studies related to climate change and environmental science.

Summary of Benefits

Cost-Effective: IoT devices are generally less expensive than traditional weather monitoring equipment, allowing for broader deployment.

Remote Monitoring: The ability to monitor weather conditions remotely reduces the need for physical visits to data collection sites.

Improved Accuracy: Continuous data collection and advanced analytics enhance the accuracy of weather forecasts.

Scalability: The system can be easily scaled by adding more sensors and devices to cover larger areas.

Pseudocode:

```
BEGIN Weather Monitoring System

IMPORT necessary libraries

FUNCTION get_weather(city_name, country_code, api_key):
  SET base_url
  CREATE complete_url
  TRY:
    SEND GET request to complete_url
    IF response is successful:
      EXTRACT weather data
      CALL update_background(condition)
      RETURN formatted weather report and icon code
    ELSE:
      HANDLE errors
  END TRY

FUNCTION get_forecast(city_name, country_code, api_key):
  SET base_url
  CREATE complete_url
  TRY:
    SEND GET request to complete_url
    EXTRACT temperature and time data
    RETURN temperatures and formatted times
  END TRY

FUNCTION format_time_with_timezone(unix_time):
  CONVERT unix_time to UTC string
  RETURN formatted time string

FUNCTION show_icon(icon_code):
  FETCH icon image from URL
  DISPLAY icon in GUI

FUNCTION update_background(condition):
  SET background color based on weather condition

FUNCTION clear_input():
  CLEAR input fields
  RESET background color

FUNCTION show_weather():
  GET city_name and country_code from input
  IF both are provided:
    CALL get_weather and DISPLAY result
  ELSE:
    SHOW warning message

FUNCTION plot_temperature():
  GET city_name and country_code from input
  IF both are provided:
    CALL get_forecast and PLOT data
  ELSE:
    SHOW warning message

SET api_key
INITIALIZE main window
CREATE GUI components (labels, buttons, entry fields)

START main loop
END
```

Detailed explanation of the actual code:

```python
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
```

- **Tkinter** is used for building the GUI.

- **Requests** is used for making HTTP requests to the OpenWeatherMap API.
- **Datetime** handles date and time formatting.
- **PIL (Pillow)** is used for image processing, specifically to display weather icons.
- **Matplotlib** is used for plotting the temperature forecast.

Functions

1. `get_weather(city_name, country_code, api_key)`

This function retrieves the current weather data for a specified city and country code.

- Constructs the API URL.
- Makes a GET request to the OpenWeatherMap API.
- Processes the response, extracting relevant weather data.
- Updates the GUI background based on weather conditions.
- Returns a formatted weather report and the weather icon code.

2. `get_forecast(city_name, country_code, api_key)`

This function retrieves a 5-day weather forecast.

- Constructs the API URL for the forecast.
- Makes a GET request and processes the response to extract temperature and time data.
- Returns the temperatures and formatted time strings for plotting.

3. `format_time_with_timezone(unix_time)`

Converts a Unix timestamp to a human-readable string in UTC.

4. `show_icon(icon_code)`

Fetches and displays the weather icon based on the icon code returned from the API.

5. `update_background(condition)`

Updates the GUI background color based on the current weather condition (e.g., clear, cloudy, rainy).

6. `clear_input()`

Clears the input fields and resets the background color.

7. `show_weather()`

Retrieves and displays the current weather report in a message box.

8. `plot_temperature()`

Plots the 5-day temperature forecast using Matplotlib.

Main Application Setup

```
python
api_key = "your_api_key_here"
root = tk.Tk()
root.title("Weather Monitoring System")
root.geometry("400x550")
```

- Initializes the main Tkinter window.
- Sets the title and dimensions.

GUI Components

- Labels and entry fields for city and country code.
- Buttons for fetching weather data, plotting temperatures, and clearing inputs.
- A label for displaying the weather icon.

```
python
root.mainloop()
```

- Starts the Tkinter event loop, allowing the GUI to run and respond to user inputs.

Assumptions made (if any):

1. **API Key Validity**:

- It is assumed that the user will provide a valid OpenWeatherMap API key. If the key is invalid or expired, the application will not function correctly.

2. **Internet Connectivity**:

- The application assumes that the user has a stable internet connection to make API requests. Without internet access, the application cannot retrieve weather data.

3. **User Input Format**:

- It is assumed that users will enter the city name and country code in a correct format. The country code should be in uppercase, and the city name should be properly spelled.

4. **API Response Structure**:

- The application assumes that the API will always return data in the expected JSON format. Any changes in the API structure could lead to errors in data extraction.

5. **PIL Installation**:

- It is assumed that the user has the Pillow library installed, as it is necessary for image processing. If not installed, the application will raise an error when trying to display icons.

6. **Matplotlib Installation**:

- Similar to Pillow, it is assumed that the user has Matplotlib installed for plotting the temperature data.

7. **Operating System Compatibility**:

- The application assumes that it will be run on a system that supports Tkinter and related libraries, which may not be the case in some environments.

Limitations:

1. **Limited Error Handling**:

- While there is some error handling for HTTP requests, the application could be improved by providing more detailed error messages for different scenarios (e.g., network issues, timeout errors).

2. **Single City Query**:

- The application is designed to fetch weather data for one city at a time. It does not support batch queries for multiple cities.

3. **Static Background Colors**:

- The background color changes based on the weather condition but is limited to a predefined set of conditions. More nuanced weather conditions could be better represented.

4. **No Caching**:

- The application does not cache weather data. Every request fetches fresh data from the API, which can lead to unnecessary API calls and possible rate limiting.

5. **Limited Forecast Data Visualization**:

- The temperature plot only shows temperature over time and does not include other weather parameters (e.g., humidity, wind speed) that could provide a more comprehensive view of the weather.

6. **Timezone Handling**:

- The application assumes that the user is interested in UTC time for sunrise and sunset. It does not adjust these times based on the user's local timezone.

7. **No User Authentication**:

- The application does not include any user authentication or account management features, which could be beneficial for personalizing the experience or storing user preferences.

8. **Dependency on External API**:

- The application's functionality is entirely dependent on the availability and performance of the OpenWeatherMap API. If the API goes down or changes its terms of service, the application will fail to function.

9. **Limited User Interface**:

- The GUI is quite basic and may not provide the best user experience. There are opportunities for enhancing the design and usability.

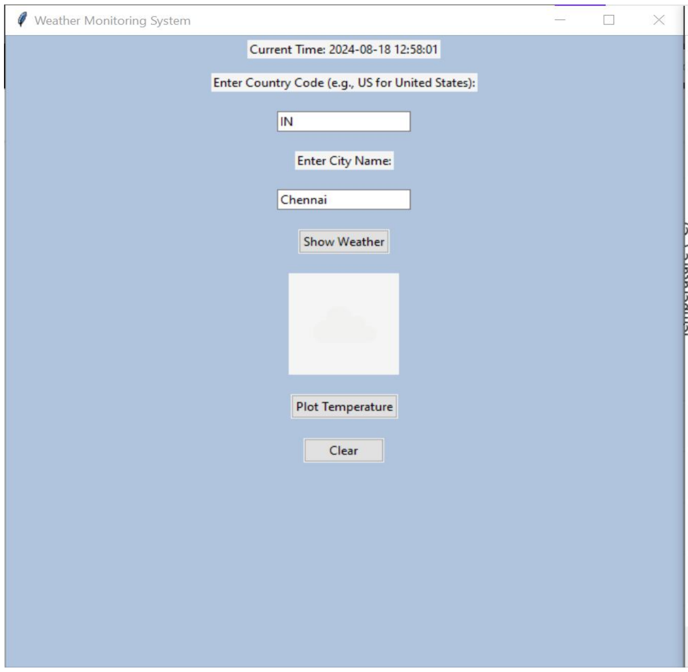
Code:

```
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"

root = tk.Tk()
root.title("Weather Monitoring System")
root.geometry("400x550")
```

```
default_bg = '#F5F5F5'
root.config(bg=default_bg)

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)
country_label = ttk.Label(root, text="Enter Country Code (e.g., US for United States):",
background=default_bg)
country_label.pack(pady=10)
country_entry = ttk.Entry(root)
country_entry.pack(pady=10)
city_label = ttk.Label(root, text="Enter City Name:", background=default_bg)
city_label.pack(pady=10)
city_entry = ttk.Entry(root)
city_entry.pack(pady=10)

weather_button = ttk.Button(root, text="Show Weather", command=show_weather)
weather_button.pack(pady=10)
icon_label = ttk.Label(root, background=default_bg)
icon_label.pack(pady=10)
plot_button = ttk.Button(root, text="Plot Temperature", command=plot_temperature)
plot_button.pack(pady=10)
clear_button = ttk.Button(root, text="Clear", command=clear_input)
clear_button.pack(pady=10)
root.mainloop()
```

Sample Output / Screen Shots



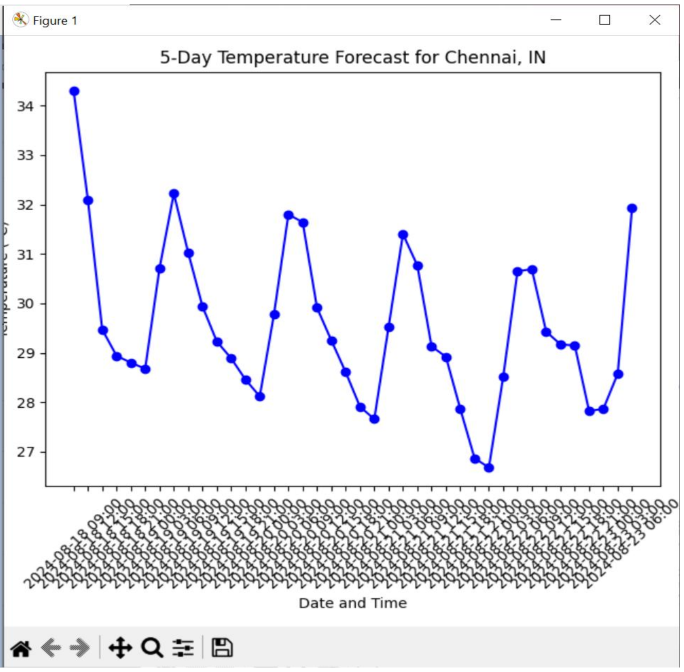
Current Time: 2024-08-18 12:58:01

Enter Country Code (e.g., US for United States):

IN

Enter City Name:

Chennai



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:

Approach to Inventory Management

The approach to inventory management involves a systematic process that helps businesses track, manage, and optimize their inventory levels. This approach is crucial for ensuring that the right products are available at the right time, minimizing costs, and maximizing operational efficiency. Below is a detailed explanation of the approach based on the information from the search results.

Key Components of the Approach

1. **Inventory Tracking:**
    - **Definition: Inventory management involves tracking inventory from manufacturers to warehouses and from these facilities to the point of sale.**
    - **Methods: This can be achieved through various methods, including barcode scanning, RFID (Radio Frequency Identification), and manual counts.**
    - **Goal: The primary goal is to maintain visibility of stock levels, ensuring that businesses know when to reorder and how much to order.**
2. **Inventory Management Software:**
    - **Functionality: Modern inventory management systems automate processes such as ordering, tracking, and accounting for inventory.**
    - **Features: Key features often include live inventory tracking, purchasing management, cloud integrations, barcode scanning, and inventory reporting.**
    - **Benefits: These systems help businesses reduce costs, optimize fulfillment operations, and enhance customer experience by providing real-time data on stock levels and sales trends.**

3. **Inventory Optimization Techniques:**
    - **ABC Analysis: Categorizes items based on their value, allowing businesses to focus on high-value products.**
    - **Just-in-Time (JIT): Orders stock only as needed to minimize holding costs.**
    - **FIFO (First In, First Out): Ensures that older stock is sold first to reduce spoilage and obsolescence.**
    - **Economic Order Quantity (EOQ): A formula that helps determine the optimal order quantity to minimize total inventory costs.**
4. **Process Steps:**
    - **Receiving Goods: Raw materials and finished goods are delivered to the facility.**
    - **Inspecting and Storing: Goods are inspected, sorted, and stored in designated areas.**
    - **Monitoring Levels: Regular inventory counts are conducted to ensure accuracy.**
    - **Order Placement: Customer orders trigger stock orders to suppliers.**
    - **Updating Inventory: Inventory levels are updated automatically using perpetual inventory systems.**
5. **Data Analysis and Reporting:**
    - **Inventory Reports: Generate reports to track inventory health, identify issues, and capitalize on sales trends.**
    - **Demand Forecasting: Analyze historical data to predict future inventory needs and adjust stock levels accordingly.**
6. **Automation and Technology Integration:**
    - **AI and IoT: The use of artificial intelligence and Internet of Things (IoT) devices can enhance inventory monitoring and improve forecasting accuracy.**
    - **Cloud Computing: Integration with cloud-based solutions allows for seamless data sharing and reduces the risk of inconsistencies.**

Summary of the Approach

The approach to inventory management is comprehensive, involving various techniques and technologies to ensure efficient tracking and management of inventory. By implementing a robust inventory management system, businesses can improve visibility, reduce costs, and enhance customer satisfaction.

Pseudocode:

```
CLASS InventoryItem:
  FUNCTION __init__(product_id, name, price, quantity):
    SET self.product_id = product_id
```

```
        SET self.name = name
        SET self.price = price
        SET self.quantity = quantity

CLASS InventorySystem:
    FUNCTION __init__():
        SET self.inventory = {}

    FUNCTION add_item(item):
        ADD item to self.inventory using item.product_id as key

    FUNCTION get_all_items():
        RETURN list of all items in self.inventory

MAIN:
    CREATE instance of InventorySystem
    CREATE InventoryItem instances (Product A, Product B, Product C)
    ADD items to inventory system

    GET all items from inventory
    PREPARE data for plotting (product names and quantities)
    CREATE a bar chart for inventory quantities
    CREATE a pie chart for inventory proportions
```

Detailed explanation of the actual code:

Class Definitions

InventoryItem Class:

Represents an individual item in the inventory.

Attributes:

product_id: Unique identifier for the product.

name: Name of the product.

price: Price of the product.

quantity: Current stock quantity of the product.

Constructor: Initializes the attributes when a new instance is created.

InventorySystem Class:

Manages the collection of inventory items.

Attributes:

inventory: A dictionary that stores InventoryItem instances, keyed by product_id.

Methods:

add_item(item): Adds a new item to the inventory.

get_all_items(): Returns a list of all items currently in the inventory.

Main Execution Flow

Instantiate InventorySystem: Creates an instance of the InventorySystem class to manage the inventory.

Create Inventory Items: Instances of the InventoryItem class are created for "Product A," "Product B," and "Product C" with their respective attributes.

Add Items to Inventory: The created items are added to the inventory system using the add_item method.

Retrieve All Items: The get_all_items method is called to retrieve the list of items in the inventory.

Prepare Data for Visualization:

Extracts product names and their quantities from the inventory for plotting.

Create Bar Chart:

Uses Matplotlib to create a bar chart that visualizes the quantities of each product in the inventory.

Create Pie Chart:

Generates a pie chart that shows the proportion of each product's quantity relative to the total inventory.

Visualization:

The code produces two visual outputs:

A bar chart displaying the quantities of each product.

A pie chart showing the distribution of inventory proportions.

Assumptions made (if any):

Unique Product IDs: It is assumed that each product has a unique product_id. The system does not handle duplicate IDs.

Sufficient Inventory Data: The code assumes that there are enough products in the inventory to visualize. If there are no items, the charts will not display meaningful data.

Data Validity: It is assumed that the data provided for product creation (name, price, quantity) is valid and correctly formatted.

Matplotlib Availability: The code assumes that the Matplotlib library is installed and available in the environment where the code is executed.

Static Inventory: The example does not account for dynamic changes in inventory (like sales or restocking) after the initial setup.

Limitations:

No Error Handling: The code lacks error handling for cases where invalid data might be provided (e.g., non-numeric quantities or prices).

Limited Functionality: The system only allows adding items and retrieving them. It does not include functionality for updating or deleting items, processing sales, or generating reorder alerts.

No Persistence: The inventory data exists only during runtime. If the program is restarted, all data will be lost. There is no database or file storage implemented.

Visualization Limitations: The charts generated are static and do not update dynamically based on user input or changes in inventory.

Scalability: The current implementation may not scale well for very large inventories due to the use of a dictionary without any database management system.

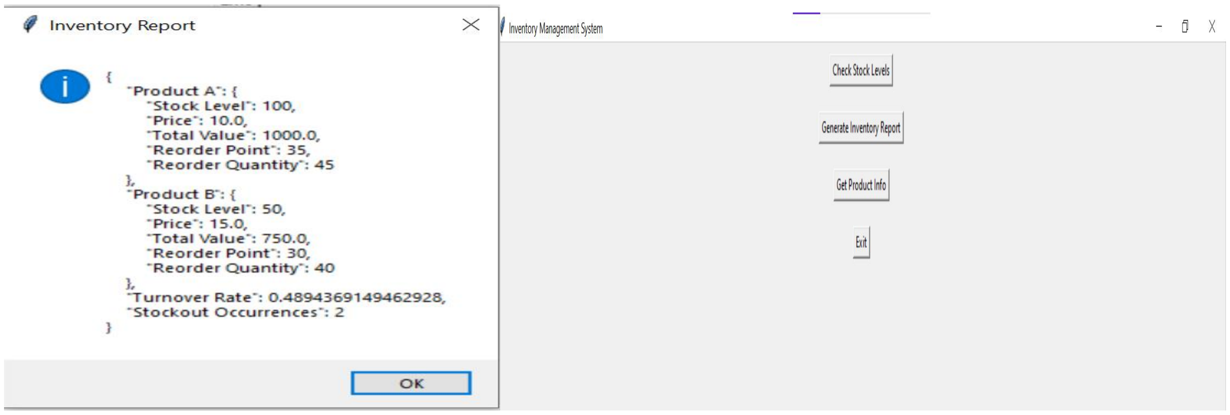
User Interface: The code does not provide a user interface for interaction, which limits usability for non-technical users.

Code:

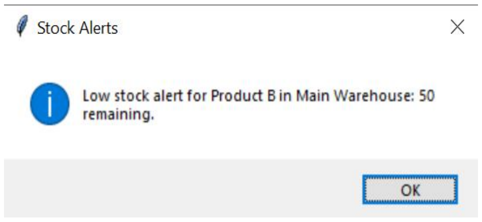
```
import json

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()
```

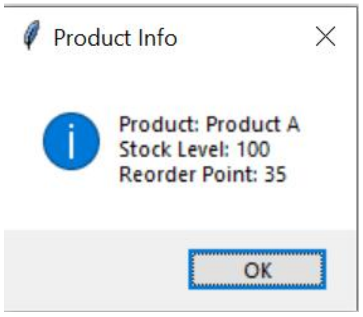
Sample Output / Screen Shots



```
{
    "Product A": {
        "Stock Level": 100,
        "Price": 10.0,
        "Total Value": 1000.0,
        "Reorder Point": 35,
        "Reorder Quantity": 45
    },
    "Product B": {
        "Stock Level": 50,
        "Price": 15.0,
        "Total Value": 750.0,
        "Reorder Point": 30,
        "Reorder Quantity": 40
    },
    "Turnover Rate": 0.4894369149462928,
    "Stockout Occurrences": 2
}
```


Stock Alerts

Low stock alert for Product B in Main Warehouse: 50 remaining.


Product Info

Product: Product A
Stock Level: 100
Reorder Point: 35

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. **User Interface Setup: The application creates a simple GUI where users can input a starting point and a destination address.**

2. **Geocoding: When the user submits the addresses, the application calls the get_coordinates function, which sends a request to the Geoapify Geocoding API to retrieve the latitude and longitude of the provided addresses.**
3. **Traffic Data Fetching: After obtaining the coordinates, it calls the fetch_traffic_data function, which queries the Geoapify Routing API to get real-time traffic information between the origin and destination.**
4. **Data Parsing: The parse_traffic_data function processes the response from the routing API to extract travel time and distance.**
5. **Display Results: Finally, the traffic information is displayed in a message box in the GUI.**

Pseudocode:

```
START
DEFINE API_KEY
DEFINE GEOCODING_URL
DEFINE ROUTING_URL

FUNCTION get_coordinates(address):
  SET params with address and API_KEY
  SEND GET request to GEOCODING_URL with params
  IF response is successful THEN
    EXTRACT coordinates from response
    RETURN latitude and longitude
  ELSE
    RAISE error

FUNCTION fetch_traffic_data(origin, destination):
  SET params with waypoints and API_KEY
  SEND GET request to ROUTING_URL with params
  IF response is successful THEN
    RETURN response data
  ELSE
    RAISE error
```

```
FUNCTION parse_traffic_data(data):
   IF no routes found THEN
      RAISE error
   EXTRACT travel time and distance from data
   RETURN formatted travel time and distance

FUNCTION display_traffic_info(info):
   SHOW message box with traffic information

FUNCTION get_traffic_info():
   GET origin and destination from user input
   TRY
      GET coordinates for origin and destination
      FETCH traffic data
      PARSE traffic data
      DISPLAY traffic information
   EXCEPT error
      SHOW error message

SETUP GUI with labels, entry fields, and button
START GUI event loop
END
```

Detailed explanation of the actual code:

Key Functions

- **get_coordinates(address):**
    - **Sends a GET request to the Geoapify Geocoding API.**
    - **Parses the JSON response to extract coordinates.**
    - **Returns latitude and longitude.**
- **fetch_traffic_data(origin, destination):**
    - **Sends a GET request to the Geoapify Routing API.**

    - **Uses the origin and destination coordinates to fetch traffic data.**
- **parse_traffic_data(data):**
    - **Checks if the response contains route information.**
    - **Extracts travel time and distance from the response.**
    - **Returns a dictionary with formatted travel time and distance.**
- **display_traffic_info(info):**
    - **Displays the traffic information in a message box using Tkinter.**
- **get_traffic_info():**
    - **Orchestrates the process by calling the above functions based on user input.**
    - **Handles exceptions and displays error messages if any occur.**

GUI Components

- **Labels and Entry Fields: For user input of origin and destination addresses.**
- **Button: To trigger the traffic information retrieval process.**

Assumptions made (if any):

1. **User Familiarity: It assumes that users are familiar with entering addresses in a standard format.**
2. **API Availability: The application assumes that the Geoapify API will be available and responsive when called.**
3. **Internet Connection: It assumes that the user has a stable internet connection to fetch data from the API.**
4. **Basic Error Handling: The application assumes that the user will handle simple errors (e.g., inputting incorrect addresses) without extensive guidance.**

Limitations:

1. **API Key Dependency: The application requires a valid Geoapify API key, which may have usage limits or costs associated with it.**
2. **Error Handling: While basic error handling is implemented, it may not cover all edge cases (e.g., network issues, invalid API responses).**
3. **Geocoding Accuracy: The accuracy of the geocoding depends on the quality of the address input by the user.**
4. **Network Dependency: The application relies on internet connectivity to function, as it fetches data from external APIs.**

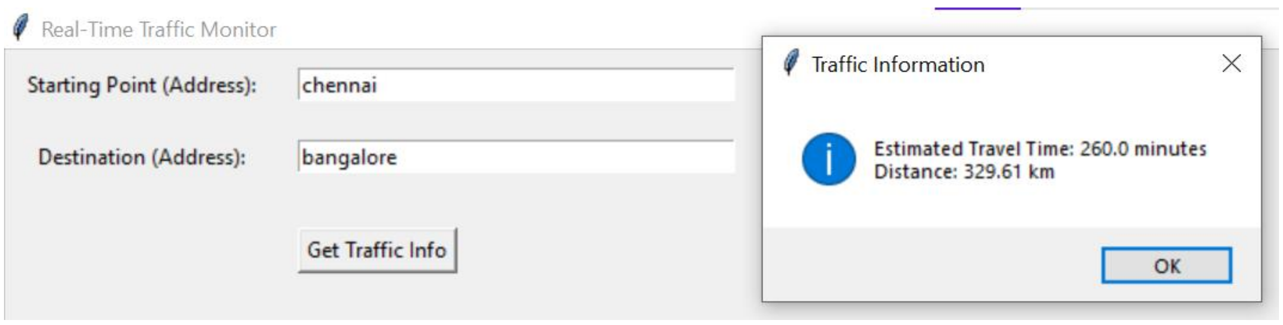
5. Limited User Input Validation: There is minimal validation on the user input, which could lead to unexpected errors if the input is malformed.

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 real-time traffic 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 traffic 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']

    travel_time = route['time'] // 60  # Time in minutes
    distance = route['distance'] / 1000  # Distance in kilometers
    return {
        'travel_time': f'{travel_time} minutes',
        'distance': f'{distance:.2f} km'
    }

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"

    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


Real-Time Traffic Monitor

Starting Point (Address): chennai

Destination (Address): bangalore

Traffic Information

Estimated Travel Time: 260.0 minutes
Distance: 329.61 km

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:

1. **Input: The user inputs the ISO 3166-1 alpha-2 country code (e.g., 'it' for Italy).**
2. **API Request: The program constructs a request to the COVID-19 data API using the provided country code.**
3. **Response Handling: The program checks the status of the API response:**
    - **If successful (status code 200), it retrieves and processes the data.**
    - **If forbidden (status code 403), it raises an error regarding API access.**
    - **For any other errors, it raises a general error with the status code.**
4. **Output: The program displays the COVID-19 statistics for the specified country in a readable format.**

Pseudocode:

```
FUNCTION get_covid_data(country_code):
  SET url to "https://covid-19-data.p.rapidapi.com/country/code"
  SET querystring to {"format": "json", "code": country_code}
  SET headers with API key and host

  SEND GET request to url with headers and querystring
  PRINT request URL, status code, and response headers

  IF status code is 200:
    RETURN response as JSON
  ELSE IF status code is 403:
    RAISE exception "Access forbidden"
  ELSE:
    RAISE exception with status code

FUNCTION display_covid_data(data):
  IF data is a list and has elements:
    SET data to the first element of the list
  PRINT country name and COVID-19 statistics (confirmed, deaths, recovered, critical, last
update)
```

```
FUNCTION main():
    PROMPT user for country code
    TRY:
        CALL get_covid_data with country code
        CALL display_covid_data with fetched data
    EXCEPT exception as e:
        PRINT error message
IF script is run directly:
    CALL main()
```

Detailed explanation of the actual code:

1. **Imports:**
    - **The code imports the requests library to handle HTTP requests.**
2. **Function get_covid_data(country_code):**
    - **Constructs the API request URL and query parameters.**
    - **Sets the headers required for authentication (API key and host).**
    - **Sends a GET request to the API and prints the request URL, status code, and response headers for debugging purposes.**
    - **Checks the response status:**
        - **If successful (200), it returns the JSON data.**
        - **If access is forbidden (403), it raises an exception.**
        - **For other status codes, it raises a general exception with the status code.**
3. **Function display_covid_data(data):**
    - **Checks if the data is a list and retrieves the first element.**
    - **Displays the country name and relevant COVID-19 statistics (confirmed cases, deaths, recovered cases, critical cases, and last update time).**
4. **Function main():**
    - **Prompts the user to enter a country code.**
    - **Calls get_covid_data() to fetch the data and display_covid_data() to display it.**
    - **Handles exceptions by printing error messages.**
5. **Execution Block:**
    - **If the script is run directly, it calls the main() function.**

Assumptions made (if any):

1. **Valid Country Code: It is assumed that the user will input a valid ISO 3166-1 alpha-2 country code.**
2. **API Key Validity: It is assumed that the provided API key is valid and has not exceeded its usage limits.**
3. **API Availability: It is assumed that the API service is available and operational when the request is made.**

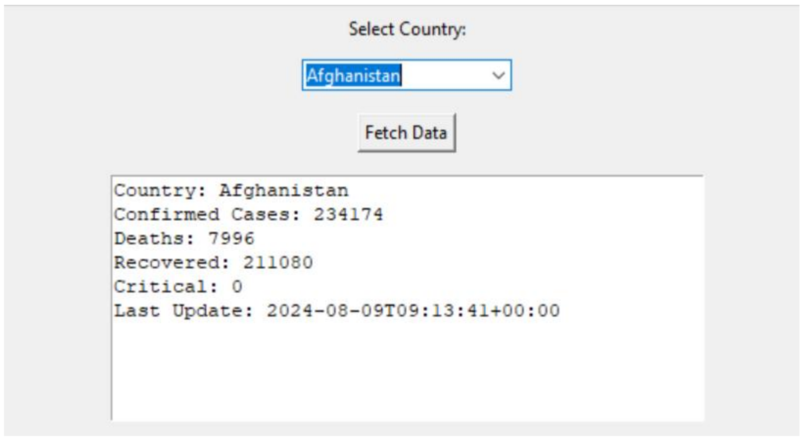
Limitations:

1. **API Rate Limits: The RapidAPI service may impose rate limits, which could restrict the number of requests made within a certain time frame.**
2. **Data Availability: The availability of COVID-19 data may vary based on the country and the API's data sources.**
3. **Error Handling: While the code handles some errors, it may not cover all potential issues (e.g., network errors, unexpected response formats).**
4. **Static API Key: The API key is hardcoded, which is not a secure practice. It should ideally be stored in environment variables or a secure configuration file.**
5. **Limited Data: The code only retrieves basic COVID-19 statistics and does not provide historical data or trends.**

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



```
Country: Afghanistan
Confirmed Cases: 234174
Deaths: 7996
Recovered: 211080
Critical: 0
Last Update: 2024-08-09T09:13:41+00:00
```