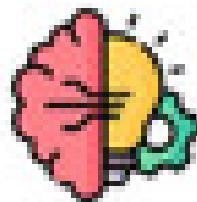
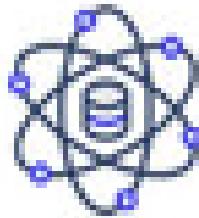




BALIGE
PUBLISHING



DATAFRAME MANIPULATION

THEORY AND APPLICATIONS

WITH

PYTHON

AND

TKINTER

BALIGE CITY
NORTH SUMATERA

VIVIAN SIAHAAN
RISMON HASIHOLAH SIANIPAR

**DATAFRAME MANIPULATION: THEORY
AND APPLICATIONS WITH PYTHON AND
TKINTER**

**DATAFRAME MANIPULATION: THEORY
AND APPLICATIONS WITH PYTHON AND
TKINTER**

**VIVIAN SIAHAAN
RISMON HASIHOLAN SIANIPAR**

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor BALIGE Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book. BALIGE Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BALIGE Publishing cannot guarantee the accuracy of this information.

Copyright © 2024 BALIGE Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor BALIGE Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book. BALIGE Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BALIGE Publishing cannot guarantee the accuracy of this information.

ABOUT THE AUTHOR

ABOUT THE AUTHOR



Vivian Siahaan is a highly motivated individual with a passion for continuous learning and exploring new areas. Born and raised in Hinalang Bagasan, Balige, situated on the picturesque banks of Lake Toba, she completed her high school education at SMAN 1 Balige. Vivian's journey into the world of programming began with a deep dive into various languages such as Java, Android, JavaScript, CSS, C++, Python, R, Visual Basic, Visual C#, MATLAB, Mathematica, PHP, JSP, MySQL, SQL Server, Oracle, Access, and more. Starting from scratch, Vivian diligently studied programming, focusing on mastering the fundamental syntax and logic. She honed her skills by creating practical GUI applications, gradually building her expertise. One particular area of interest for Vivian is animation and game development, where she aspires to make significant contributions. Alongside her programming and mathematical pursuits, she also finds joy in indulging in novels, nurturing her love for literature. Vivian Siahaan's passion for programming and her extensive knowledge are reflected in the numerous ebooks she has authored. Her works, published by Sparta Publisher, cover a wide range of topics, including "Data Structure with Java," "Java Programming: Cookbook," "C++ Programming: Cookbook," "C Programming For High Schools/Vocational Schools and Students," "Java Programming for SMA/SMK," "Java Tutorial: GUI, Graphics and Animation," "Visual Basic Programming: From A to Z," "Java Programming for Animation and Games," "C# Programming for SMA/SMK and Students," "MATLAB For Students and Researchers," "Graphics in JavaScript: Quick Learning Series," "JavaScript Image Processing Methods: From A to Z," "Java GUI Case Study: AWT & Swing," "Basic CSS and JavaScript," "PHP/MySQL Programming: Cookbook," "Visual Basic: Cookbook," "C++ Programming for High Schools/Vocational Schools and Students," "Concepts and Practices of C++," "PHP/MySQL For Students," "C# Programming: From A to Z," "Visual Basic for SMA/SMK and Students," and "C# .NET and SQL Server for High School/Vocational School and Students." Furthermore, at the ANDI Yogyakarta publisher, Vivian Siahaan has contributed to several notable books, including "Python Programming Theory and Practice," "Python GUI Programming," "Python GUI and Database," "Build From Zero School Database Management System In Python/MySQL," "Database Management System in Python/MySQL," "Python/MySQL For Management Systems of Criminal Track Record Database," "Java/MySQL For Management Systems of Criminal Track Records Database," "Database and Cryptography Using Java/MySQL," and "Build From Zero School Database Management System With Java/MySQL." Vivian's diverse range of expertise in programming languages, combined with her passion for exploring new horizons, makes her a dynamic and versatile

individual in the field of technology. Her dedication to learning, coupled with her strong analytical and problem-solving skills, positions her as a valuable asset in any programming endeavor. Vivian Siahaan's contributions to the world of programming and literature continue to inspire and empower aspiring programmers and readers alike.



Rismon Hasiholan Sianipar, born in Pematang Siantar in 1994, is a distinguished researcher and expert in the field of electrical engineering. After completing his education at SMAN 3 Pematang Siantar, Rismon ventured to the city of Jogjakarta to pursue his academic journey. He obtained his Bachelor of Engineering (S.T) and Master of Engineering (M.T) degrees in Electrical Engineering from Gadjah Mada University in 1998 and 2001, respectively, under the guidance of esteemed professors, Dr. Adhi Soesanto and Dr. Thomas Sri Widodo. During his studies, Rismon focused on researching non-stationary signals and their energy analysis using time-frequency maps. He explored the dynamic nature of signal energy distribution on time-frequency maps and developed innovative techniques using discrete wavelet transformations to design non-linear filters for data pattern analysis. His research showcased the application of these techniques in various fields. In recognition of his academic prowess, Rismon was awarded the prestigious Monbukagakusho scholarship by the Japanese Government in 2003. He went on to pursue his Master of Engineering (M.Eng) and Doctor of Engineering (Dr.Eng) degrees at Yamaguchi University, supervised by Prof. Dr. Hidetoshi Miike. Rismon's master's and doctoral theses revolved around combining the SR-FHN (Stochastic Resonance Fitzhugh-Nagumo) filter strength with the cryptosystem ECC (elliptic curve cryptography) 4096-bit. This innovative approach effectively suppressed noise in digital images and videos while ensuring their authenticity. Rismon's research findings have been published in renowned international scientific journals, and his patents have been officially registered in Japan. Notably, one of his patents, with registration number 2008-009549, gained recognition. He actively collaborates with several universities and research institutions in Japan, specializing in cryptography, cryptanalysis, and digital forensics, particularly in the areas of audio, image, and video analysis. With a passion for knowledge sharing, Rismon has authored numerous national and international scientific articles and authored several national books. He has also actively participated in workshops related to cryptography, cryptanalysis, digital watermarking, and digital forensics. During these workshops, Rismon has assisted Prof. Hidetoshi Miike in developing applications related to digital image and video processing, steganography, cryptography, watermarking, and more, which serve as valuable training materials. Rismon's

field of interest encompasses multimedia security, signal processing, digital image and video analysis, cryptography, digital communication, digital forensics, and data compression. He continues to advance his research by developing applications using programming languages such as Python, MATLAB, C++, C, VB.NET, C#.NET, R, and Java. These applications serve both research and commercial purposes, further contributing to the advancement of signal and image analysis. Rismon Hasiholan Sianipar is a dedicated researcher and expert in the field of electrical engineering, particularly in the areas of signal processing, cryptography, and digital forensics. His academic achievements, patented inventions, and extensive publications demonstrate his commitment to advancing knowledge in these fields. Rismon's contributions to academia and his collaborations with prestigious institutions in Japan have solidified his position as a respected figure in the scientific community. Through his ongoing research and development of innovative applications, Rismon continues to make significant contributions to the field of electrical engineering.

ABOUT THE BOOK

ABOUT THE BOOK

A DataFrame is a fundamental data structure in pandas, a powerful Python library for data manipulation and analysis, designed to handle two-dimensional, labeled data akin to a spreadsheet or SQL table. It simplifies working with tabular data by supporting various operations like filtering, sorting, grouping, and aggregating. DataFrames are easily created from lists, dictionaries, or NumPy arrays and offer flexible data handling, including managing missing values and performing input/output operations with different file formats. Key features include hierarchical indexing for multi-level grouping, time series functionality, and integration with libraries such as NumPy and Matplotlib. DataFrame manipulation encompasses filtering, sorting, merging, grouping, pivoting, and reshaping data, while also allowing custom functions, handling missing data, and managing data types. Mastering these techniques is crucial for efficient data analysis, ensuring clean, transformed data ready for deeper insights and decision-making.

In chapter 2, in the first project, we filter a DataFrame named `employee_data`, which includes columns like `'Name'`, `'Department'`, `'Age'`, `'Salary'`, and `'Years_Worked'`, to find employees in the `'Engineering'` department with a salary exceeding \$70,000. We create the DataFrame using sample data and apply boolean indexing to achieve this. The boolean masks `employee_data['Department'] == 'Engineering'` and `employee_data['Salary'] > 70000` identify rows meeting each condition. Combining these masks with the `&` operator filters the DataFrame to include only those rows where both

conditions are met, resulting in a subset of employees who fit the criteria. The final output displays this filtered DataFrame.

In second project, we filter a DataFrame named sales_data, which includes columns such as 'Product', 'Category', 'Quantity Sold', 'Unit Price', and 'Total Revenue', to find products in the 'Electronics' category with quantities sold exceeding 100. We use boolean indexing to achieve this: sales_data['Category'] == 'Electronics' creates a mask for rows in the 'Electronics' category, while sales_data['Quantity_Sold'] > 100 identifies rows where quantities sold are above 100. By combining these masks with the & operator, we filter the DataFrame to include only rows meeting both conditions. The final output displays this filtered subset of products.

In third project, we filter a DataFrame named movie_data, which includes columns such as 'Title', 'Genre', 'Release Year', 'Rating', and 'Box Office Earnings', to find movies released after 2010 with a rating above 8. We use boolean indexing where movie_data['Release_Year'] > 2010 creates a mask for movies released after 2010, and movie_data['Rating'] > 8 identifies movies with ratings higher than 8. By combining these masks with the & operator, we filter the DataFrame to include only the rows meeting both conditions. The final output displays the subset of movies that fit these criteria.

The fourth project demonstrates a Tkinter-based GUI application for filtering a sales dataset using Python libraries Tkinter, Pandas, and PandasTable. The application allows users to interact with a table displaying sales data, applying filters based on product category and quantity sold. The filter_data() function updates the table to show only items

from the selected category with quantities exceeding the specified value, while the refresh_data() function resets the table to display the original dataset. The GUI includes input fields for category selection and quantity entry, along with buttons for filtering and refreshing. The sales data is initially presented in a PandasTable with a toolbar and status bar. Users interact with the interface, which updates and displays filtered data or the full dataset as needed.

The fifth project features a Tkinter GUI application that lets users filter a movie dataset by minimum release year and rating using Python libraries Tkinter, Pandas, and PandasTable. The filter_data() function updates the displayed table based on user inputs, while the refresh_data() function resets it to show the original dataset. The GUI includes fields for entering minimum release year and rating, buttons for filtering and refreshing, and a PandasTable for displaying the data. The application allows for interactive data filtering and visualization, with the table initially populated with sample movie data.

In the sixth project, a retail store manager uses a DataFrame containing sales data to identify products that are both popular and profitable. By applying logical operators to filter the DataFrame, the goal is to isolate products that have sold more than 100 units and generated revenue exceeding \$5000. This filtering is achieved using the Pandas library in Python, where the & operator combines conditions to select the relevant rows. The resulting DataFrame, which includes only products meeting both criteria, provides insights for decision-making and analysis in retail management.

The seventh project involves creating a Tkinter-based GUI application to manage and visualize sales data. The GUI displays data in a table and a bar graph, allowing users to filter products based on minimum quantity sold and total revenue. The application uses pandas for data manipulation, pandastable for table display, and matplotlib for the bar graph. The GUI consists of an input frame for user filters and a display frame for showing the table and graph side by side. Users can update the table and graph by clicking "Filter Data" or reset them to the original data with the "Refresh" button, providing an interactive way to analyze sales performance.

In chapter three, the first project demonstrates how to sort synthetic financial data for analysis. The code imports libraries, sets random seeds for reproducibility, and generates data for businesses including revenue and expenses. It then creates a DataFrame with this data, sorts it by monthly revenue in descending order, and saves the sorted DataFrame to an Excel file. This process aids in organizing and analyzing financial data, making it easier to identify top-performing businesses.

The second project creates a Tkinter GUI to view and interact with synthetic financial data, displaying monthly revenue and expenses for various businesses. It generates random data, stores it in a DataFrame, and sets up a GUI with two tabs: one for sorting by revenue and another for expenses. Each tab features a table to display the data and a matplotlib plot for visual representation. The GUI allows users to sort and view data dynamically, with alternating row colors for readability and embedded plots for better analysis.

The third project generates synthetic unemployment data for 10 regions over 5 years, sets random seeds for reproducibility, and creates a DataFrame with the data. It then sorts the DataFrame alphabetically by region and saves it to an Excel file named "synthetic_unemployment_data.xlsx". Finally, the script prints a confirmation message indicating that the data has been successfully saved.

The fourth project generates synthetic unemployment data for 25 regions over a 5-year period and creates a Tkinter GUI for interactive data exploration. The data, organized into a DataFrame and saved to an Excel file, is displayed in a tabbed interface with two views: one sorted by unemployment rate and another by year. Each tab features scrollable tables and corresponding bar charts for visual analysis. The UnemploymentDataGUI class manages the interface, updating tables and graphs dynamically to allow users to explore regional and yearly unemployment variations effectively.

The fifth project demonstrates how to concatenate dataframes with synthetic temperature data for various countries. Initially, we generate temperature data for countries like the USA and Canada for each month. Next, we create an additional dataframe with temperature data for other countries such as the UK and Germany. We then concatenate the original and additional dataframes into a single dataframe and save the combined data to an Excel file named combined_temperatures.xlsx. The steps involve generating synthetic data, creating additional dataframes, concatenating them, and exporting the result to Excel.

The sixth project demonstrates how to build a Tkinter application to visualize synthetic temperature data. The app features a tabbed interface with tabs for displaying raw data, temperature graphs, and filters. It uses alternating row colors for better readability and includes functionality for filtering data by country and month. Users can view and analyze temperature data across different countries through tables and graphical representations, and apply or reset filters as needed.

The seventh project demonstrates how to perform an inner join on two synthetic dataframes: one containing housing details and the other containing owner information. First, synthetic data is generated for houses and their owners. The dataframes are then merged on the common key, HouseID, using an inner join to include only rows with matching keys. Finally, the combined data is saved to an Excel file named `combined_housing_data.xlsx`. The result is an Excel file that contains details about houses along with their respective owners.

The eighth project provides an interactive platform for managing and visualizing synthetic housing data. Users can view comprehensive tables, apply filters for location and house type, and analyze house price distributions with Matplotlib plots. The application includes tabs for displaying data, filtering results, and generating visualizations, with functionalities to reset filters, save filtered data to Excel, and ensure a user-friendly experience with alternating row colors in tables and dynamic updates.

To demonstrate an outer join on DataFrames with synthetic medical data, in ninth project, we create two DataFrames: one

for patient information and another for medical records. We then perform an outer join to ensure all patients and records are included, even if some records don't have corresponding patient data. The code generates synthetic data, performs the outer join using `pd.merge()` on the PatientID column, and saves the result to an Excel file named `outer_join_medical_data.xlsx`. This approach provides a comprehensive dataset with complete patient and medical record information.

The tenth project involves creating a Tkinter-based desktop application to visualize and interact with synthetic medical data. The application uses an outer join to merge patient and medical record datasets, displaying the comprehensive result in a user-friendly table. Users can filter data by patient ID and condition, view distribution graphs of medical conditions, and save filtered results to an Excel file. The GUI, leveraging Tkinter and Matplotlib, includes tabs for data display, filtering, and graph visualization, providing a robust tool for exploring medical datasets.

In chapter four, the first project demonstrates creating and manipulating a synthetic insurance dataset. Using numpy and pandas, the script generates random data including columns for Policyholder, Age, State, Coverage_Type, and Premium. It groups this data by State and Coverage_Type to show basic data segmentation, then saves the dataset to an Excel file for further analysis. The code provides a practical framework for simulating and analyzing insurance data by illustrating the process of data creation, grouping, and storage.

The second project demonstrates a Tkinter GUI application designed for analyzing a synthetic insurance dataset. The GUI displays 1,000 records of policyholder data in a scrollable table using the Treeview widget, with options to filter by state and coverage type. Users can save filtered data to an Excel file and generate a bar plot of policy distribution by state, integrated into the Tkinter window using Matplotlib. This application provides interactive tools for data exploration, filtering, exporting, and visualization in a user-friendly interface.

The third project focuses on creating, analyzing, and aggregating a large synthetic sales dataset with 10,000 records. This dataset includes salespersons, regions, products, sales amounts, and timestamps, simulating a detailed sales environment. The core task involves grouping the data by region, product, and salesperson to calculate total sales and transaction counts. This aggregated data is saved to an Excel file, providing insights into sales performance and trends, which helps businesses optimize their sales strategies and make informed decisions.

The fourth project develops a Tkinter GUI for analyzing synthetic sales data, allowing users to explore raw and aggregated data interactively. The application includes a dual-view setup with raw and aggregated data tables, filtering options for region, product, and salesperson, and visualization features for generating plots. Users can apply filters, view data summaries, save results to Excel, and visualize sales trends by region. The GUI is designed to provide a comprehensive tool for data analysis, visualization, and reporting. The dataset includes 10,000 records with attributes such as salesperson,

region, product, sales amount, and date, and is grouped by region, product, and salesperson to aggregate sales data.

The fifth project demonstrates how to create and analyze a synthetic transportation dataset. The code generates a large dataset simulating vehicle and route data, including distances traveled and durations. It groups the data by vehicle and route, calculating total and average distances and durations, and then saves these aggregated results to an Excel file. This approach allows for detailed examination of transportation patterns and performance metrics, facilitating reporting and decision-making.

The sixth project outlines a Tkinter GUI project for analyzing synthetic transportation data using Python. This GUI, combining Tkinter and Matplotlib, provides a user-friendly interface to inspect and visualize large datasets involving vehicle routes, distances, and durations. It features interactive tables for raw and aggregated data, filter options for vehicle, route, and date, and integrates various plots like histograms and bar charts for data visualization. Users can apply filters, view dynamic updates, and save filtered data to Excel. The goal is to facilitate comprehensive data analysis and enhance decision-making through an intuitive, interactive tool.

In chapter five, the first project involves generating and analyzing a synthetic dataset representing gold production across countries, years, and regions. The dataset, created with attributes like country, year, region, and production quantities, simulates complex real-world data for detailed analysis. By using the pivot_table method, the data is transformed to aggregate gold production metrics by country and region over

different years, revealing trends and patterns. The results are saved as both original and pivoted datasets in Excel files for easy access and further analysis, aiding in decision-making related to mining and resource management.

The second project creates an interactive Tkinter GUI to visualize and interact with a large synthetic dataset on gold production, including details on countries, regions, mines, and yearly production. Using pandas and numpy to generate the dataset, the GUI features multiple tabs for viewing the original data, pivoted data, and various summary statistics, alongside graphical visualizations of gold production trends across countries, regions, and years. The application integrates matplotlib for embedding charts within the Tkinter interface, making it a comprehensive tool for exploring and analyzing the data effectively.

The third project demonstrates how to create a synthetic dataset simulating stock prices for multiple companies over 10,000 days, using random number generation to simulate stock prices for AAPL, GOOG, AMZN, MSFT, TSLA, and META. The dataset, initially in a wide format with separate columns for each company's stock prices, is then reshaped to a long format using pd.melt(). This long format, where each row represents a single date, stock, and its price, is often better suited for data analysis and visualization. Finally, both the original and unpivoted DataFrames are saved to separate Excel files for further use.

The fourth project involves developing a visually engaging Tkinter GUI to analyze and visualize a synthetic stock dataset. The application handles stock price data for multiple

companies, offering users both the original and unpivoted DataFrames, along with summary statistics and graphical representations. The GUI includes tabs for viewing raw and transformed data, statistical summaries, and interactive graphs, utilizing Tkinter's advanced widgets for a polished user experience. Data is saved to Excel files, and Matplotlib charts are integrated for clear data visualization, making the tool useful for both casual and advanced analysis of stock market trends.

In chapter six, the first project demonstrates creating a large synthetic road traffic dataset with 10,000 rows using randomization techniques. Fields include Date, Time, Location, Vehicle_Count, Average_Speed, and Incident. Random NaN values are introduced into 10% of the dataset to simulate missing data. The dataset is then cleaned by removing rows with any missing values using dropna(), and the resulting cleaned DataFrame is saved to 'cleaned_large_road_traffic_data.xlsx' for further analysis.

The second project creates a Tkinter-based GUI to analyze and visualize a synthetic road traffic dataset. It generates a dataset with 10,000 rows, including fields like date, time, location, vehicle count, average speed, and incidents. Random missing values are introduced and then removed by dropping rows with any NaNs. The GUI features four tabs: one for the original dataset, one for the cleaned dataset, one for summary statistics, and one for distribution graphs. Users can explore data tables with Tkinter's Treeview widget and view visualizations such as histograms and bar charts using Matplotlib, providing a comprehensive tool for data analysis.

The third project generates a large synthetic electricity dataset to simulate real-world patterns in electricity consumption, temperature, and pricing. Missing values are introduced and then handled by filling gaps with regional averages for consumption, forward-filling temperature data, and using overall means for pricing. The cleaned dataset is saved to an Excel file, offering a valuable resource for testing data processing methods and developing data analysis algorithms in a controlled environment.

The fourth project demonstrates a Tkinter GUI for handling missing data in a synthetic electricity dataset. The application offers a multi-tab interface to analyze electricity consumption data, including features for displaying the original and cleaned DataFrames, summary statistics, distribution graphs, and time-series plots. Users can view raw and processed data, explore statistical summaries, and visualize distributions and trends in electricity consumption, temperature, and pricing over time. The GUI integrates data generation, cleaning, and visualization techniques, providing a comprehensive tool for electricity data analysis.

CONTENT

CONTENT

| | |
|--|----------|
| INTRODUCTION | 1 |
| INTRODUCTION | 1 |
| DATAFRAME MANIPULATION | 4 |
| | |
| FILTERING | 7 |
| INTRODUCTION | 7 |
| BOOLEAN INDEXING | 9 |
| EXAMPLE 2.1: Filtering Employees Database | 9 |
| EXAMPLE 2.2: Filtering Sales Dataset | 10 |
| EXAMPLE 2.3: Filtering Movies Dataset | 11 |
| EXAMPLE 2.4: GUI Tkinter for Filtering Sales Dataset | 12 |
| EXAMPLE 2.5: GUI Tkinter for Filtering Movies Dataset | 15 |
| LOGICAL OPERATORS | 19 |
| EXAMPLE 2.6: Identifying Popular and Profitable Products | 20 |
| EXAMPLE 2.7: GUI Tkinter for Identifying Popular and Profitable Products | 21 |
| 26 | 26 |
| EXAMPLE 2.8: Analyzing Employee Performance | 28 |
| EXAMPLE 2.9: GUI Tkinter for Analyzing Employee Performance | 34 |
| 36 | 36 |

| | |
|--|------------|
| QUERY METHOD | 38 |
| EXAMPLE 2.10: Filtering Data Population | 43 |
| EXAMPLE 2.11: GUI Tkinter for Filtering Data Population | 44 |
| FILTERING COLUMNS | 46 |
| EXAMPLE 2.12: Filtering Traffic Web Data | 52 |
| EXAMPLE 2.13: GUI Tkinter for Filtering Traffic Web Data | 55 |
| EXAMPLE 2.14: Generating A Synthetic Dataset for Web Traffic | 63 |
| EXAMPLE 2.15: GUI Tkinter for Analyzing Synthetic Dataset for Web Traffic | 64 |
| USING FUNCTION | 76 |
| EXAMPLE 2.16: Filtering World Food Data | 77 |
| EXAMPLE 2.17: GUI Tkinter for Filtering World Food Data | 86 |
| COMBINING FILTERING WITH OTHER OPERATIONS | 88 |
| EXAMPLE 2.18: Filtering and Aggregating World Food Data | 97 |
| EXAMPLE 2.19: GUI Tkinter for Filtering and Aggregating World Food Data | 97 |
| EXAMPLE 2.20: Filtering and Aggregating Synthetic Forest Data | 99 |
| EXAMPLE 2.21: GUI Tkinter for Filtering and Aggregating Synthetic Forest Data | 103 |
| SORTING, JOINING, AND MERGING | 109 |
| SORTING | 111 |
| EXAMPLE 3.1: Sorting Synthetic Financial Data | 118 |
| EXAMPLE 3.2: GUI Tkinter for Sorting Synthetic Financial Data | 125 |
| EXAMPLE 3.3: Sorting Synthetic Unemployment Data by Index | 127 |
| EXAMPLE 3.4: GUI Tkinter for Sorting Synthetic Unemployment Data by Index | 155 |
| JOINING AND MERGING | 156 |
| EXAMPLE 3.5: Concatenating Dataframes Using Synthetic Temperature Data for Different Countries | 172 |
| | 174 |

| | |
|--|------------|
| EXAMPLE 3.6: GUI Tkinter for Concatenating Dataframes Using Synthetic Temperature Data for Different Countries | 187 |
| EXAMPLE 3.7: Performing Inner Join on Dataframes with Synthetic Housing Data | 189 |
| EXAMPLE 3.8: GUI Tkinter for Performing Inner Join on Dataframes with Synthetic Housing Data | 205 |
| EXAMPLE 3.9: Performing Outer Join on Dataframes with Synthetic Medical Data | 205 |
| EXAMPLE 3.10: GUI Tkinter for Performing Outer Join on Dataframes with Synthetic Medical Data | 209 |
| EXAMPLE 3.11: Performing Left Join on Dataframes with Synthetic Hospital Data | 213 |
| EXAMPLE 3.12: GUI Tkinter for Performing Left Join on Dataframes with Synthetic Hospital Data | 223 |
| EXAMPLE 3.13: Performing Right Join on Dataframes with Synthetic University Data | 227 |
| EXAMPLE 3.14: GUI Tkinter for Performing Right Join on Dataframes with Synthetic University Data | 238 |
| | 243 |
| | 257 |
| | 261 |
| | 278 |
| GROUPING AND AGGREGATING | |
| INTRODUCTION | 282 |
| EXAMPLE 4.1: Grouping with Synthetic Insurance Data | |
| EXAMPLE 4.2: GUI Tkinter for Grouping with Synthetic Insurance Data | |
| EXAMPLE 4.3: Grouping Aggregating with Synthetic Sales Data | 298 |
| EXAMPLE 4.4: GUI Tkinter for Grouping Aggregating with Synthetic Sales Data | 298 |
| EXAMPLE 4.5: Grouping and Multiple Aggregations with Synthetic Transportation Data | 302 |
| EXAMPLE 4.6: GUI Tkinter for Grouping and Multiple Aggregations with Synthetic Transportation Data | 306 |
| EXAMPLE 4.7: Advanced Grouping Multiple Columns and Multiple Aggregations with Synthetic Marketing Data | 309 |
| | 327 |
| | 331 |
| | 347 |
| | 352 |

| | |
|---|------------|
| EXAMPLE 4.8: GUI Tkinter for Advanced Grouping Multiple Columns and Multiple Aggregations with Synthetic Marketing Data | 366 |
| EXAMPLE 4.9: Advanced Grouping Multiple Columns and Multiple Aggregations with Synthetic Weather Data | 366 371 |
| EXAMPLE 4.10: GUI Tkinter Advanced Grouping Multiple Columns and Multiple Aggregations with Synthetic Weather Data | 376 |
| | 383 |
| | 388 |
| <i>PIVOTING AND RERSHAPING</i> | |
| PIVOTING | 403 |
| RESHAPING | |
| EXAMPLE 5.1: Pivoting Dataframe with Synthetic Gold Dataset | 407 |
| EXAMPLE 5.2: GUI Tkinter for Pivoting Dataframe with Synthetic Gold Dataset | 420 |
| EXAMPLE 5.3: UnPivoting Dataframe with Synthetic Stock Dataset | |
| EXAMPLE 5.4: GUI Tkinter for UnPivoting Dataframe with Synthetic Stock Dataset | |
| EXAMPLE 5.5: Stacking Dataframe with Synthetic Heart Disease Dataset | |
| EXAMPLE 5.6: GUI Tkinter for Stacking Dataframe with Synthetic Heart Disease Dataset | |
| <i>HANDLING MISSING DATA</i> | |
| INTRODUCTION | |
| EXAMPLE 6.1: Handling Missing Data with Dropping Rows Using Road Traffic Dataset | |
| EXAMPLE 6.2: GUI Tkinter for Handling Missing Data with Dropping Rows Using Road Traffic Dataset | |
| EXAMPLE 6.3: Handling Missing Data by Filling It Using Synthetic Electricity Dataset | |

EXAMPLE 6.4: GUI Tkinter for Handling Missing Data by Filling It Using Synthetic Electricity Dataset

EXAMPLE 6.5: Advanced Handling Missing Data by Filling It Using Synthetic Wind Power Generation Dataset

EXAMPLE 6.6: GUI Tkinter for Advanced Handling Missing Data by Filling It Using Synthetic Wind Power Generation Dataset

BIBLIOGRAPHY

INTRODUCTION
INTRODUCTION

INTRODUCTION

A DataFrame is a fundamental data structure in pandas, a powerful Python library for data manipulation and analysis. It provides a two-dimensional, labeled data structure with columns of potentially different types, similar to a spreadsheet or SQL table.

When you work with data, especially in fields like data science, machine learning, or finance, you often deal with tabular data. DataFrames make it easy to represent and manipulate this type of data efficiently.

Creating a DataFrame is straightforward. You can create one from various data structures like lists, dictionaries, or NumPy arrays. Once you have your data in a DataFrame, you can perform numerous operations on it, such as filtering, sorting, grouping, and aggregating.

DataFrames have rows and columns, and each column has a unique label. These labels make it easy to reference and manipulate specific data within the DataFrame. You can access columns by their labels or indices, and you can also select rows based on conditions.

One of the key features of DataFrames is their ability to handle missing data gracefully. Pandas provides functions to detect, remove, or fill missing values, which is crucial when working with real-world datasets.

DataFrames also support various input/output operations, allowing you to read data from and write data to different file formats like CSV, Excel, SQL databases, JSON, and more.

Pandas offers extensive functionality for data manipulation and analysis, including methods for merging and joining DataFrames, reshaping data, and performing statistical calculations.

DataFrames can be sliced and diced in multiple ways to extract the information you need. Whether you want to select specific rows and columns, apply functions row-wise or column-wise, or reshape the DataFrame, pandas has you covered.

You can perform arithmetic and logical operations on DataFrames, either element-wise or across entire rows or columns. This capability is invaluable for data cleaning, transformation, and calculation.

Pandas DataFrames are highly customizable. You can rename columns, set custom indices, change data types, and apply custom functions to manipulate the data according to your requirements.

DataFrames support hierarchical indexing, also known as multi-indexing, which allows you to represent higher-dimensional data in a two-dimensional DataFrame. This feature is useful for handling complex datasets with multiple levels of grouping.

Pandas provides powerful time series functionality, making it easy to work with temporal data in DataFrames. You can resample, interpolate, and perform various time-based operations with minimal effort.

DataFrames integrate seamlessly with other Python libraries like NumPy, Matplotlib, and scikit-learn, enabling you to build end-to-end data analysis and machine learning pipelines.

Pandas DataFrames are incredibly efficient, even with large datasets. The underlying implementation is optimized for speed and memory usage, ensuring that you can work with data efficiently, even on modest hardware.

You can visualize DataFrames and their contents using built-in plotting functions in pandas or by integrating with external visualization libraries like Matplotlib or Seaborn.

DataFrames support method chaining, allowing you to perform multiple operations in a single line of code. This approach promotes concise and readable code, enhancing your productivity.

Pandas provides powerful indexing and selection mechanisms, including label-based, positional, and boolean indexing, enabling you to extract data from DataFrames with precision and flexibility.

You can perform group-wise operations on DataFrames using the groupby function, which splits the data into groups based on one or more keys and then applies a function to each group.

Pandas DataFrames support both row-wise and column-wise iteration, allowing you to loop over rows or columns and perform operations or calculations as needed.

DataFrames are versatile data structures that can represent a wide range of data types, including numerical, categorical, textual, and temporal data.

You can apply custom functions to DataFrames using the apply function, which allows you to process each row or column independently and generate new data based on your logic.

Pandas provides robust support for data cleaning and preprocessing tasks, including handling duplicates, outliers, and inconsistent data, as well as converting between different data types.

DataFrames are immutable objects, meaning that operations like filtering or transforming data create new DataFrame objects without modifying the original data, ensuring data integrity and reproducibility.

You can concatenate, append, or merge DataFrames horizontally or vertically to combine data from multiple sources or perform advanced data transformations.

Pandas DataFrames are interoperable with SQL databases, allowing you to execute SQL queries directly on DataFrame objects using the pandasql library or the to_sql method.

You can serialize and deserialize DataFrames using the built-in pickle module or the more efficient HDF5 format, enabling you to save and load data efficiently for future analysis.

DataFrames are not only limited to numerical and tabular data; you can store and manipulate complex data structures like nested dictionaries or JSON objects within DataFrame cells.

Pandas provides extensive documentation, tutorials, and community support, making it easy for beginners to learn and master the DataFrame manipulation techniques.

In summary, DataFrames are a versatile and powerful tool for data manipulation and analysis in Python, offering a wide range of functionality for working with tabular data efficiently and effectively. Whether you're cleaning messy data, performing complex calculations, or building predictive models, pandas DataFrames provide the tools you need to get the job done.

DATAFRAME MANIPULATION

DataFrame manipulation involves performing various operations to modify, reshape, clean, or transform data within a DataFrame. These operations are essential for preparing data for analysis, visualization, or machine learning tasks. DataFrame manipulation encompasses a wide range of techniques, including filtering, sorting, merging, joining, grouping, pivoting, and aggregating data. Let's delve into each of these aspects in detail:

1. Filtering: Filtering involves selecting a subset of rows or columns from the DataFrame based on specific conditions. You can use boolean indexing or the query()

method to filter rows based on logical conditions applied to column values.

2. Sorting: Sorting rearranges the rows of the DataFrame based on the values in one or more columns. You can sort data in ascending or descending order using the `sort_values()` method, specifying the column(s) to sort by.
3. Merging and Joining: Merging and joining involve combining data from multiple DataFrames based on a common key or index. Pandas provides several functions like `merge()`, `concat()`, and `join()` to perform these operations, allowing you to combine data horizontally or vertically.
4. Grouping: Grouping involves splitting the DataFrame into groups based on one or more keys and then applying a function to each group independently. You can use the `groupby()` method to create groups and then apply aggregation functions like `sum()`, `mean()`, `count()`, etc., to calculate summary statistics for each group.
5. Pivoting: Pivoting reshapes the DataFrame by rearranging the data based on the values in one or more columns. You can use the `pivot_table()` function to create a pivot table, where the rows represent one variable, the columns represent another variable, and the values are aggregated based on a third variable.
6. Aggregating: Aggregating involves computing summary statistics or metrics from the data. You can use functions like `sum()`, `mean()`, `min()`, `max()`, `count()`, etc., to aggregate data across rows or columns.
7. Reshaping: Reshaping transforms the layout or structure of the DataFrame. Pandas provides functions like `stack()`, `unstack()`, `melt()`, and `pivot()` for reshaping data between long and wide formats, or vice versa.

8. Applying Functions: You can apply custom functions to each row or column of the DataFrame using the `apply()` method. This allows you to perform complex calculations or transformations on the data.
9. Handling Missing Data: DataFrames often contain missing or null values, which need to be handled appropriately. Pandas provides functions like `dropna()`, `fillna()`, and `interpolate()` to handle missing data by either removing, filling, or interpolating missing values.
10. Dropping Columns or Rows: Sometimes, you may need to remove certain columns or rows from the DataFrame. You can use the `drop()` method to drop columns or rows based on their labels or indices.
11. Renaming Columns: You can rename columns in the DataFrame using the `rename()` method, allowing you to make column labels more descriptive or meaningful.
12. Changing Data Types: DataFrames allow you to convert the data type of columns using the `astype()` method, enabling you to ensure consistency or optimize memory usage.
13. Duplicating and Dropping Duplicates: Pandas provides functions like `duplicated()` and `drop_duplicates()` to identify and remove duplicate rows from the DataFrame based on specific columns or criteria.
14. Shifting and Lagging: You can shift or lag values within a column using the `shift()` method, which is useful for calculating differences or changes between consecutive rows.
15. Rolling and Window Functions: Pandas supports rolling and window functions, which apply a function over a rolling window of data. This is useful for calculating

moving averages, cumulative sums, or other window-based statistics.

These are just some of the key DataFrame manipulation techniques in pandas. Mastering these operations allows you to efficiently manipulate, transform, and analyze data to derive meaningful insights and make informed decisions.

DATAFRAME FILTERING DATAFRAME FILTERING

INTRODUCTION

DataFrame filtering is the process of selecting a subset of rows or columns from a DataFrame based on specific conditions. Filtering allows you to focus on the data that meets certain criteria, facilitating data exploration, analysis, and visualization.

Here's how DataFrame filtering works:

1. Boolean Indexing: Boolean indexing is the most common method for filtering DataFrames in pandas. It involves creating a boolean mask—a series of True and False values—that indicates which rows or columns satisfy the given condition(s). You can then use this boolean mask to select the desired subset of data.

```
# Example: Filtering rows where the 'age' column is greater than 30
df_filtered = df[df['age'] > 30]
```

2. Logical Operators: You can use logical operators like & (and), | (or), and ~ (not) to combine multiple conditions when filtering DataFrames. These operators allow you to create complex filtering criteria by combining simpler conditions.

```
# Example: Filtering rows where the 'age' column is greater than 30 and the 'gender' column is 'Male'
df_filtered = df[(df['age'] > 30) & (df['gender'] == 'Male')]
```

3. Query Method: The query() method provides an alternative way to filter DataFrames using SQL-like syntax. You can pass a string containing a boolean expression to the query() method, and it will return the subset of data that satisfies the expression.

```
# Example: Filtering rows where the 'age' column is greater than 30 using the query method
df_filtered = df.query('age > 30')
```

4. Filtering Columns: In addition to filtering rows, you can also filter columns based on specific criteria. You can use

boolean indexing or the loc accessor to select columns that meet certain conditions.

```
# Example: Filtering columns where all values are greater than 0
df_filtered = df.loc[:, (df > 0).all()]
```

5. Using Functions: You can apply custom functions to filter DataFrames based on more complex or dynamic criteria. This is achieved by using the apply() method along with a lambda function or a custom function defined separately.

```
# Example: Filtering rows where the 'name' column starts with 'A'
df_filtered = df[df['name'].apply(lambda x: x.startswith('A'))]
```

6. Combining Filtering with Other Operations: Filtering can be combined with other DataFrame operations like sorting, grouping, and aggregation to perform more sophisticated data analysis tasks. For example, you can filter data before computing summary statistics or generating visualizations.

```
# Example: Filtering rows where the 'age' column is greater than 30 and computing the average salary
average_salary = df[df['age'] > 30]['salary'].mean()
```

By using DataFrame filtering, you can extract relevant subsets of data from large datasets, focus on specific observations or variables, and perform targeted analyses to gain insights into your data.

BOOLEAN INDEXING

EXAMPLE 2.1

Filtering Employees Database

Let's consider a dataset containing information about employees in a company. Suppose the DataFrame `employee_data` has columns such as 'Name', 'Department', 'Age', 'Salary', and 'Years_Worked'. We want to filter the DataFrame to select employees who meet certain criteria, such as being in the 'Engineering' department and having a salary greater than \$70,000.

```
import pandas as pd

# Sample employee data
data = {
    'Name': ['John', 'Jane', 'Doe', 'Alice',
    'Bob'],
    'Department': ['Engineering', 'HR',
    'Engineering', 'Finance', 'Engineering'],
    'Age': [35, 28, 40, 45, 30],
    'Salary': [80000, 60000, 75000, 90000,
    70000],
    'Years_Worked': [5, 3, 8, 10, 6]
}

employee_data = pd.DataFrame(data)

# Boolean indexing to filter employees
```

```

engineering_high_salary = 
employee_data[(employee_data['Department'] == 
'Engineering') & (employee_data['Salary'] > 
70000)]

print(engineering_high_salary)

```

Output:

| | Name | Department | Age | Salary |
|--------------|------|-------------|-----|--------|
| Years_Worked | | | | |
| 0 | John | Engineering | 35 | 80000 |
| 5 | | | | |
| 2 | Doe | Engineering | 40 | 75000 |
| 8 | | | | |
| 4 | Bob | Engineering | 30 | 70000 |
| 6 | | | | |

In this example, we used boolean indexing to filter the DataFrame `employee_data`. The expression `employee_data['Department'] == 'Engineering'` creates a boolean mask indicating which rows have 'Engineering' as the department. Similarly, `employee_data['Salary'] > 70000` creates a boolean mask indicating which rows have a salary greater than \$70,000.

We then combined these boolean masks using the `&` (and) operator to filter the DataFrame, selecting only the rows where both conditions are True. Finally, we printed the resulting DataFrame, which contains employees who work in the 'Engineering' department and have a salary greater than \$70,000.

EXAMPLE 2.2

Filtering Sales Dataset

Let's say we have a sales dataset containing information about various products sold by an e-commerce company. The DataFrame `sales_data` has columns such as 'Product', 'Category', 'Quantity Sold', 'Unit Price', and 'Total Revenue'. We want to filter the DataFrame to select products from the 'Electronics' category with a quantity sold greater than 100.

```
import pandas as pd

# Sample sales data
data = {
    'Product': ['Laptop', 'Smartphone',
    'Headphones', 'Tablet', 'Smartwatch'],
    'Category': ['Electronics',
    'Electronics', 'Electronics', 'Electronics',
    'Wearable Tech'],
    'Quantity_Sold': [120, 90, 150, 80, 200],
    'Unit_Price': [1200, 800, 100, 400, 300],
    'Total_Revenue': [144000, 72000, 15000,
    32000, 60000]
}

sales_data = pd.DataFrame(data)

# Boolean indexing to filter products
electronics_high_quantity =
sales_data[(sales_data['Category'] ==
    'Electronics') & (sales_data['Quantity_Sold'] > 100)]

print(electronics_high_quantity)
```

Output:

| | Product | Category | Quantity_Sold |
|------------|---------------|-------------|---------------|
| Unit_Price | Total_Revenue | | |
| 0 | Laptop | Electronics | 120 |
| 1200 | 144000 | | |
| 2 | Headphones | Electronics | 150 |
| 100 | 15000 | | |

In this example, we used boolean indexing to filter the DataFrame sales_data. The expression sales_data['Category'] == 'Electronics' creates a boolean mask indicating which rows have 'Electronics' as the category. Similarly, sales_data['Quantity_Sold'] > 100 creates a boolean mask indicating which rows have a quantity sold greater than 100.

We then combined these boolean masks using the & (and) operator to filter the DataFrame, selecting only the rows where both conditions are True. Finally, we printed the resulting DataFrame, which contains products from the 'Electronics' category with a quantity sold greater than 100.

EXAMPLE 2.3

Filtering Movies Dataset

Let's consider a dataset containing information about movies, including their titles, genres, release years, ratings, and box office earnings. Suppose the DataFrame movie_data represents this dataset, and we want to filter the DataFrame to select movies released after 2010 with a rating higher than 8.

```
import pandas as pd
```

```

# Sample movie data
data = {
    'Title': ['Inception', 'The Dark Knight',
    'Interstellar', 'The Shawshank Redemption', 'Pulp
    Fiction'],
    'Genre': ['Sci-Fi', 'Action', 'Sci-Fi',
    'Drama', 'Crime'],
    'Release_Year': [2010, 2008, 2014, 1994,
    1994],
    'Rating': [8.8, 9.0, 8.6, 9.3, 8.9],
    'Box_Office_Earnings': [830000000, 1000000000,
    675000000, 28000000, 213900000]
}

movie_data = pd.DataFrame(data)

# Boolean indexing to filter movies
high_rated_recent_movies = movie_data[(movie_data['Release_Year'] > 2010) &
                                         (movie_data['Rating'] > 8)]

print(high_rated_recent_movies)

```

Output:

| | Title | Genre | Release_Year | Rating |
|---------------------|--------------|--------|--------------|--------|
| Box_Office_Earnings | | | | |
| 2 | Interstellar | Sci-Fi | 2014 | 8.6 |
| 675000000 | | | | |

In this example, we used boolean indexing to filter the DataFrame movie_data. The expression movie_data['Release_Year'] > 2010 creates a boolean mask indicating which rows have a release year

after 2010. Similarly, `movie_data['Rating'] > 8` creates a boolean mask indicating which rows have a rating higher than 8.

We then combined these boolean masks using the `&` (and) operator to filter the DataFrame, selecting only the rows where both conditions are True. Finally, we printed the resulting DataFrame, which contains movies released after 2010 with a rating higher than 8. In this case, only the movie "Interstellar" meets the criteria.

EXAMPLE 2.4

GUI Tkinter for Filtering Sales Dataset

This project is a GUI application built using Tkinter, Pandas, and PandasTable libraries in Python. It allows users to interact with sales data presented in a table format, filter the data based on certain criteria, and refresh the table to its original state.

```
import tkinter as tk
from tkinter import ttk
from pandastable import Table,TableModel
import pandas as pd

def filter_data():
    category = category_combobox.get()
    quantity = int(quantity_entry.get())

        # Filter sales data based on selected category
        and quantity
        filtered_data =
sales_data[(sales_data['Category'] == category) &
(sales_data['Quantity_Sold'] > quantity)]

        # Update PandasTable with filtered data
table.model.df = filtered_data
```

```
table.redraw()

def refresh_data():
    # Reset the table with the original DataFrame
    table.model.df = sales_data
    table.redraw()

# Sample sales data
data = {
    'Product': ['Laptop', 'Smartphone',
    'Headphones', 'Tablet', 'Smartwatch'],
    'Category': ['Electronics', 'Electronics',
    'Electronics', 'Electronics', 'Wearable Tech'],
    'Quantity_Sold': [120, 90, 150, 80, 200],
    'Unit_Price': [1200, 800, 100, 400, 300],
    'Total_Revenue': [144000, 72000, 15000, 32000,
    60000]
}

sales_data = pd.DataFrame(data)

# Create tkinter window
root = tk.Tk()
root.title("Sales Data Filter")

# Create a frame for user input
input_frame = tk.Frame(root)
input_frame.pack(pady=10)

# Category Label and Combobox
category_label = tk.Label(input_frame,
text="Select Category:")
category_label.grid(row=0, column=0, padx=5,
pady=5, sticky="e")
```

```
categories =
tuple(sales_data['Category'].unique()) # Convert
to tuple
category_combobox = ttk.Combobox(input_frame,
values=categories, state="readonly")
category_combobox.grid(row=0, column=1, padx=5,
pady=5)

# Quantity Label and Entry
quantity_label = tk.Label(input_frame, text="Enter
Quantity:")
quantity_label.grid(row=1, column=0, padx=5,
pady=5, sticky="e")

quantity_entry = tk.Entry(input_frame)
quantity_entry.grid(row=1, column=1, padx=5,
pady=5)

# Filter Button
filter_button = tk.Button(input_frame,
text="Filter Data", command=filter_data)
filter_button.grid(row=2, columnspan=2, pady=5)

# Refresh Button
refresh_button = tk.Button(input_frame,
text="Refresh", command=refresh_data)
refresh_button.grid(row=3, columnspan=2, pady=5)

# Create a frame for displaying the table
table_frame = tk.Frame(root)
table_frame.pack(padx=10, pady=10)

# Initialize PandasTable with initial DataFrame
initial_table_model =TableModel()
initial_table_model.df = sales_data
```

```
table = Table(table_frame,  
model=initial_table_model, showtoolbar=True,  
showstatusbar=True)  
table.show()  
  
root.mainloop()
```

Here's an overview of the components and functionality of the project:

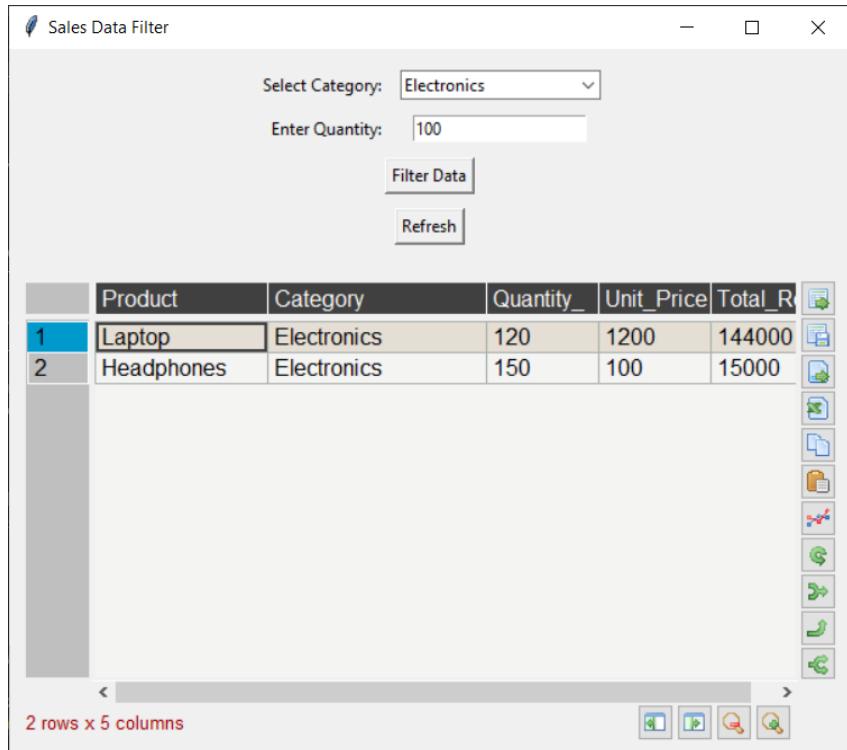
1. Importing Libraries: The project starts by importing necessary libraries including tkinter, ttk from tkinter for GUI components, pandas for data manipulation, and pandastable for displaying DataFrame in a table format.
2. Defining Functions:
 - filter_data(): This function is called when the "Filter Data" button is clicked. It retrieves the selected category and quantity entered by the user, filters the sales data DataFrame based on these criteria, and updates the PandasTable with the filtered data.
 - refresh_data(): This function is called when the "Refresh" button is clicked. It resets the PandasTable to display the original sales data DataFrame.
3. Sample Sales Data: A sample sales dataset is defined as a dictionary containing information about products, categories, quantity sold, unit price, and total revenue. This data is then converted into a pandas DataFrame.
4. Creating GUI Components:
 - root: The main tkinter window is created.
 - input_frame: A frame for user input components such as category combobox, quantity entry, and

buttons.

- category_label, categories, category_combobox: Label, combobox, and values for selecting a category.
- quantity_label, quantity_entry: Label and entry widget for entering quantity.
- filter_button, refresh_button: Buttons for filtering data and refreshing the table.
- table_frame: A frame for displaying the PandasTable.

5. Initializing PandasTable: The initial PandasTable is created using the TableModel initialized with the original sales data DataFrame. This table is displayed in the GUI with toolbar and status bar enabled.
6. Main Loop: The tkinter event loop (root.mainloop()) is started to run the application, allowing users to interact with the GUI components.

In summary, this project provides a simple GUI interface for users to filter and visualize sales data. Users can select a category and enter a quantity to filter the data, and they can also refresh the table to view the original data.



EXAMPLE 2.5

GUI Tkinter for Filtering Movies Dataset

This project is a GUI application built using Tkinter, Pandas, and PandasTable libraries in Python. It allows users to interact with movie data presented in a table format, filter the data based on minimum release year and minimum rating, and refresh the table to its original state.

```
import tkinter as tk
from tkinter import ttk
from pandastable import Table,TableModel
import pandas as pd

def filter_data():
    min_year = int(min_year_entry.get())
    min_rating = float(min_rating_entry.get())
```

```
# Filter movie data based on release year and rating
    filtered_data =
movie_data[(movie_data['Release_Year'] > min_year) &
(movie_data['Rating'] > min_rating)]

# Update PandasTable with filtered data
table.model.df = filtered_data
table.redraw()

def refresh_data():
    # Reset the table with the original DataFrame
    table.model.df = movie_data
    table.redraw()

# Sample movie data
data = {
    'Title': ['Inception', 'The Dark Knight', 'Interstellar', 'The Shawshank Redemption', 'Pulp Fiction'],
    'Genre': ['Sci-Fi', 'Action', 'Sci-Fi', 'Drama', 'Crime'],
    'Release_Year': [2010, 2008, 2014, 1994, 1994],
    'Rating': [8.8, 9.0, 8.6, 9.3, 8.9],
    'Box_Office_Earnings': [830000000, 1000000000, 675000000, 28000000, 213900000]
}

movie_data = pd.DataFrame(data)

# Create tkinter window
root = tk.Tk()
root.title("Movie Data Filter")
```

```
# Create a frame for user input
input_frame = tk.Frame(root)
input_frame.pack(pady=10)

# Minimum Year Label and Entry
min_year_label = tk.Label(input_frame,
text="Minimum Release Year:")
min_year_label.grid(row=0, column=0, padx=5,
pady=5, sticky="e")

min_year_entry = tk.Entry(input_frame)
min_year_entry.grid(row=0, column=1, padx=5,
pady=5)

# Minimum Rating Label and Entry
min_rating_label = tk.Label(input_frame,
text="Minimum Rating:")
min_rating_label.grid(row=1, column=0, padx=5,
pady=5, sticky="e")

min_rating_entry = tk.Entry(input_frame)
min_rating_entry.grid(row=1, column=1, padx=5,
pady=5)

# Filter Button
filter_button = tk.Button(input_frame,
text="Filter Data", command=filter_data)
filter_button.grid(row=2, columnspan=2, pady=5)

# Refresh Button
refresh_button = tk.Button(input_frame,
text="Refresh", command=refresh_data)
refresh_button.grid(row=3, columnspan=2, pady=5)

# Create a frame for displaying the table
```

```
table_frame = tk.Frame(root)
table_frame.pack(padx=10, pady=10)

# Initialize PandasTable with initial DataFrame
initial_table_model =TableModel()
initial_table_model.df = movie_data
table = Table(table_frame,
model=initial_table_model, showtoolbar=True,
showstatusbar=True)
table.show()

root.mainloop()
```

Here's the explanation of the components and functionality of the project:

1. Importing Libraries: The project starts by importing necessary libraries including tkinter, ttk from tkinter for GUI components, pandas for data manipulation, and pandastable for displaying DataFrame in a table format.
2. Defining Functions:
 - filter_data(): This function is called when the "Filter Data" button is clicked. It retrieves the minimum release year and minimum rating entered by the user, filters the movie data DataFrame based on these criteria, and updates the PandasTable with the filtered data.
 - refresh_data(): This function is called when the "Refresh" button is clicked. It resets the PandasTable to display the original movie data DataFrame.
3. Sample Movie Data: A sample movie dataset is defined as a dictionary containing information about movie titles,

genres, release years, ratings, and box office earnings. This data is then converted into a pandas DataFrame.

4. Creating GUI Components:

- root: The main tkinter window is created.
- input_frame: A frame for user input components such as entry widgets for minimum release year and minimum rating, and buttons for filtering and refreshing data.
- min_year_label, min_year_entry: Label and entry widget for entering minimum release year.
- min_rating_label, min_rating_entry: Label and entry widget for entering minimum rating.
- filter_button, refresh_button: Buttons for filtering data and refreshing the table.
- table_frame: A frame for displaying the PandasTable.

5. Initializing PandasTable: The initial PandasTable is created using the TableModel initialized with the original movie data DataFrame. This table is displayed in the GUI with toolbar and status bar enabled.

6. Main Loop: The tkinter event loop (root.mainloop()) is started to run the application, allowing users to interact with the GUI components.

In summary, this project provides a simple GUI interface for users to filter and visualize movie data. Users can input minimum release year and minimum rating to filter the data, and they can also refresh the table to view the original data.

Movie Data Filter

Minimum Release Year:

Minimum Rating:

Filter Data

Refresh

| | Title | Genre | Release_ | Rating | |
|---|--------------------------|--------|----------|--------|---|
| 1 | Inception | Sci-Fi | 2010 | 8.80 |  |
| 2 | The Dark Knight | Action | 2008 | 9.00 |  |
| 3 | Interstellar | Sci-Fi | 2014 | 8.60 |  |
| 4 | The Shawshank Redemption | Drama | 1994 | 9.30 |  |
| 5 | Pulp Fiction | Crime | 1994 | 8.90 |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | |  |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

LOGICAL OPERATORS

Logical operators in DataFrame operations are used to perform element-wise comparisons between two DataFrames or between a DataFrame and a scalar value. These operators help in filtering, masking, or modifying data based on specified conditions.

Here are the main logical operators used in DataFrame operations:

1. Element-wise Comparison:

- `==` (equals): Returns True if elements are equal.
- `!=` (not equals): Returns True if elements are not equal.
- `>` (greater than): Returns True if left operand is greater than right operand.
- `<` (less than): Returns True if left operand is less than right operand.
- `>=` (greater than or equal to): Returns True if left operand is greater than or equal to right operand.
- `<=` (less than or equal to): Returns True if left operand is less than or equal to right operand.

2. Element-wise Logical Operators:

- `&` (and): Element-wise logical AND operator. Returns True if both elements are True.
- `|` (or): Element-wise logical OR operator. Returns True if at least one of the elements is True.
- `~` (not): Element-wise logical NOT operator. Returns True if element is False.

These operators can be used to create boolean masks or filters to subset data based on specific conditions. For example, to filter rows in a DataFrame where a certain column meets multiple conditions, logical operators can be combined:

```
# Filter DataFrame for rows where 'Age' is greater than 30 and 'Gender' is 'Female'  
filtered_data = df[(df['Age'] > 30) & (df['Gender'] == 'Female')]
```

Similarly, these operators can also be used to modify DataFrame values based on conditions:

```
# Update 'Status' column to 'High' for rows where 'Sales' is greater than 1000  
df.loc[df['Sales'] > 1000, 'Status'] = 'High'
```

Overall, logical operators in DataFrame operations are essential for performing conditional operations, filtering, and data manipulation based on specified criteria.

EXAMPLE 2.6

Identifying Popular and Profitable Products

Imagine you're managing a retail store and you have a DataFrame containing sales data of various products. You want to identify products that are both popular (sold more than 100 units) and profitable (have generated revenue greater than \$5000). You can use logical operators to filter the DataFrame based on these conditions.

Here's how you can do it:

```
import pandas as pd  
  
# Sample sales data  
data = {
```

```

        'Product': ['Laptop', 'Smartphone',
'Headphones', 'Tablet', 'Smartwatch'],
        'Category': ['Electronics', 'Electronics',
'Electronics', 'Electronics', 'Wearable Tech'],
        'Quantity_Sold': [120, 90, 150, 80, 200],
        'Unit_Price': [1200, 800, 100, 400, 300],
        'Total_Revenue': [144000, 72000, 15000, 32000,
60000]
}

sales_data = pd.DataFrame(data)

# Filter products that are popular and profitable
popular_profitable_products =
sales_data[(sales_data['Quantity_Sold'] > 100) &
(sales_data['Total_Revenue'] > 5000)]

print(popular_profitable_products)

```

In this example, we're using the `&` logical operator to filter the DataFrame `sales_data` to include only rows where both conditions are true: `Quantity_Sold` greater than 100 units and `Total_Revenue` greater than \$5000.

The resulting DataFrame `popular_profitable_products` will contain only the rows corresponding to products that are both popular and profitable, meeting the specified criteria. This filtered DataFrame can then be used for further analysis or decision-making in the retail store management.

EXAMPLE 2.7

GUI Tkinter for Identifying Popular and Profitable Products

This project creates a Tkinter-based GUI application that displays sales data in a table and visualizes it with a bar graph. The GUI allows users to filter the data based on specific criteria and dynamically updates the table and graph accordingly.

```
import tkinter as tk
from tkinter import ttk
from pandastable import Table,TableModel
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

def filter_data():
    min_quantity = int(min_quantity_entry.get())
    min_revenue = int(min_revenue_entry.get())

        # Filter sales data based on quantity sold and
        total revenue
        filtered_data =
sales_data[(sales_data['Quantity_Sold'] >
min_quantity) &

(sales_data['Total_Revenue'] > min_revenue)]

        # Update PandasTable with filtered data
table.model.df = filtered_data
table.redraw()

        # Update bar graph
update_bar_graph(filtered_data)

def refresh_data():
    # Reset the table with the original DataFrame
table.model.df = sales_data
```

```
table.redraw()

# Update bar graph with original data
update_bar_graph(sales_data)

def update_bar_graph(data):
    # Group data by product and calculate total
    revenue
    product_revenue = data.groupby('Product')
    ['Total_Revenue'].sum()

    # Create bar graph
    fig, ax = plt.subplots()
    product_revenue.plot(kind='bar', ax=ax)
    ax.set_ylabel('Total Revenue')
    ax.set_title('Product vs. Revenue')

    # Clear previous canvas content
    for widget in graph_frame.winfo_children():
        widget.destroy()

    # Embed bar graph into Tkinter GUI
    canvas = FigureCanvasTkAgg(fig,
master=graph_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(side=tk.TOP,
fill=tk.BOTH, expand=1)

# Sample sales data
data = {
    'Product': ['Laptop', 'Smartphone',
'Headphones', 'Tablet', 'Smartwatch'],
    'Category': ['Electronics', 'Electronics',
'Electronics', 'Electronics', 'Wearable Tech'],
    'Quantity_Sold': [120, 90, 150, 80, 200],
    'Unit_Price': [1200, 800, 100, 400, 300],
```

```
'Total_Revenue' : [144000, 72000, 15000, 32000,  
60000]  
}  
  
sales_data = pd.DataFrame(data)  
  
# Create tkinter window  
root = tk.Tk()  
root.title("Sales Data Filter")  
  
# Create a frame for user input  
input_frame = tk.Frame(root)  
input_frame.pack(pady=10)  
  
# Minimum Quantity Label and Entry  
min_quantity_label = tk.Label(input_frame,  
text="Minimum Quantity Sold:")  
min_quantity_label.grid(row=0, column=0, padx=5,  
pady=5, sticky="e")  
  
min_quantity_entry = tk.Entry(input_frame)  
min_quantity_entry.grid(row=0, column=1, padx=5,  
pady=5)  
  
# Minimum Revenue Label and Entry  
min_revenue_label = tk.Label(input_frame,  
text="Minimum Total Revenue:")  
min_revenue_label.grid(row=1, column=0, padx=5,  
pady=5, sticky="e")  
  
min_revenue_entry = tk.Entry(input_frame)  
min_revenue_entry.grid(row=1, column=1, padx=5,  
pady=5)  
  
# Filter Button
```

```
filter_button = tk.Button(input_frame,
text="Filter Data", command=filter_data)
filter_button.grid(row=2, columnspan=2, pady=5)

# Refresh Button
refresh_button = tk.Button(input_frame,
text="Refresh", command=refresh_data)
refresh_button.grid(row=3, columnspan=2, pady=5)

# Create a frame for displaying the table and
graph side by side
display_frame = tk.Frame(root)
display_frame.pack(padx=10, pady=10, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the table on the
left side
table_frame = tk.Frame(display_frame)
table_frame.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the graph on the
right side
graph_frame = tk.Frame(display_frame)
graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH,
expand=True)

# Initialize PandasTable with initial DataFrame
initial_table_model =TableModel()
initial_table_model.df = sales_data
table = Table(table_frame,
model=initial_table_model, showtoolbar=True,
showstatusbar=True)
table.show()

# Update bar graph initially
```

```
update_bar_graph(sales_data)

root.mainloop()
```

Here's a detailed breakdown of the project:

Components and Functionality

1. Libraries Used:

- tkinter and ttk for creating the GUI elements.
- pandastable for displaying data in a table format within Tkinter.
- pandas for data manipulation.
- matplotlib for plotting the bar graph.
- FigureCanvasTkAgg from matplotlib.backends.backend_tkagg for embedding the Matplotlib graph into the Tkinter GUI.

2. Sample Data:

The sales_data DataFrame contains sample sales data, including columns for product names, categories, quantity sold, unit prices, and total revenue.

3. GUI Layout:

- User Input Frame: Contains input fields for the user to enter minimum quantity and minimum revenue values to filter the data.
- Display Frame: Contains two sub-frames:
 - Table Frame: Displays the sales data in a table format.
 - Graph Frame: Displays a bar graph of product versus revenue.

4. Functions:

- filter_data(): Filters the sales data based on user input for minimum quantity and revenue. Updates

both the table and the bar graph with the filtered data.

- `refresh_data()`: Resets the table and bar graph to display the original data.
- `update_bar_graph(data)`: Creates and displays a bar graph of product versus total revenue. This function is called to update the graph whenever the data changes (either filtered or refreshed).

5. GUI Elements:

- Labels and Entries: For user inputs related to minimum quantity and revenue.
- Buttons: For filtering the data and refreshing the view.
- PandasTable: Displays the sales data in a table format within the `table_frame`.
- Matplotlib Bar Graph: Displays a bar graph in the `graph_frame` to visually represent total revenue by product.

Detailed Steps:

1. Initialization:

- A Tkinter window is created, and its title is set to "Sales Data Filter".
- The `input_frame` is created to collect user inputs for filtering the data.
- Labels and entries for minimum quantity and minimum revenue are added to the `input_frame`.
- Buttons for filtering the data and refreshing the view are also added to the `input_frame`.

2. Display Frames:

- The display_frame is used to hold both the table and the graph side by side.
- table_frame and graph_frame are used to separate the table and the graph, respectively.

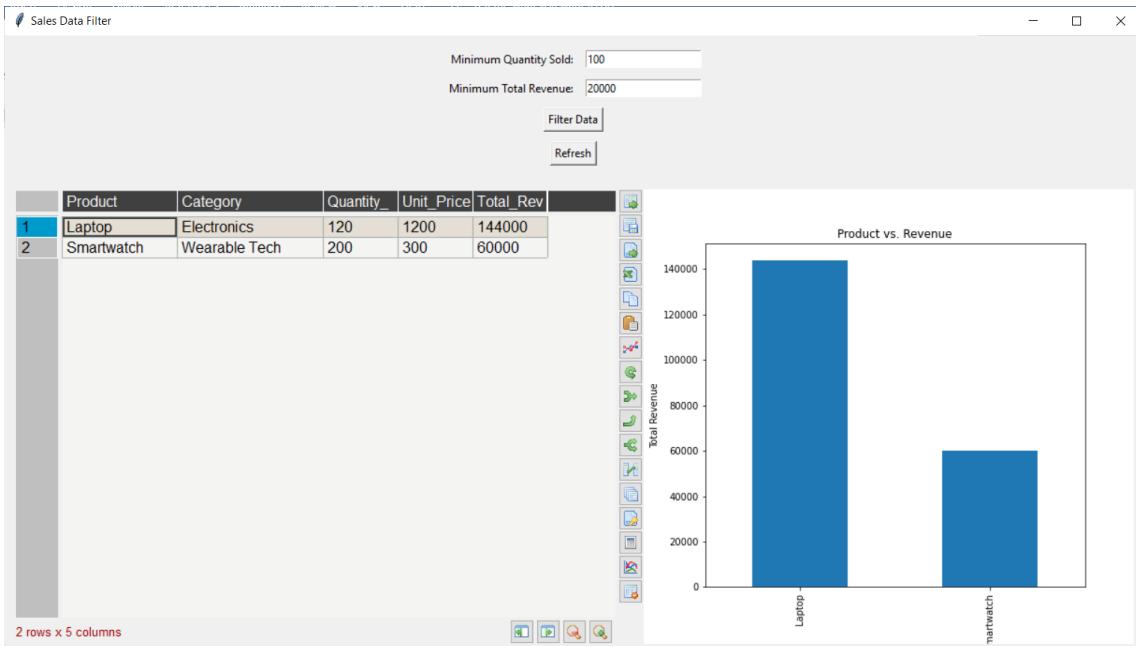
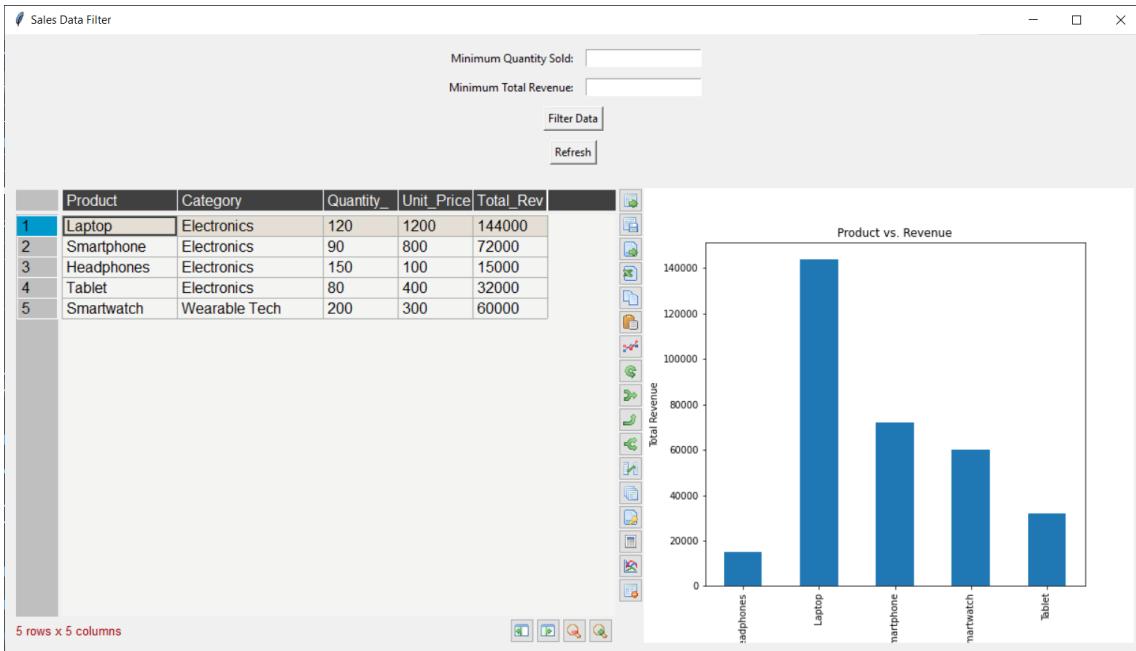
3. Table and Graph Initialization:

- The PandasTable is initialized with the sales_data DataFrame and displayed in table_frame.
- The update_bar_graph function is called initially to display the graph with the full dataset.

4. Event Handling:

- When the "Filter Data" button is clicked, the filter_data function is called to update the table and graph based on user input.
- When the "Refresh" button is clicked, the refresh_data function is called to reset the table and graph to the original data.

This project effectively demonstrates how to combine data manipulation, table display, and graphical visualization within a Tkinter-based GUI.



EXAMPLE 2.8

Analyzing Employee Performance

Imagine a company wants to analyze employee performance based on several criteria to identify top performers, areas for improvement, and those who may need additional support. The dataset includes employee details such as performance scores, years of experience, and project completions.

Goals

1. Identify high performers: Employees with a high performance score, at least 5 years of experience, and who have completed more than 10 projects.
2. Identify employees needing improvement: Employees with a low performance score, less than 3 years of experience, and have completed fewer than 7 projects.
3. Highlight overworked employees: Employees working more than 40 hours per week with a performance score of less than 70.

```
import pandas as pd

# Sample employee performance data
data = {
    'Employee_ID': [101, 102, 103, 104, 105, 106,
107, 108, 109, 110],
    'Name': ['Alice', 'Bob', 'Charlie', 'David',
'Eva', 'Frank', 'Grace', 'Hannah', 'Ivy', 'Jack'],
    'Performance_Score': [88, 75, 92, 85, 77, 60,
82, 90, 70, 66],
    'Years_of_Experience': [5, 7, 4, 6, 8, 2, 5,
6, 3, 1],
    'Projects_Completed': [12, 8, 15, 10, 7, 5,
11, 13, 6, 4],
    'Hours_Worked_Per_Week': [42, 38, 45, 40, 35,
30, 41, 44, 33, 28]
```

```
}
```

```
employee_data = pd.DataFrame(data)
```

```
# 1. Identify high performers
```

```
high_performers = employee_data[
```

```
    (employee_data['Performance_Score'] > 85) &
```

```
    (employee_data['Years_of_Experience'] >= 5) &
```

```
    (employee_data['Projects_Completed'] > 10)
```

```
]
```

```
# 2. Identify employees needing improvement
```

```
needing_improvement = employee_data[
```

```
    (employee_data['Performance_Score'] < 70) &
```

```
    (employee_data['Years_of_Experience'] < 3) &
```

```
    (employee_data['Projects_Completed'] < 7)
```

```
]
```

```
# 3. Highlight overworked employees
```

```
overworked_employees = employee_data[
```

```
    (employee_data['Hours_Worked_Per_Week'] > 40)
```

```
&
```

```
    (employee_data['Performance_Score'] < 70)
```

```
]
```

```
# Display results
```

```
print("High Performers:")
```

```
print(high_performers)
```

```
print("\nEmployees Needing Improvement:")
```

```
print(needing_improvement)
```

```
print("\nOverworked Employees:")
```

```
print(overworked_employees)
```

Explanation:

1. High Performers:

- Criteria: Performance score greater than 85, at least 5 years of experience, and more than 10 projects completed.
- Logical Operators Used:
 - & (AND) to combine multiple conditions for filtering high performers.

2. Employees Needing Improvement:

- Criteria: Performance score less than 70, less than 3 years of experience, and fewer than 7 projects completed.
- Logical Operators Used:
 - & (AND) to combine conditions to identify employees who may need more support or training.

3. Overworked Employees:

- Criteria: Working more than 40 hours per week and a performance score less than 70.
- Logical Operators Used:
 - & (AND) to find employees who might be overworked and underperforming.

EXAMPLE 2.9

GUI Tkinter for Analyzing Employee Performance

This Python project uses Tkinter to create a graphical user interface (GUI) for analyzing employee performance data. The interface allows users to filter data based on certain criteria and view the results in a table and a bar graph.

```
import tkinter as tk
from tkinter import ttk
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.colors as mcolors

def filter_data():
    min_performance =
int(min_performance_entry.get())
    min_experience =
int(min_experience_entry.get())
    min_projects = int(min_projects_entry.get())
    max_hours = int(max_hours_entry.get())

    # Filter high performers
    high_performers = employee_data[
        (employee_data['Performance_Score'] >
min_performance) &
        (employee_data['Years_of_Experience'] >=
min_experience) &
        (employee_data['Projects_Completed'] >
min_projects)
    ]

    # Filter employees needing improvement
    needing_improvement = employee_data[
        (employee_data['Performance_Score'] <
min_performance) &
        (employee_data['Years_of_Experience'] <
min_experience) &
        (employee_data['Projects_Completed'] <
min_projects)
    ]
```

```
# Filter overworked employees
overworked_employees = employee_data[
    (employee_data['Hours_Worked_Per_Week'] >
max_hours) &
    (employee_data['Performance_Score'] <
min_performance)
]

# Combine filtered data
combined_data = pd.concat([high_performers,
needing_improvement, overworked_employees])

# Update Treeview with filtered data
update_treeview(combined_data)

# Update bar graph
update_bar_graph(employee_data)

def refresh_data():
    # Reset the Treeview with the original
    DataFrame
    update_treeview(employee_data)

    # Update bar graph with original data
    update_bar_graph(employee_data)

def update_treeview(data):
    # Clear existing rows
    for row in tree.get_children():
        tree.delete(row)

    # Insert new rows with alternating row colors
    for index, row in data.iterrows():
        color = '#f0f0f0' if index % 2 == 0 else
'#ffffff'
```

```
        tree.insert('', 'end', values=list(row),
tags=('row%d' % index, ))
        tree.tag_configure('row%d' % index,
background=color)

def update_bar_graph(data):
    # Group data by department and calculate
    average performance score
    if 'Department' in data.columns:
        department_performance =
data.groupby('Department')
['Performance_Score'].mean()
    else:
        department_performance =
data['Performance_Score']

    # Create bar graph
    fig, ax = plt.subplots(figsize=(6, 4))
    department_performance.plot(kind='bar', ax=ax,
color='skyblue', label='Average Performance
Score')
    ax.set_ylabel('Average Performance Score')
    ax.set_title('Average Performance Score by
Department')
    ax.legend()

    # Clear previous canvas content
    for widget in graph_frame.winfo_children():
        widget.destroy()

    # Embed bar graph into Tkinter GUI
    canvas = FigureCanvasTkAgg(fig,
master=graph_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(side=tk.TOP,
fill=tk.BOTH, expand=1)
```

```
# Sample employee performance data
data = {
    'Employee_ID': [101, 102, 103, 104, 105, 106,
107, 108, 109, 110],
    'Name': ['Alice', 'Bob', 'Charlie', 'David',
'Eva', 'Frank', 'Grace', 'Hannah', 'Ivy', 'Jack'],
    'Performance_Score': [88, 75, 92, 85, 77, 60,
82, 90, 70, 66],
    'Years_of_Experience': [5, 7, 4, 6, 8, 2, 5,
6, 3, 1],
    'Projects_Completed': [12, 8, 15, 10, 7, 5,
11, 13, 6, 4],
    'Hours_Worked_Per_Week': [42, 38, 45, 40, 35,
30, 41, 44, 33, 28]
}

employee_data = pd.DataFrame(data)

# Create tkinter window
root = tk.Tk()
root.title("Employee Performance Analysis")

# Create a frame for user input
input_frame = tk.Frame(root)
input_frame.pack(pady=10, fill=tk.X)

# Minimum Performance Score Label and Entry
min_performance_label = tk.Label(input_frame,
text="Minimum Performance Score:")
min_performance_label.grid(row=0, column=0,
padx=5, pady=5, sticky="e")

min_performance_entry = tk.Entry(input_frame)
min_performance_entry.grid(row=0, column=1,
padx=5, pady=5)
```

```
# Minimum Experience Label and Entry
min_experience_label = tk.Label(input_frame,
text="Minimum Years of Experience:")
min_experience_label.grid(row=1, column=0, padx=5,
pady=5, sticky="e")

min_experience_entry = tk.Entry(input_frame)
min_experience_entry.grid(row=1, column=1, padx=5,
pady=5)

# Minimum Projects Label and Entry
min_projects_label = tk.Label(input_frame,
text="Minimum Projects Completed:")
min_projects_label.grid(row=2, column=0, padx=5,
pady=5, sticky="e")

min_projects_entry = tk.Entry(input_frame)
min_projects_entry.grid(row=2, column=1, padx=5,
pady=5)

# Maximum Hours Label and Entry
max_hours_label = tk.Label(input_frame,
text="Maximum Hours Worked Per Week:")
max_hours_label.grid(row=3, column=0, padx=5,
pady=5, sticky="e")

max_hours_entry = tk.Entry(input_frame)
max_hours_entry.grid(row=3, column=1, padx=5,
pady=5)

# Filter Button
filter_button = tk.Button(input_frame,
text="Filter Data", command=filter_data)
filter_button.grid(row=4, columnspan=2, pady=5)
```

```
# Refresh Button
refresh_button = tk.Button(input_frame,
text="Refresh", command=refresh_data)
refresh_button.grid(row=5, columnspan=2, pady=5)

# Create a frame for displaying the table and
graph side by side
display_frame = tk.Frame(root)
display_frame.pack(padx=10, pady=10, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the table on the
left side
table_frame = tk.Frame(display_frame)
table_frame.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the graph on the
right side
graph_frame = tk.Frame(display_frame)
graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH,
expand=True)

# Define columns and create Treeview
columns = list(employee_data.columns)
tree = ttk.Treeview(table_frame, columns=columns,
show='headings')
for col in columns:
    tree.heading(col, text=col)
    tree.column(col, width=120, anchor='center')
tree.pack(fill=tk.BOTH, expand=True)

# Update Treeview initially
update_treeview(employee_data)

# Update bar graph initially
```

```
update_bar_graph(employee_data)

root.mainloop()
```

Key Components

1. Data Handling:

- The project uses pandas to handle and manipulate the employee data stored in a DataFrame.
- The data includes fields such as Employee_ID, Name, Performance_Score, Years_of_Experience, Projects_Completed, and Hours_Worked_Per_Week.

2. GUI Layout:

- Tkinter: The project uses Tkinter for creating the GUI. It includes input fields for filtering data, a table to display the filtered results, and a bar graph to visualize the performance data.
- Frames: The layout is organized into frames:
 - input_frame for user input fields and buttons.
 - display_frame for displaying the table and graph side by side.
 - table_frame for the table on the left side.
 - graph_frame for the graph on the right side.

3. User Input:

- Input Fields: Users can enter values to filter employees based on performance score, years of experience, projects completed, and hours worked per week.
- Filter Button: When clicked, it triggers the filter_data() function to filter the DataFrame based

on the input values.

- Refresh Button: Resets the table and graph to display the original unfiltered data.

4. Data Filtering:

- Filter Data: The `filter_data()` function filters employees into three categories:
 - High Performers: Employees who exceed the specified performance score, experience, and project completion thresholds.
 - Needing Improvement: Employees who fall below the specified thresholds.
 - Overworked Employees: Employees who work more hours than specified and have a low performance score.
- The filtered data is combined into a single `DataFrame` for display.

5. Treeview (Table):

- Display: Uses the `ttk.Treeview` widget to display the data in a table format.
- Alternating Row Colors: Rows in the table have alternating colors for better readability.
- Update Function: The `update_treeview()` function updates the table with filtered data and applies alternating row colors.

6. Bar Graph:

- Display: Uses `matplotlib` to create a bar graph showing the average performance score by department (if the `Department` column exists in the data).
- Legend: The graph includes a legend to identify the data being displayed.

- Update Function: The `update_bar_graph()` function updates the graph based on the current data.

Functions

1. filter_data():

- Retrieves input values.
- Filters the `employee_data` DataFrame based on criteria.
- Updates the table and graph with the filtered data.

2. refresh_data():

Resets the table and graph to display the original data.

3. update_treeview(data):

- Clears existing rows in the table.
- Inserts new rows with alternating colors.

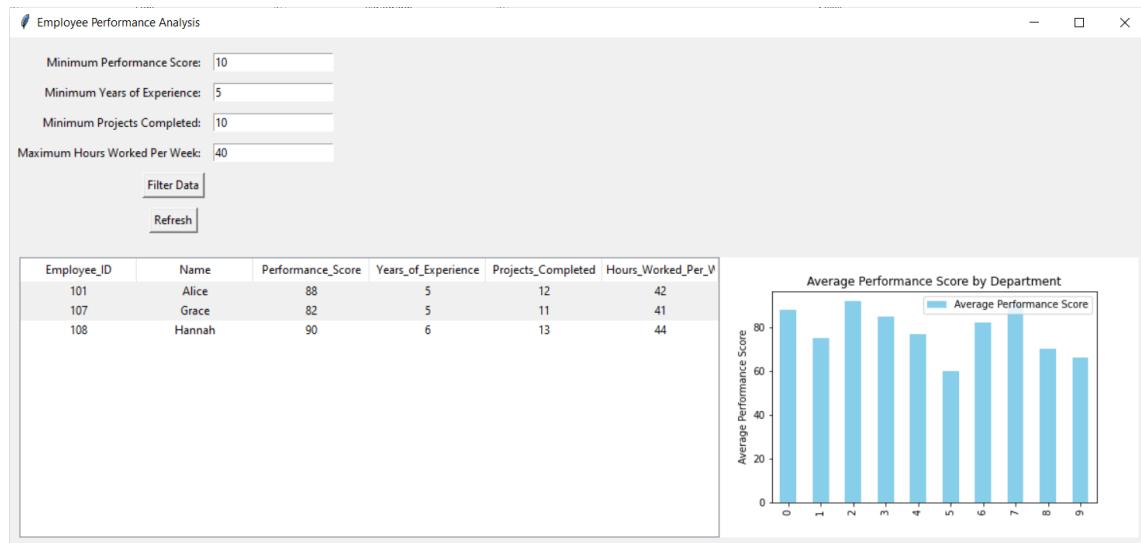
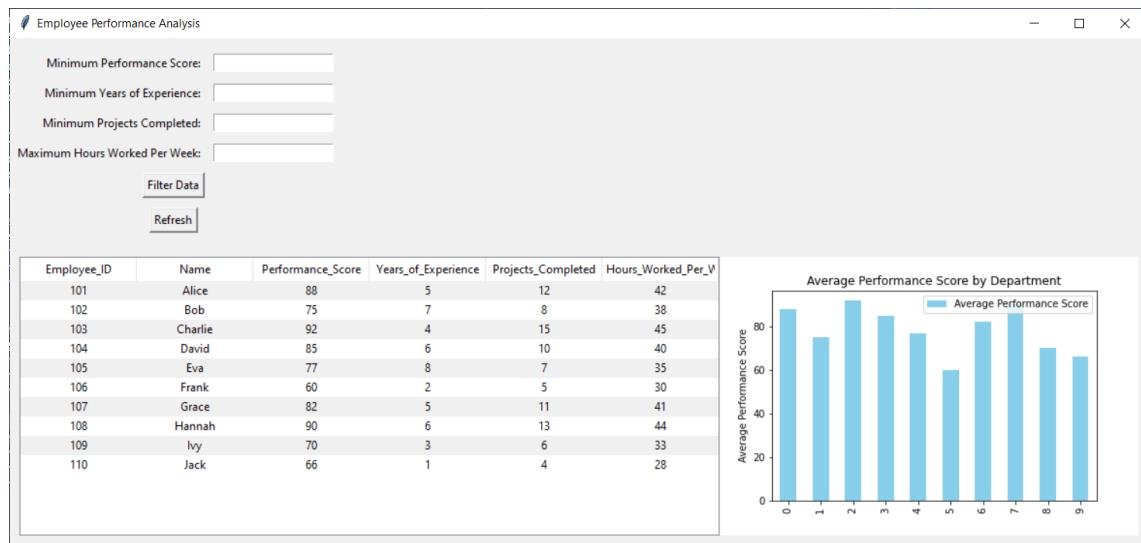
4. update_bar_graph(data):

- Groups data by department and calculates the average performance score.
- Creates and displays a bar graph.

Execution

- The `tk.Tk()` window is initialized, and various widgets are created and packed into the window.
- The Treeview and bar graph are updated initially to show the original data.
- The main loop (`root.mainloop()`) starts the Tkinter event loop, allowing the user to interact with the GUI.

This project provides a comprehensive solution for visualizing and analyzing employee performance data with interactive filtering capabilities.



QUERY METHOD

The `query()` method in pandas is a powerful tool for filtering DataFrames using a string expression. It allows you to select rows from your DataFrame based on conditions, similar to SQL queries, but written in a more Pythonic and readable way. The method is

particularly useful for filtering data when dealing with complex conditions.

Basic Syntax

```
DataFrame.query(expr, inplace=False, **kwargs)
```

- `expr`: A string expression that specifies the filtering conditions. The expression is evaluated against the columns of the DataFrame.
- `inplace`: If True, the DataFrame is filtered in place, and the original DataFrame is modified. If False (default), a new filtered DataFrame is returned.
- `**kwargs`: Additional arguments that can be passed, such as local variables or other parameters.

Features of query()

- String Expression: The expression is written as a string, making it intuitive and similar to writing SQL queries.
- Logical Operators: You can use logical operators like and, or, not, in, etc., within the expression.
- Variables in Expressions: You can include variables from the local environment in the query expression by using the @ symbol.
- Handling Column Names with Spaces: If your column names have spaces or special characters, you can refer to them using backticks: `Column Name`.

Examples

1. Basic Filtering

Suppose you have a DataFrame df with columns A, B, and C. You want to filter rows where the value in column A is greater than 5.

```
filtered_df = df.query('A > 5')
```

2. Combining Conditions

You can combine multiple conditions using logical operators.

```
# Filter rows where A > 5 and B < 10  
filtered_df = df.query('A > 5 and B < 10')
```

3. Using Variables in Expressions

If you want to filter based on a variable, you can do so by referencing the variable with the @ symbol.

```
threshold = 10  
filtered_df = df.query('A > @threshold')
```

4. Handling Special Column Names

If your DataFrame has columns with spaces or special characters, use backticks:

```
# Column with a space in its name  
filtered_df = df.query(`Column Name` > 5)
```

5. Using in Operator

You can filter rows where a column's value belongs to a list:

```
filtered_df = df.query('A in [1, 2, 3]')
```

Benefits of Using query()

- Readability: The query() method allows for more readable and concise code, especially when dealing with multiple conditions.

- Performance: For large DataFrames, `query()` can be faster than using the standard indexing method (`df[df['A'] > 5]`) because it is optimized for complex expressions.
- Flexibility: It supports complex filtering logic, such as using variables, and can handle columns with special characters without requiring additional transformations.

Limitations

- String-Based: Since the expression is a string, it doesn't benefit from Python's standard syntax checking until runtime, so errors in the string are harder to catch before execution.
- Limited Operators: The method uses a specific set of operators (and, or, not), so you can't use Python's bitwise operators (&, |, ~) directly in the expression.

The `query()` method is particularly useful when you want to filter data using clear and SQL-like expressions, making your code more readable and easier to maintain.

EXAMPLE 2.10

Filtering Data Population

Let's use population data to filter a DataFrame. Suppose you have a DataFrame with data about different cities, and you want to filter it based on specific criteria.

Scenario

You have a DataFrame containing information about cities, including their population, area, and continent. You want to filter the DataFrame to find cities with:

- A population greater than 1 million

- Located in either Asia or Europe
- Having an area larger than 500 square kilometers

Filtering with query()

You want to filter the DataFrame to find cities that meet the following criteria:

- Population greater than 1 million
- Located in either Asia or Europe
- Area larger than 500 square kilometers

```
import pandas as pd

# Sample city data
data = {
    'City': ['Tokyo', 'New York', 'London',
    'Shanghai', 'Mumbai', 'Paris', 'Berlin',
    'Beijing', 'Istanbul', 'Moscow'],
    'Continent': ['Asia', 'North America',
    'Europe', 'Asia', 'Asia', 'Europe', 'Europe',
    'Asia', 'Europe', 'Europe'],
    'Population': [13929286, 8419000, 8982000,
    24150000, 12478447, 2148000, 3645000, 21540000,
    15460000, 11920000],
    'Area_km2': [2194, 789, 1572, 6340, 603, 105,
    891, 16410, 5461, 2511]
}

df = pd.DataFrame(data)

# Filtering the DataFrame using query()
filtered_df = df.query('Population > 1000000 and
Continent in ["Asia", "Europe"] and Area_km2 >
500')
```

```
# Display the filtered DataFrame  
print(filtered_df)
```

Output

| | City | Continent | Population | Area_km2 |
|---|----------|-----------|------------|----------|
| 0 | Tokyo | Asia | 13929286 | 2194 |
| 2 | London | Europe | 8982000 | 1572 |
| 3 | Shanghai | Asia | 24150000 | 6340 |
| 4 | Mumbai | Asia | 12478447 | 603 |
| 6 | Berlin | Europe | 3645000 | 891 |
| 7 | Beijing | Asia | 21540000 | 16410 |
| 8 | Istanbul | Europe | 15460000 | 5461 |
| 9 | Moscow | Europe | 11920000 | 2511 |

Explanation

- query('Population > 1000000 and Continent in ["Asia", "Europe"] and Area_km2 > 500'): This line filters the DataFrame to include only rows where:
 - Population is greater than 1 million
 - Continent is either 'Asia' or 'Europe'
 - Area_km2 is larger than 500 square kilometers

The query() method provides a powerful way to perform complex filtering operations in a concise manner, especially useful for datasets with multiple conditions.

EXAMPLE 2.11

GUI Tkinter for Filtering Data Population

This Python script creates a graphical user interface (GUI) using the tkinter library, allowing users to filter and visualize city population data from a DataFrame.

```
import tkinter as tk
```

```
from tkinter import ttk
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.cm as cm
import numpy as np

# Sample city data
data = {
    'City': ['Tokyo', 'New York', 'London',
'Shanghai', 'Mumbai', 'Paris', 'Berlin',
'Beijing', 'Istanbul', 'Moscow'],
    'Continent': ['Asia', 'North America',
'Europe', 'Asia', 'Asia', 'Europe', 'Europe',
'Asia', 'Europe', 'Europe'],
    'Population': [13929286, 8419000, 8982000,
24150000, 12478447, 2148000, 3645000, 21540000,
15460000, 11920000],
    'Area_km2': [2194, 789, 1572, 6340, 603, 105,
891, 16410, 5461, 2511]
}

df = pd.DataFrame(data)

# Create tkinter window
root = tk.Tk()
root.title("City Population and Area Analysis")

# Create a frame for user input
input_frame = tk.Frame(root)
input_frame.pack(pady=10)

# Filter inputs
population_label = tk.Label(input_frame, text="Min
Population:")
```

```
population_label.grid(row=0, column=0, padx=5,
pady=5)
population_entry = tk.Entry(input_frame)
population_entry.grid(row=0, column=1, padx=5,
pady=5)
population_entry.insert(0, "1000000")

continent_label = tk.Label(input_frame,
text="Continent (comma-separated):")
continent_label.grid(row=1, column=0, padx=5,
pady=5)
continent_entry = tk.Entry(input_frame)
continent_entry.grid(row=1, column=1, padx=5,
pady=5)
continent_entry.insert(0, "Asia,Europe")

area_label = tk.Label(input_frame, text="Min Area
(km2):")
area_label.grid(row=2, column=0, padx=5, pady=5)
area_entry = tk.Entry(input_frame)
area_entry.grid(row=2, column=1, padx=5, pady=5)
area_entry.insert(0, "500")

# Function to filter DataFrame and update table
and graph
def filter_data():
    min_population = int(population_entry.get())
    continents = continent_entry.get().split(',')
    min_area = int(area_entry.get())

    query_str = f'Population > {min_population}
and Continent in {continents} and Area_km2 >
{min_area}'
    filtered_df = df.query(query_str)

    # Update Treeview with filtered data
```

```
update_treeview(filtered_df)
# Update bar graph with filtered data
update_bar_graph(filtered_df)

# Function to refresh DataFrame and reset table
and graph
def refresh_data():
    update_treeview(df)
    update_bar_graph(df)

# Create a frame for displaying the table and
graph side by side
display_frame = tk.Frame(root)
display_frame.pack(padx=10, pady=10, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the table on the
left side
table_frame = tk.Frame(display_frame)
table_frame.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the graph on the
right side
graph_frame = tk.Frame(display_frame)
graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH,
expand=True)

# Define columns and create Treeview
columns = list(df.columns)
tree = ttk.Treeview(table_frame, columns=columns,
show='headings')
for col in columns:
    tree.heading(col, text=col)
    tree.column(col, width=100, anchor='center')
tree.pack(fill=tk.BOTH, expand=True)
```

```
# Insert data into Treeview with alternating row colors
def update_treeview(data):
    # Clear existing rows
    for row in tree.get_children():
        tree.delete(row)

    # Insert new rows with alternating row colors
    for index, row in data.iterrows():
        color = '#f0f0f0' if index % 2 == 0 else '#ffffff'
        tree.insert('', 'end', values=list(row),
tags=('row%d' % index,))
        tree.tag_configure('row%d' % index,
background=color)

# Create a bar graph of Population vs City with different colors for each bar
def update_bar_graph(data):
    fig, ax = plt.subplots(figsize=(6, 4))

    # Generate colors using a colormap
    num_bars = len(data['City'])
    colors = cm.get_cmap('tab10')(np.linspace(0,
1, num_bars))

    # Plot each bar with a different color
    ax.bar(data['City'], data['Population'],
color=colors, label='Population')
    ax.set_xlabel('City')
    ax.set_ylabel('Population')
    ax.set_title('Population of Cities')
    ax.legend() # Turn on the legend

    # Clear previous canvas content
```

```

for widget in graph_frame.winfo_children():
    widget.destroy()

# Embed bar graph into Tkinter GUI
canvas = FigureCanvasTkAgg(fig,
master=graph_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(side=tk.TOP,
fill=tk.BOTH, expand=True)

# Filter and Refresh Buttons
filter_button = tk.Button(input_frame,
text="Filter", command=filter_data)
filter_button.grid(row=3, columnspan=2, pady=5)

refresh_button = tk.Button(input_frame,
text="Refresh", command=refresh_data)
refresh_button.grid(row=4, columnspan=2, pady=5)

# Update Treeview and Graph initially
update_treeview(df)
update_bar_graph(df)

root.mainloop()

```

Here's a step-by-step explanation of the code:

1. Imports and Setup

The program starts by importing necessary libraries:

- tkinter: The core library used to create the GUI.
- ttk: A themed widget set from tkinter for more modern looking components.
- pandas: Used to manage and manipulate the data in a tabular format (DataFrame).

- `matplotlib.pyplot`: Used to create visual plots like the bar graph.
- `FigureCanvasTkAgg`: A Tkinter-specific backend for embedding Matplotlib figures into the GUI.
- `matplotlib.cm`: Provides colormap functions for generating colors.
- `numpy`: Used here to help in generating a range of values for color mapping.

2. Creating Sample Data

A dictionary called `data` is created with city information, including city names, continents, populations, and area sizes. This dictionary is then converted into a pandas DataFrame, `df`, which makes it easy to manipulate and analyze the data.

3. Creating the Main Window

A tkinter window (`root`) is created as the main application window. The window is titled "City Population and Area Analysis".

4. Input Frame for User Filtering

- A frame (`input_frame`) is added to the window to hold user inputs:
- Population input: A label and entry widget allow users to input a minimum population value.
- Continent input: A label and entry widget for inputting a comma-separated list of continents.
- Area input: A label and entry widget for inputting a minimum area in square kilometers.

5. Data Filtering Function

A function (`filter_data`) is defined to:

- Retrieve the values from the entry widgets.
- Construct a query string based on the user inputs.
- Use pandas' `query()` method to filter the DataFrame according to the query string.

- Update both the data table and the bar graph using the filtered data.

6. Refreshing Data Function

Another function (`refresh_data`) is defined to reset the displayed data back to the original, unfiltered state. It does this by calling the functions that update the table and graph, passing the unfiltered DataFrame (`df`).

7. Creating Display Frames

Two frames (`table_frame` and `graph_frame`) are created to hold the data table and the graph side by side within the main window.

8. Data Table (Treeview)

A Treeview widget is created within `table_frame` to display the city data as a table:

- Column Setup: The columns are defined, and each column is given a header and a specific width.
- Data Insertion: A function (`update_treeview`) is defined to populate the Treeview with data. The rows are added with alternating background colors for better readability.

9. Bar Graph Creation

A function (`update_bar_graph`) is defined to create a bar graph:

- A Matplotlib figure and axes are created.
- Colors for the bars are generated using a colormap, ensuring each bar has a different color.
- The bar graph plots city names on the x-axis and population on the y-axis.
- The graph is embedded into the GUI using `FigureCanvasTkAgg`.

10. Filter and Refresh Buttons

Two buttons are created in the `input_frame`:

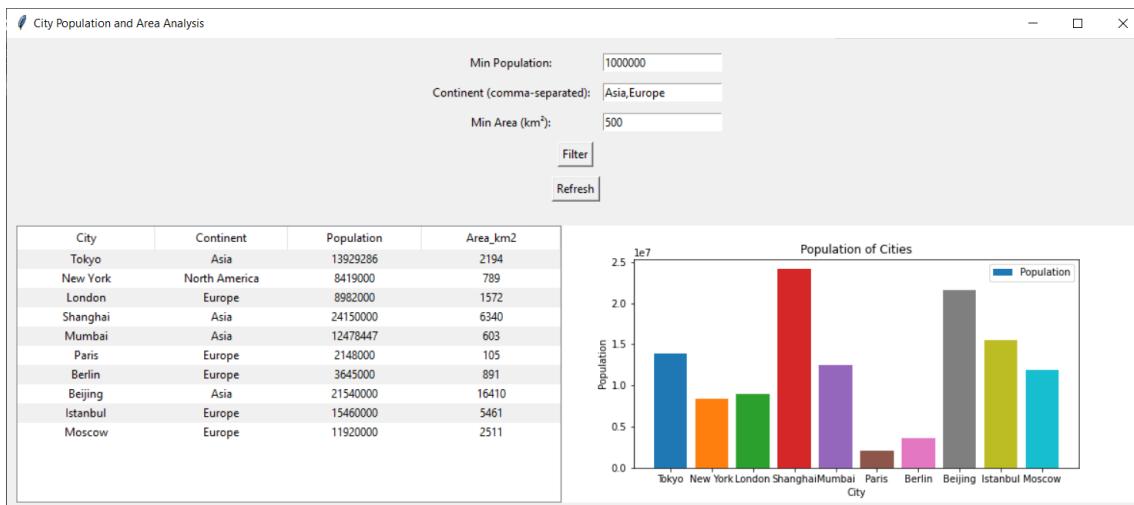
- Filter Button: Calls `filter_data()` when clicked, updating the table and graph based on the user's input.
- Refresh Button: Calls `refresh_data()` when clicked, resetting the table and graph to their original states.

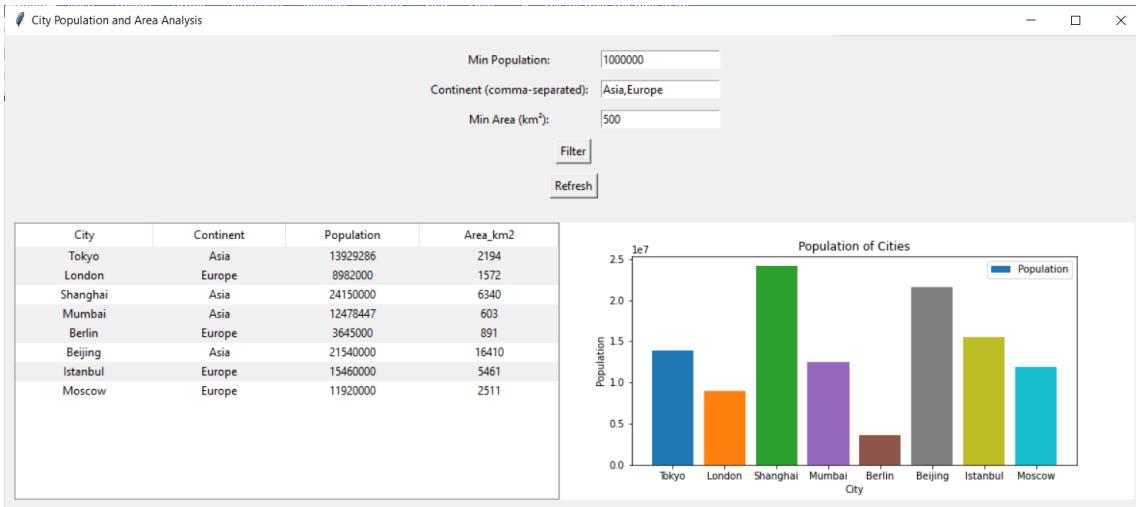
11. Initial Display

The table and graph are populated with the full dataset initially by calling `update_treeview(df)` and `update_bar_graph(df)`.

12. Main Application Loop

Finally, the `root.mainloop()` starts the Tkinter event loop, which keeps the application running and responsive to user inputs (like button clicks).





FILTERING COLUMNS

Filtering columns in a DataFrame refers to the process of selecting specific columns from the DataFrame based on certain conditions or requirements. This is commonly done to narrow down the data to only those columns that are relevant for a particular analysis or operation.

Methods to Filter Columns

1. Selecting Specific Columns by Name

You can filter columns by specifying the column names as a list.

```
selected_columns = df[['Column1', 'Column2']]
```

This will create a new DataFrame containing only Column1 and Column2 from the original DataFrame df.

2. Using Column Indexes

You can also filter columns using their index positions.

```
selected_columns = df.iloc[:, [0, 2]]
```

Here, `iloc[:, [0, 2]]` selects all rows but only the first and third columns (index starts from 0).

3. Filtering Columns Based on a Condition

You can apply a condition to filter columns. For example, you can filter columns based on their data types or column names that match a certain pattern.

- By Data Type:

```
numeric_columns = df.select_dtypes(include='number')
```

This selects only the columns with numeric data types.

- By Column Name Pattern:

```
columns_starting_with_C = df.filter(like='C')
```

This will select columns whose names contain the letter "C".

4. Dropping Unwanted Columns

Instead of selecting specific columns, you can drop the columns you don't need.

```
df_dropped = df.drop(['UnwantedColumn1',  
                      'UnwantedColumn2'], axis=1)
```

This removes UnwantedColumn1 and UnwantedColumn2 from the DataFrame `df`.

5. Filtering Columns Dynamically

Sometimes, you might want to filter columns dynamically based on some criteria or external input.

```
columns_to_keep = [col for col in df.columns if  
                  '2021' in col]  
df_filtered = df[columns_to_keep]
```

This example keeps only the columns whose names contain "2021".

EXAMPLE 2.12

Filtering Traffic Web Data

Let's use a DataFrame containing traffic web data. For this example, imagine you have a DataFrame with web traffic statistics including the date, number of visits, page views, and unique users. Here's how you might filter columns from such a DataFrame.

```
import pandas as pd

# Creating a DataFrame with web traffic data
data = {
    'Date': ['2024-08-01', '2024-08-02', '2024-08-03', '2024-08-04', '2024-08-05'],
    'Visits': [1200, 1300, 1400, 1500, 1600],
    'Page Views': [3000, 3200, 3100, 3300, 3400],
    'Unique Users': [800, 850, 900, 950, 1000],
    'Bounce Rate': [0.4, 0.35, 0.38, 0.37, 0.36],
    'Country': ['USA', 'Canada', 'USA', 'UK', 'Canada'],
    'Device Type': ['Mobile', 'Desktop', 'Mobile', 'Tablet', 'Desktop'],
    'Revenue': [5000, 6000, 5500, 7000, 6500]
}

df = pd.DataFrame(data)

# Filtering based on multiple conditions and
# selecting specific columns
filtered_df = df[(df['Country'] == 'USA') &
                  (df['Revenue'] > 5000)]
```

```
filtered_df = filtered_df[['Date', 'Visits',  
'Revenue']]  
  
print(filtered_df)
```

Output

| | Date | Visits | Revenue |
|---|------------|--------|---------|
| 2 | 2024-08-03 | 1400 | 5500 |

Here's an explanation of the process of the code:

1. Importing the Library:

Start by importing the pandas library, which is a popular tool for handling and analyzing data in Python.

2. Creating the DataFrame:

- Define a set of data with columns such as Date, Visits, Page Views, Unique Users, Bounce Rate, Country, Device Type, and Revenue. Each column is represented as a list of values.
- Combine these columns into a structured table called a DataFrame, which organizes the data in rows and columns.

3. Filtering the Data:

- Apply filters to select only the rows where the Country column has the value 'USA' and the Revenue column has a value greater than 5000.
- This step narrows down the DataFrame to only those rows meeting both conditions.

4. Selecting Specific Columns:

- From the filtered rows, choose only the columns of interest, such as Date, Visits, and Revenue.

- This reduces the DataFrame to just the relevant columns, focusing on the information needed.

5. Displaying the Result:

Print or display the final DataFrame to see the result of the filtering and column selection process. This shows the filtered data with only the selected columns.

In summary, you create a DataFrame from your data, filter it based on specific criteria, select only the columns you need, and then view the result.

EXAMPLE 2.13

GUI Tkinter for Filtering Traffic Web Data

This project is a graphical user interface (GUI) application built using Tkinter, designed to analyze and visualize web traffic data. The application allows users to filter data based on specific criteria and view the results in a table and several bar graphs. It combines data manipulation using pandas with visualization using matplotlib.

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

# Sample DataFrame
data = {
    'Date': ['2024-08-01', '2024-08-02', '2024-08-
03', '2024-08-04', '2024-08-05'],
    'Visits': [1200, 1300, 1400, 1500, 1600],
    'Page Views': [3000, 3200, 3100, 3300, 3400],
    'Unique Users': [800, 850, 900, 950, 1000],
```

```
'Bounce Rate': [0.4, 0.35, 0.38, 0.37, 0.36],  
'Country': ['USA', 'Canada', 'USA', 'UK',  
'Canada'],  
'Device Type': ['Mobile', 'Desktop', 'Mobile',  
'Tablet', 'Desktop'],  
'Revenue': [5000, 6000, 5500, 7000, 6500]  
}  
df = pd.DataFrame(data)  
  
# Create tkinter window  
root = tk.Tk()  
root.title("Web Traffic Data Analysis")  
  
# Create a frame for user input  
input_frame = tk.Frame(root)  
input_frame.pack(pady=10)  
  
# Filter inputs  
country_label = tk.Label(input_frame,  
text="Country:")  
country_label.grid(row=0, column=0, padx=5,  
pady=5)  
country_entry = ttk.Combobox(input_frame)  
country_entry.grid(row=0, column=1, padx=5,  
pady=5)  
  
revenue_label = tk.Label(input_frame, text="Min  
Revenue:")  
revenue_label.grid(row=1, column=0, padx=5,  
pady=5)  
revenue_entry = tk.Entry(input_frame)  
revenue_entry.grid(row=1, column=1, padx=5,  
pady=5)  
revenue_entry.insert(0, "5000")
```

```
# Function to update combobox with unique country values
def update_combobox():
    # Get unique countries from the DataFrame
    unique_countries = df['Country'].unique()

    # Ensure the list is clean and formatted as strings
    unique_countries = [str(country) for country
in unique_countries]

    # Update the Combobox with the cleaned list of countries
    country_entry['values'] = unique_countries

    # Set a default value if available
    if unique_countries:
        country_entry.set(unique_countries[0])

# Function to filter DataFrame and update table and graphs
def filter_data():
    country_filter = country_entry.get()
    revenue_filter = int(revenue_entry.get())

    filtered_df = df[(df['Country'] ==
country_filter) & (df['Revenue'] >
revenue_filter)]

    # Update Treeview with filtered data
    update_treeview(filtered_df)
    # Update bar graphs with filtered data
    update_bar_graphs(filtered_df)

# Function to refresh DataFrame and reset table and graphs
```

```
def refresh_data():
    update_combobox() # Update combobox with
current unique countries
    update_treeview(df)
    update_bar_graphs(df)

# Create a frame for displaying the table and
graphs side by side
display_frame = tk.Frame(root)
display_frame.pack(padx=10, pady=10, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the table on the
left side
table_frame = tk.Frame(display_frame)
table_frame.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the graphs on the
right side
graph_frame = tk.Frame(display_frame)
graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH,
expand=True)

# Define columns and create Treeview
columns = list(df.columns)
tree = ttk.Treeview(table_frame, columns=columns,
show='headings')
for col in columns:
    tree.heading(col, text=col)
    tree.column(col, width=100, anchor='center')
tree.pack(fill=tk.BOTH, expand=True)

# Insert data into Treeview
def update_treeview(data):
    # Clear existing rows
```

```

    for row in tree.get_children():
        tree.delete(row)

    # Insert new rows with alternating row colors
    for index, row in data.iterrows():
        color = '#f0f0f0' if index % 2 == 0 else
'#ffffff'
        tree.insert('', 'end', values=list(row),
tags=('row%d' % index,))
        tree.tag_configure('row%d' % index,
background=color)

# Create separate bar graphs for Visits, Page
Views, Unique Users, and Revenue
def update_bar_graphs(data):
    fig, axs = plt.subplots(2, 2, figsize=(10, 6),
dpi=100) # 2x2 grid of subplots
    fig.tight_layout(pad=5.0) # Adjust spacing
between subplots

    # Plot Visits
    axs[0, 0].bar(data['Date'], data['Visits'],
color='blue', label='Visits')
    axs[0, 0].set_title('Visits')
    axs[0, 0].set_xlabel('Date')
    axs[0, 0].set_ylabel('Visits')
    axs[0, 0].tick_params(axis='x', rotation=45,
labelsize=8) # Set font size of x-ticks
    axs[0, 0].tick_params(axis='y', labelsize=8)
    # Set font size of y-ticks
    for index, value in enumerate(data['Visits']):
        axs[0, 0].text(index, value + 50,
str(value), ha='center', va='bottom', fontsize=8)

    # Plot Page Views

```

```

        axs[0, 1].bar(data['Date'], data['Page
Views'], color='green', label='Page Views')
        axs[0, 1].set_title('Page Views')
        axs[0, 1].set_xlabel('Date')
        axs[0, 1].set_ylabel('Page Views')
        axs[0, 1].tick_params(axis='x', rotation=45,
labelsize=8) # Set font size of x-ticks
        axs[0, 1].tick_params(axis='y', labelsize=8)
# Set font size of y-ticks
        for index, value in enumerate(data['Page
Views']):
            axs[0, 1].text(index, value + 50,
str(value), ha='center', va='bottom', fontsize=8)

        # Plot Unique Users
        axs[1, 0].bar(data['Date'], data['Unique
Users'], color='orange', label='Unique Users')
        axs[1, 0].set_title('Unique Users')
        axs[1, 0].set_xlabel('Date')
        axs[1, 0].set_ylabel('Unique Users')
        axs[1, 0].tick_params(axis='x', rotation=45,
labelsize=8) # Set font size of x-ticks
        axs[1, 0].tick_params(axis='y', labelsize=8)
# Set font size of y-ticks
        for index, value in enumerate(data['Unique
Users']):
            axs[1, 0].text(index, value + 50,
str(value), ha='center', va='bottom', fontsize=8)

        # Plot Revenue
        axs[1, 1].bar(data['Date'], data['Revenue'],
color='red', label='Revenue')
        axs[1, 1].set_title('Revenue')
        axs[1, 1].set_xlabel('Date')
        axs[1, 1].set_ylabel('Revenue')

```

```
    axs[1, 1].tick_params(axis='x', rotation=45,
labelsize=8) # Set font size of x-ticks
    axs[1, 1].tick_params(axis='y', labelsize=8)
# Set font size of y-ticks
    for index, value in
enumerate(data['Revenue']):
        axs[1, 1].text(index, value + 50,
str(value), ha='center', va='bottom', fontsize=8)

    # Clear previous canvas content
    for widget in graph_frame.winfo_children():
        widget.destroy()

    # Embed bar graphs into Tkinter GUI
    canvas = FigureCanvasTkAgg(fig,
master=graph_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(side=tk.TOP,
fill=tk.BOTH, expand=True)

# Filter and Refresh Buttons
filter_button = tk.Button(input_frame,
text="Filter", command=filter_data)
filter_button.grid(row=2, columnspan=2, pady=5)

refresh_button = tk.Button(input_frame,
text="Refresh", command=refresh_data)
refresh_button.grid(row=3, columnspan=2, pady=5)

# Initialize Combobox, Treeview, and Graphs with
original data
update_combobox() # Initialize combobox with data
update_treeview(df)
update_bar_graphs(df)

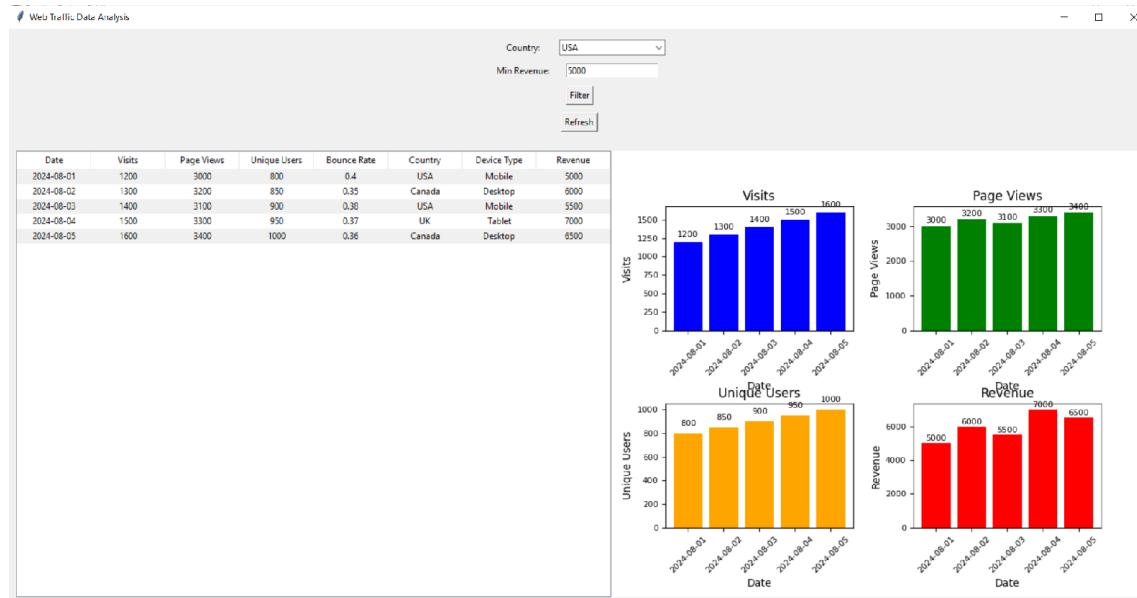
root.mainloop()
```

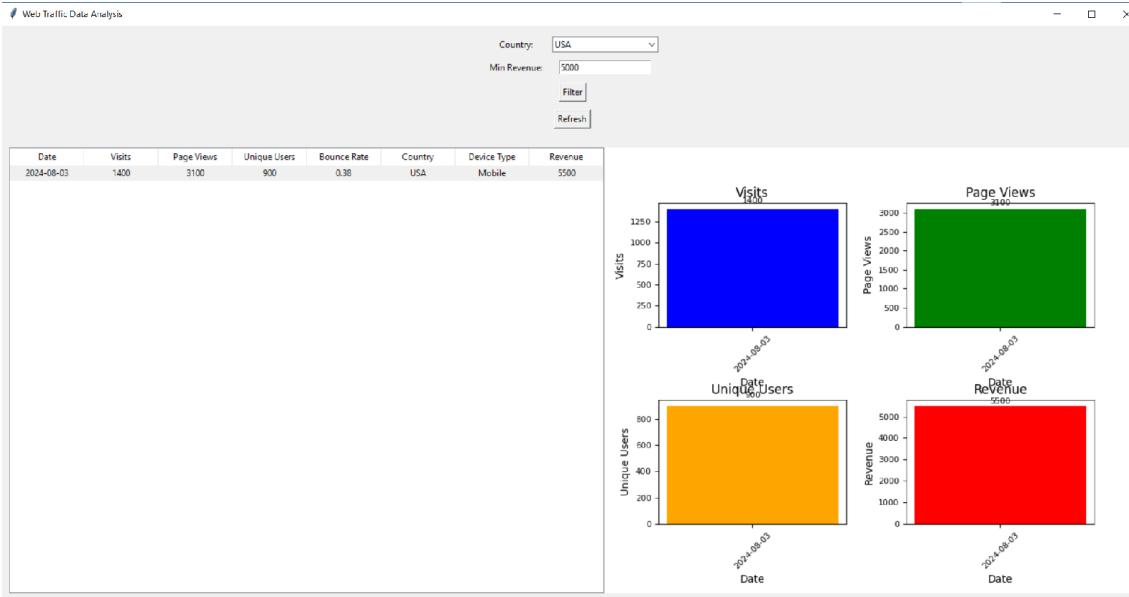
Components and Functionality

1. Data Preparation:

A sample dataset is created using a pandas DataFrame. This dataset simulates web traffic metrics and includes columns such as:

- Date: The date of the data entry.
- Visits: The number of visits on that date.
- Page Views: The number of page views on that date.
- Unique Users: The number of unique users on that date.
- Bounce Rate: The percentage of visitors who navigate away after viewing only one page.
- Country: The country of the users.
- Device Type: The type of device used (e.g., Mobile, Desktop, Tablet).
- Revenue: The revenue generated on that date.





2. Tkinter GUI Setup:

Main Window: The main application window is created with the title "Web Traffic Data Analysis".

3. Input Frame:

- Country Filter: A `ttk.Combobox` widget is used to select the country. This allows users to filter the data based on the country of the visitors.
- Revenue Filter: An `Entry` widget allows users to specify a minimum revenue threshold. Only entries with revenue above this threshold will be displayed.

4. Buttons:

- Filter Button: This button applies the selected filters (country and minimum revenue) to the dataset and updates the displayed data and graphs accordingly.
- Refresh Button: This button resets the filters to their default state and updates the displayed data and graphs with the original, unfiltered dataset.

5. Functions:

- `update_combobox()`: This function populates the country filter with unique country values extracted from the DataFrame. It ensures that all country names are formatted correctly and sets a default value if available.
- `filter_data()`: This function applies the filters selected by the user. It filters the DataFrame based on the chosen country and minimum revenue, then updates both the table and the bar graphs with the filtered data.
- `refresh_data()`: This function resets the filters and refreshes the data display to show the original dataset. It updates the combobox with all unique countries, and refreshes both the table and bar graphs.
- `update_treeview(data)`: This function updates a `ttk.Treeview` widget, which is used to display the dataset in a tabular format. It clears existing rows, then inserts the new rows from the filtered or original DataFrame. The rows alternate in color for better readability.
- `update_bar_graphs(data)`: This function creates and updates four bar graphs, each displayed in a 2x2 grid layout. The graphs show:
 - Visits: Number of visits for each date.
 - Page Views: Number of page views for each date.
 - Unique Users: Number of unique users for each date.
 - Revenue: Revenue generated for each date.
 - Each graph includes:
 - Titles and labels for axes.

- X-axis tick labels rotated for readability.
- Y-axis labels and tick sizes.
- Data values displayed on top of each bar for clarity.

6. Layout and Embedding:

- Display Frame: A frame (`display_frame`) is used to organize the table and graphs side by side. This frame contains:
 - Table Frame: Holds the `ttk.Treeview` widget that displays the data.
 - Graph Frame: Holds the `matplotlib` bar graphs.
- The graphs are embedded in the Tkinter window using `FigureCanvasTkAgg` from the `matplotlib` library, which integrates the plots into the GUI.

7. Initialization:

When the application starts, it initializes the combobox with country values, displays the full dataset in the table, and shows the bar graphs for all data. This setup allows users to see the complete dataset before applying any filters.

Summary

This project provides an interactive data analysis and visualization tool that allows users to filter web traffic data based on country and revenue, and view the results in a table and various bar graphs. It uses Tkinter for the GUI, pandas for data manipulation, and matplotlib for visualization, creating a comprehensive tool for analyzing web traffic metrics.

EXAMPLE 2.14

Generating A Synthetic Dataset for Web Traffic

This code generates a synthetic dataset that simulates web traffic metrics and saves it to an Excel file.

```
import pandas as pd
import numpy as np
import random
from datetime import datetime, timedelta

# Define the number of rows
num_rows = 1000

# Generate random dates
start_date = datetime(2024, 1, 1)
date_range = [start_date + timedelta(days=i) for i
in range(num_rows)]
dates = [start_date +
timedelta(days=random.randint(0, 364)) for _ in
range(num_rows)]

# Generate random data for other columns
visits = np.random.randint(1000, 2000,
size=num_rows)
page_views = np.random.randint(2500, 3500,
size=num_rows)
unique_users = np.random.randint(700, 1100,
size=num_rows)
bounce_rate = np.round(np.random.uniform(0.3, 0.5,
size=num_rows), 2)
countries = [random.choice(['USA', 'Canada', 'UK',
'Australia', 'Germany']) for _ in range(num_rows)]
device_types = [random.choice(['Mobile',
'Desktop', 'Tablet']) for _ in range(num_rows)]
revenue = np.random.randint(4000, 8000,
size=num_rows)

# Create DataFrame
data = {
    'Date': dates,
```

```

    'Visits': visits,
    'Page Views': page_views,
    'Unique Users': unique_users,
    'Bounce Rate': bounce_rate,
    'Country': countries,
    'Device Type': device_types,
    'Revenue': revenue
}

df = pd.DataFrame(data)

# Save to Excel
excel_filename = 'web_traffic_data.xlsx'
df.to_excel(excel_filename, index=False)

print(f"DataFrame with {num_rows} rows has been
      saved to {excel_filename}.")

```

Here's a detailed explanation of what each part of the code does:

1. Importing Libraries:

- pandas as pd: Used for creating and manipulating data in DataFrame format.
- numpy as np: Provides numerical operations, especially for generating random numbers.
- random: Used for generating random choices and random numbers within specified ranges.
- datetime and timedelta: Used for handling date and time-related operations.

2. Setting Up the Data Generation:

`num_rows = 1000`: Defines the number of rows to be generated in the dataset, representing 1000 entries of web traffic data.

3. Generating Random Dates:

- `start_date = datetime(2024, 1, 1)`: Sets the start date for the random date generation as January 1, 2024.
- `date_range`: A list comprehension creates a sequence of dates starting from `start_date` and extending over 1000 days.
- `dates`: A list of 1000 randomly selected dates within the year 2024. The `random.randint(0, 364)` function ensures that dates are chosen within a one-year range from the start date.

4. Generating Random Data for Other Columns:

- `visits`: An array of 1000 random integers representing the number of visits, ranging between 1000 and 2000.
- `page_views`: An array of 1000 random integers for page views, ranging between 2500 and 3500.
- `unique_users`: An array of 1000 random integers for unique users, ranging between 700 and 1100.
- `bounce_rate`: An array of 1000 random floating-point numbers rounded to two decimal places, representing bounce rates between 0.3 and 0.5.
- `countries`: A list of 1000 random choices of countries from a predefined list `['USA', 'Canada', 'UK', 'Australia', 'Germany']`.
- `device_types`: A list of 1000 random choices of device types from the list `['Mobile', 'Desktop', 'Tablet']`.
- `revenue`: An array of 1000 random integers for revenue, ranging between 4000 and 8000.

5. Creating the DataFrame:

- `data`: A dictionary where each key corresponds to a column name and each value is the generated data for that column.

- `df = pd.DataFrame(data)`: Converts the dictionary into a pandas DataFrame, where each key in the dictionary becomes a column in the DataFrame.

6. Saving the DataFrame to Excel:

- `excel_filename = 'web_traffic_data.xlsx'`: Specifies the filename for the Excel file to be saved.
- `df.to_excel(excel_filename, index=False)`: Exports the DataFrame to an Excel file named `web_traffic_data.xlsx`. The `index=False` argument ensures that the row indices are not included in the Excel file.

7. Output Message:

```
print(f"DataFrame with {num_rows} rows has been saved to {excel_filename}.")
```

Prints a confirmation message that the DataFrame has been successfully saved to the specified Excel file.

Summary

This script generates a synthetic dataset containing 1000 rows of web traffic data, including columns for dates, visits, page views, unique users, bounce rate, country, device type, and revenue. The data is generated randomly within specified ranges and choices, and then saved to an Excel file named `web_traffic_data.xlsx`. This can be useful for testing data analysis tools, developing applications, or simulating web traffic scenarios without using real data.

EXAMPLE 2.15

GUI Tkinter for Analyzing Synthetic Dataset for Web Traffic

This code creates a Tkinter-based GUI application that reads web traffic data from an Excel file, allows users to filter this data by country and minimum revenue, and displays the results in both a table and a series of bar graphs.

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

# Read data from Excel
df = pd.read_excel('web_traffic_data.xlsx')

# Convert 'Date' to datetime format and extract
month
df['Date'] = pd.to_datetime(df['Date'])
df['Month'] = df['Date'].dt.to_period('M')

# Aggregate data by month
monthly_data = df.groupby('Month').agg({
    'Visits': 'sum',
    'Page Views': 'sum',
    'Unique Users': 'sum',
    'Revenue': 'sum'
}).reset_index()

# Create tkinter window
root = tk.Tk()
root.title("Web Traffic Data Analysis")

# Create a frame for user input
input_frame = tk.Frame(root)
input_frame.pack(pady=10)
```

```
# Filter inputs
country_label = tk.Label(input_frame,
text="Country:")
country_label.grid(row=0, column=0, padx=5,
pady=5)
country_entry = ttk.Combobox(input_frame)
country_entry.grid(row=0, column=1, padx=5,
pady=5)

revenue_label = tk.Label(input_frame, text="Min
Revenue:")
revenue_label.grid(row=1, column=0, padx=5,
pady=5)
revenue_entry = tk.Entry(input_frame)
revenue_entry.grid(row=1, column=1, padx=5,
pady=5)
revenue_entry.insert(0, "5000")

# Function to update combobox with unique country
values
def update_combobox():
    # Get unique countries from the DataFrame
    unique_countries = df['Country'].unique()

        # Ensure the list is clean and formatted as
    strings
    unique_countries = [str(country) for country
in unique_countries]

        # Update the Combobox with the cleaned list of
    countries
    country_entry['values'] = unique_countries

        # Set a default value if available
    if unique_countries:
```

```
country_entry.set(unique_countries[0])

# Function to filter DataFrame and update table
and graphs
def filter_data():
    country_filter = country_entry.get()
    revenue_filter = int(revenue_entry.get())

    filtered_df = df[(df['Country'] ==
country_filter) & (df['Revenue'] >
revenue_filter)]

    # Convert filtered dates to months and
aggregate
    filtered_df['Month'] =
filtered_df['Date'].dt.to_period('M')
    filtered_monthly_data =
filtered_df.groupby('Month').agg({
        'Visits': 'sum',
        'Page Views': 'sum',
        'Unique Users': 'sum',
        'Revenue': 'sum'
    }).reset_index()

    # Update Treeview with filtered data
    update_treeview(filtered_df)
    # Update bar graphs with filtered data
    update_bar_graphs(filtered_monthly_data)

# Function to refresh DataFrame and reset table
and graphs
def refresh_data():
    update_combobox() # Update combobox with
current unique countries
    update_treeview(df)
    update_bar_graphs(monthly_data)
```

```
# Create a frame for displaying the table and
graphs side by side
display_frame = tk.Frame(root)
display_frame.pack(padx=10, pady=10, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the table on the
left side
table_frame = tk.Frame(display_frame)
table_frame.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the graphs on the
right side
graph_frame = tk.Frame(display_frame)
graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH,
expand=True)

# Define columns and create Treeview
columns = list(df.columns)
tree = ttk.Treeview(table_frame, columns=columns,
show='headings')
for col in columns:
    tree.heading(col, text=col)
    tree.column(col, width=100, anchor='center')
tree.pack(fill=tk.BOTH, expand=True)

# Insert data into Treeview
def update_treeview(data):
    # Clear existing rows
    for row in tree.get_children():
        tree.delete(row)

    # Insert new rows with alternating row colors
    for index, row in data.iterrows():
```

```

        color = '#f0f0f0' if index % 2 == 0 else
'#ffffff'
        tree.insert('', 'end', values=list(row),
tags=('row%d' % index,))
        tree.tag_configure('row%d' % index,
background=color)

# Create separate bar graphs for Visits, Page
Views, Unique Users, and Revenue
def update_bar_graphs(data):
    fig, axs = plt.subplots(2, 2, figsize=(10, 6),
dpi=100) # 2x2 grid of subplots
    fig.tight_layout(pad=5.0) # Adjust spacing
between subplots

    # Plot Visits
    axs[0, 0].bar(data['Month'].astype(str),
data['Visits'], color='blue', label='Visits')
    axs[0, 0].set_title('Visits')
    axs[0, 0].set_xlabel('Month')
    axs[0, 0].set_ylabel('Visits')
    axs[0, 0].tick_params(axis='x', rotation=45,
labelsize=8) # Set font size of x-ticks
    axs[0, 0].tick_params(axis='y', labelsize=8)
    # Set font size of y-ticks
    for index, value in enumerate(data['Visits']):
        axs[0, 0].text(index, value + 50,
str(value), ha='center', va='bottom', fontsize=8)

    # Plot Page Views
    axs[0, 1].bar(data['Month'].astype(str),
data['Page Views'], color='green', label='Page
Views')
    axs[0, 1].set_title('Page Views')
    axs[0, 1].set_xlabel('Month')
    axs[0, 1].set_ylabel('Page Views')

```

```
    axs[0, 1].tick_params(axis='x', rotation=45,
labelsize=8) # Set font size of x-ticks
    axs[0, 1].tick_params(axis='y', labelsize=8)
# Set font size of y-ticks
    for index, value in enumerate(data['Page
Views']):
        axs[0, 1].text(index, value + 50,
str(value), ha='center', va='bottom', fontsize=8)

    # Plot Unique Users
    axs[1, 0].bar(data['Month'].astype(str),
data['Unique Users'], color='orange',
label='Unique Users')
    axs[1, 0].set_title('Unique Users')
    axs[1, 0].set_xlabel('Month')
    axs[1, 0].set_ylabel('Unique Users')
    axs[1, 0].tick_params(axis='x', rotation=45,
labelsize=8) # Set font size of x-ticks
    axs[1, 0].tick_params(axis='y', labelsize=8)
# Set font size of y-ticks
    for index, value in enumerate(data['Unique
Users']):
        axs[1, 0].text(index, value + 50,
str(value), ha='center', va='bottom', fontsize=8)

    # Plot Revenue
    axs[1, 1].bar(data['Month'].astype(str),
data['Revenue'], color='red', label='Revenue')
    axs[1, 1].set_title('Revenue')
    axs[1, 1].set_xlabel('Month')
    axs[1, 1].set_ylabel('Revenue')
    axs[1, 1].tick_params(axis='x', rotation=45,
labelsize=8) # Set font size of x-ticks
    axs[1, 1].tick_params(axis='y', labelsize=8)
# Set font size of y-ticks
```

```

    for index, value in
enumerate(data['Revenue']):
        axs[1, 1].text(index, value + 50,
str(value), ha='center', va='bottom', fontsize=8)

    # Clear previous canvas content
    for widget in graph_frame.winfo_children():
        widget.destroy()

    # Embed bar graphs into Tkinter GUI
    canvas = FigureCanvasTkAgg(fig,
master=graph_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(side=tk.TOP,
fill=tk.BOTH, expand=True)

# Filter and Refresh Buttons
filter_button = tk.Button(input_frame,
text="Filter", command=filter_data)
filter_button.grid(row=2, columnspan=2, pady=5)

refresh_button = tk.Button(input_frame,
text="Refresh", command=refresh_data)
refresh_button.grid(row=3, columnspan=2, pady=5)

# Initialize Combobox, Treeview, and Graphs with
original data
update_combobox() # Initialize combobox with data
update_treeview(df)
update_bar_graphs(monthly_data)

root.mainloop()

```

Here's a detailed explanation of the code:

1. Importing Required Libraries:

- tkinter as tk and ttk: Used for creating the GUI, including widgets like buttons, labels, and comboboxes.
- pandas as pd: Used for data manipulation and analysis, especially for reading the Excel file and processing the DataFrame.
- matplotlib.pyplot as plt and FigureCanvasTkAgg: Used for creating bar graphs and embedding them in the Tkinter application.

2. Reading and Processing the Data:

- df = pd.read_excel('web_traffic_data.xlsx'): Reads the data from an Excel file into a pandas DataFrame.
- Convert 'Date' to datetime format and extract month:
 - df['Date'] = pd.to_datetime(df['Date']): Converts the 'Date' column to a datetime format.
 - df['Month'] = df['Date'].dt.to_period('M'): Extracts the month from the date and creates a new 'Month' column.
- Aggregate data by month: Groups the data by 'Month' and calculates the sum of 'Visits', 'Page Views', 'Unique Users', and 'Revenue' for each month. This aggregated data is stored in monthly_data.

3. Creating the Tkinter Window:

- root = tk.Tk() and root.title("Web Traffic Data Analysis"): Creates the main application window with the title "Web Traffic Data Analysis".
- input_frame = tk.Frame(root): Creates a frame within the main window for user input (country selection and minimum revenue filter).

4. Adding User Input Widgets:

- Country Filter:
 - A label and a combobox (`ttk.Combobox`) are created for the user to select a country.
 - The combobox will be populated with the unique countries found in the dataset.
- Revenue Filter:
 - A label and an entry widget (`tk.Entry`) are created for the user to input the minimum revenue filter.
 - The entry widget is pre-populated with a default value of "5000".

5. Functions to Handle Filtering and Updates:

- `update_combobox`:
 - This function populates the country combobox with unique country values from the DataFrame.
- `filter_data`:
 - Filters the DataFrame based on the selected country and minimum revenue.
 - Updates the table and bar graphs with the filtered data.
- `refresh_data`:
 - Resets the application by refreshing the combobox, table, and graphs to display the original, unfiltered data.

6. Creating Frames for Displaying Data and Graphs:

- `display_frame`:

A frame that holds both the table and the graphs, placed side by side.

- `table_frame`:

A frame inside `display_frame` to display the table on the left side.

- graph_frame:

A frame inside display_frame to display the graphs on the right side.

7. Displaying the Data in a Table (Treeview):

- tree = ttk.Treeview(table_frame, columns=columns, show='headings'):
 - Creates a Treeview widget to display the data in a table format.
 - The columns of the DataFrame are used as headings, and each column is given a width and centered alignment.
- update_treeview(data):
 - This function updates the Treeview with data, clearing any previous rows and inserting new ones.
 - Rows are alternately colored for better readability.

8. Creating and Displaying Bar Graphs:

- update_bar_graphs(data):
 - This function creates a 2x2 grid of bar graphs for 'Visits', 'Page Views', 'Unique Users', and 'Revenue' using the filtered data.
 - The graphs are then embedded into the Tkinter GUI using FigureCanvasTkAgg.

9. Adding Buttons for Filtering and Refreshing:

- filter_button and refresh_button:

Buttons are created to allow users to apply the filter or reset the data to its original state.

10. Initializing the GUI with Original Data:

- update_combobox(): Populates the country combobox when the application starts.

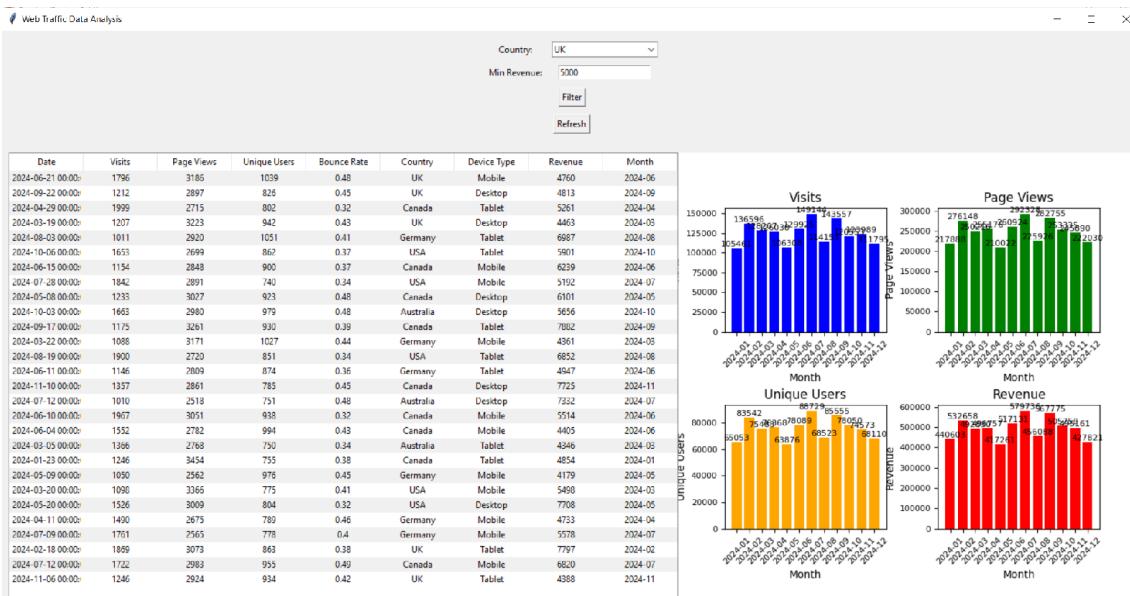
- `update_treeview(df)` and
`update_bar_graphs(monthly_data)`: Displays the original unfiltered data in the table and graphs.

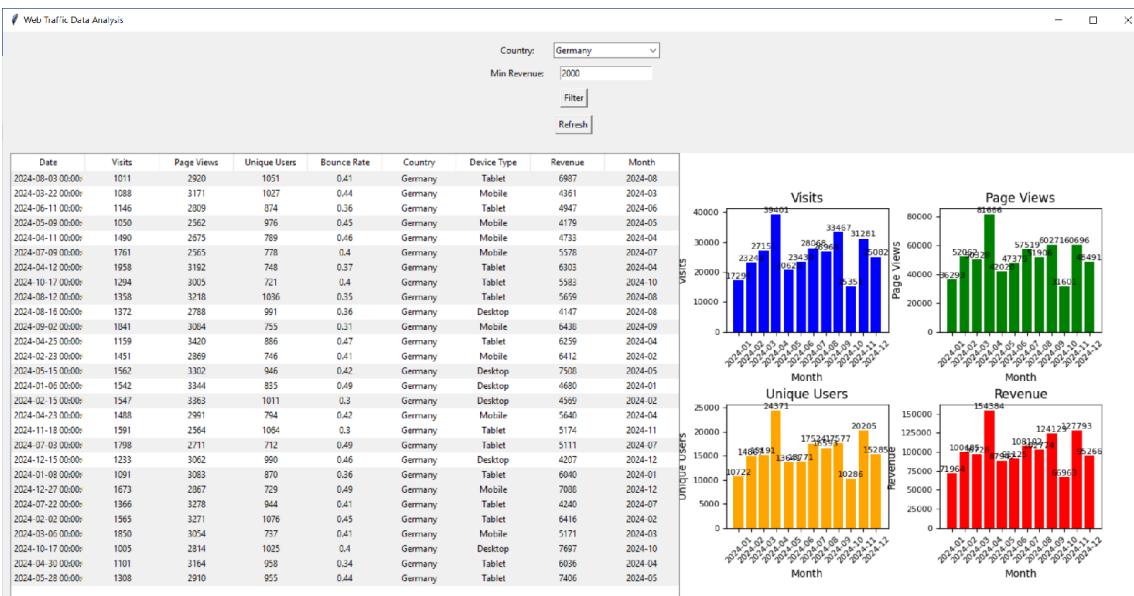
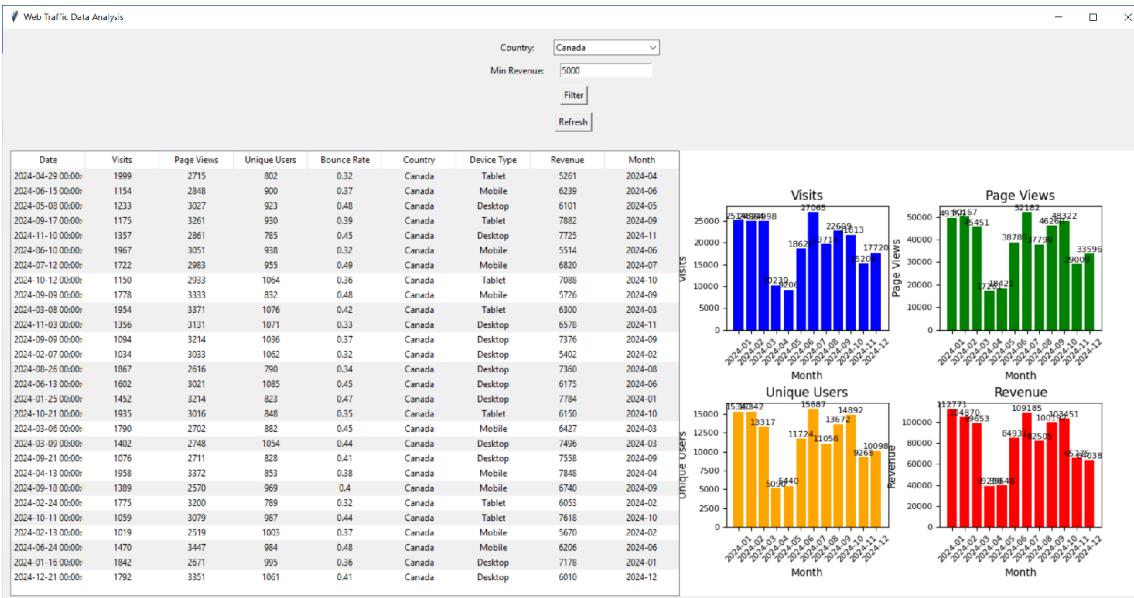
11. Running the Tkinter Main Loop:

`root.mainloop()`: Starts the Tkinter event loop, which waits for user interactions and updates the GUI accordingly.

Summary

This script creates a Tkinter-based GUI that allows users to analyze web traffic data. Users can filter the data by country and minimum revenue, and the filtered results are displayed in both a table and a series of bar graphs. The application also allows users to reset the filters and view the original data. This is useful for visualizing and interacting with data in a user-friendly way.





USING FUNCTION

In pandas, you can filter DataFrames based on more complex or dynamic criteria by using custom functions. This approach is particularly useful when the filtering logic is too complex to be easily expressed with simple comparisons or when you need to apply multiple conditions that may vary dynamically.

Here's how this can be done using the `apply()` method in combination with a lambda function or a separately defined custom function:

Using `apply()` with a Lambda Function:

The `apply()` method allows you to apply a function along an axis (rows or columns) of the DataFrame. When combined with a lambda function, you can filter the DataFrame based on custom logic.

```
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'Country': ['USA', 'Canada', 'UK',
'Australia']
}
df = pd.DataFrame(data)

# Filter rows where Age is greater than 30
filtered_df = df[df.apply(lambda row: row['Age'] > 30, axis=1)]

print(filtered_df)
```

Explanation:

- Lambda Function: The lambda function here is `lambda row: row['Age'] > 30`, which checks if the 'Age' in each row is greater than 30.
- `apply()` Method: `apply()` applies this lambda function to each row (since `axis=1`), and returns a boolean Series indicating whether each row meets the condition.
- Filtering: The DataFrame is then filtered using this boolean Series to retain only the rows where the condition is True.

Using a Separately Defined Custom Function:

You can also define a custom function outside of `apply()` for more complex filtering logic.

```
def custom_filter(row):
    # Complex logic: filter rows where Age is
    # greater than 30 and Country is 'UK'
    return row['Age'] > 30 and row['Country'] ==
    'UK'

# Apply the custom function to filter the
# DataFrame
filtered_df = df[df.apply(custom_filter, axis=1)]

print(filtered_df)
```

Explanation:

- Custom Function: The `custom_filter` function implements more complex logic, checking both the 'Age' and 'Country' columns.
- `apply()` Method: Again, `apply()` is used to apply this function to each row of the DataFrame.

- Filtering: The DataFrame is filtered based on the result of the custom function, keeping only the rows that satisfy all conditions.

Why Use This Approach?

- Flexibility: Allows for complex and dynamic filtering criteria that can't be easily expressed with basic operations.
- Reusability: Custom functions can be reused across different filtering operations, making your code more modular and easier to maintain.
- Clarity: Separating the filtering logic into its own function can make your code more readable and easier to understand.

This method is powerful when dealing with complex datasets that require intricate filtering logic.

EXAMPLE 2.16

Filtering World Food Data

This script creates a synthetic dataset of world food data, saves it to an Excel file, filters the data based on specific criteria, and optionally saves the filtered data to a new Excel file. The use of `apply()` with a custom function is a powerful way to filter DataFrames based on complex criteria.

```
import pandas as pd
import numpy as np
import random

# Step 1: Generate Synthetic World Food Data
```

```
# Define the number of rows
num_rows = 1000

# Generate random data
countries = [random.choice(['USA', 'China',
'India', 'Brazil', 'Germany', 'Australia',
'Canada', 'Russia', 'France', 'UK']) for _ in
range(num_rows)]
food_types = [random.choice(['Wheat', 'Rice',
'Corn', 'Soybean', 'Barley', 'Sugar', 'Potato',
'Tomato', 'Apple', 'Banana']) for _ in
range(num_rows)]
production = np.random.randint(50000, 5000000,
size=num_rows) # in tons
consumption = np.random.randint(40000, 4500000,
size=num_rows) # in tons
export = np.random.randint(10000, 2000000,
size=num_rows) # in tons

# Create DataFrame
data = {
    'Country': countries,
    'Food Type': food_types,
    'Production (tons)': production,
    'Consumption (tons)': consumption,
    'Export (tons)': export
}

df = pd.DataFrame(data)

# Step 2: Save the Data in Excel
excel_filename = 'world_food_data.xlsx'
df.to_excel(excel_filename, index=False)

print(f"Synthetic world food data with {num_rows} rows has been saved to {excel_filename}.")
```

```
# Step 3: Filter the DataFrame Using apply() with
# a Custom Function

# Define the custom function
def filter_food_data(row):
    # Criteria: Production must exceed consumption
    # and exports must be above 500,000 tons
    return row['Production (tons)'] >
    row['Consumption (tons)'] and row['Export (tons)']
    > 500000

# Apply the custom function to filter the
# DataFrame
filtered_df = df[df.apply(filter_food_data,
axis=1)]

print("Filtered DataFrame:")
print(filtered_df)

# Optionally, save the filtered DataFrame to a new
# Excel file
filtered_excel_filename =
'filtered_world_food_data.xlsx'
filtered_df.to_excel(filtered_excel_filename,
index=False)

print(f"Filtered data has been saved to
{filtered_excel_filename}.")
```

Here's a detailed breakdown of each part of the code:

1. Generate Synthetic Data:

- Define the Number of Rows:

Set the number of rows (`num_rows`) for the dataset. In this case, it's 1000 rows.

- Generate Random Data:

- Countries: Create a list of country names by randomly selecting from a predefined list of countries. Each entry in the list corresponds to one row in the dataset.
- Food Types: Generate a list of food types by randomly choosing from a predefined list of food items. Each entry in the list corresponds to one row in the dataset.
- Production: Use a random integer generator to create production values between 50,000 and 5,000,000 tons for each row.
- Consumption: Similarly, generate consumption values between 40,000 and 4,500,000 tons.
- Export: Create export values between 10,000 and 2,000,000 tons.

- Create DataFrame:

Combine the lists of countries, food types, production, consumption, and export values into a structured DataFrame using pandas. This DataFrame organizes the data into columns with meaningful labels.

2. Save Data to Excel:

- Define the File Name:

Specify the name of the Excel file (`world_food_data.xlsx`) where the DataFrame will be saved.

- Save the DataFrame:

Use pandas' `to_excel` method to write the DataFrame to the specified Excel file. This method saves the data in a format that can be easily opened and analyzed in spreadsheet software.

3. Filter Data:

- Define a Custom Filtering Function:

Create a function (`filter_food_data`) that takes a row of data as input and returns True if:

- The production value in the row exceeds the consumption value.
- The export value in the row is greater than 500,000 tons.

- Apply the Filtering Function:

Use pandas' `apply` method to apply the custom function to each row of the DataFrame. This method evaluates the function for each row and returns a boolean mask indicating which rows meet the criteria.

- Filter the DataFrame:

Use the boolean mask to filter the original DataFrame and create a new DataFrame (`filtered_df`) that only includes rows where the custom function returned True.

4. Optional: Save Filtered Data:

- Define the Filtered File Name:

Specify the name of the new Excel file (`filtered_world_food_data.xlsx`) where the filtered DataFrame will be saved.

- Save the Filtered DataFrame:

Write the filtered DataFrame to the new Excel file using pandas' `to_excel` method. This file will contain only the rows that passed the filtering criteria and can be used for further analysis.

EXAMPLE 2.17

GUI Tkinter for Filtering World Food Data

The code provides a user-friendly interface for analyzing and visualizing food data, enabling users to explore different aspects of the dataset through interactive filters and visual representations. This can help in making data-driven decisions or gaining insights into global food production and trade patterns.

```
import pandas as pd
import numpy as np
import random

# Generate Synthetic World Food Data
num_rows = 1000
countries = [random.choice(['USA', 'China',
'India', 'Brazil', 'Germany', 'Australia',
'Canada', 'Russia', 'France', 'UK']) for _ in
range(num_rows)]
food_types = [random.choice(['Wheat', 'Rice',
'Corn', 'Soybean', 'Barley', 'Sugar', 'Potato',
'Tomato', 'Apple', 'Banana']) for _ in
range(num_rows)]
production = np.random.randint(50000, 5000000,
size=num_rows) # in tons
consumption = np.random.randint(40000, 4500000,
size=num_rows) # in tons
export = np.random.randint(10000, 2000000,
size=num_rows) # in tons

# Create DataFrame
data = {
    'Country': countries,
    'Food Type': food_types,
```

```
'Production (tons)': production,
'Consumption (tons)': consumption,
'Export (tons)': export
}

df = pd.DataFrame(data)

# Save DataFrame to Excel
df.to_excel('world_food_data.xlsx', index=False)

import tkinter as tk
from tkinter import ttk
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Read data from Excel
df = pd.read_excel('world_food_data.xlsx')

# Example of using apply() to create a new column
based on a condition
# Adding a new column to classify food type by
production scale
df['Production Scale'] = df['Production
(tons)'].apply(lambda x: 'High' if x > 1500000
else 'Low')

# Create tkinter window
root = tk.Tk()
root.title("World Food Data Analysis")

# Create a frame for user input
input_frame = tk.Frame(root)
input_frame.pack(pady=10)
```

```
# Filter inputs
country_label = tk.Label(input_frame,
text="Country:")
country_label.grid(row=0, column=0, padx=5,
pady=5)
country_entry = ttk.Combobox(input_frame)
country_entry.grid(row=0, column=1, padx=5,
pady=5)

production_label = tk.Label(input_frame, text="Min
Export (tons):")
production_label.grid(row=1, column=0, padx=5,
pady=5)
production_entry = tk.Entry(input_frame)
production_entry.grid(row=1, column=1, padx=5,
pady=5)
production_entry.insert(0, "50000")

# Function to update combobox with unique country
values
def update_combobox():
    unique_countries = df['Country'].unique()
    unique_countries = [str(country) for country
in unique_countries]
    country_entry['values'] = unique_countries
    if unique_countries:
        country_entry.set(unique_countries[0])

# Function to determine if a row meets the filter
criteria
def filter_criteria(row):
    country_filter = country_entry.get()
    production_filter =
int(production_entry.get())
```

```
    return (row['Country'] == country_filter) and
    (row['Export (tons)'] > production_filter)

# Function to filter DataFrame and update table
# and graphs
def filter_data():
    # Apply the filter criteria to each row
    filtered_df = df[df.apply(filter_criteria,
axis=1)]

    update_treeview(filtered_df)
    update_bar_graphs(filtered_df)

# Function to refresh DataFrame and reset table
# and graphs
def refresh_data():
    update_combobox()
    update_treeview(df)
    update_bar_graphs(df)

# Create a frame for displaying the table and
# graphs side by side
display_frame = tk.Frame(root)
display_frame.pack(padx=10, pady=10, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the table on the
# left side
table_frame = tk.Frame(display_frame)
table_frame.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Create a frame for displaying the graphs on the
# right side
graph_frame = tk.Frame(display_frame)
```

```

graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH,
expand=True)

# Define columns and create Treeview
columns = list(df.columns)
tree = ttk.Treeview(table_frame, columns=columns,
show='headings')
for col in columns:
    tree.heading(col, text=col)
    tree.column(col, width=100, anchor='center')
tree.pack(fill=tk.BOTH, expand=True)

# Insert data into Treeview
def update_treeview(data):
    # Clear existing rows
    for row in tree.get_children():
        tree.delete(row)

    # Insert new rows with alternating row colors
    # based on their position in the filtered data
    for index, (i, row) in enumerate(data.iterrows()):
        color = '#f0f0f0' if index % 2 == 0 else
'#ffffff'
        tree.insert('', 'end', values=list(row),
tags=('row%d' % index,))
        tree.tag_configure('row%d' % index,
background=color)

def add_sum_on_bars(ax, bars, sums):
    """ Add the sum label at the top of each bar.
"""
    for bar, food_type_sum in zip(bars, sums):
        height = bar.get_height()
        ax.text(bar.get_x() + bar.get_width() / 2,
height, str(int(food_type_sum)), ha='center',

```

```

va='bottom', fontsize=8)

def add_values_on_bars(ax, bars, values):
    """ Add the sum label at the top of each bar.
"""
    for bar, value in zip(bars, values):
        height = bar.get_height()
        ax.text(bar.get_x() + bar.get_width() / 2,
height, str(int(value)), ha='center', va='bottom',
        fontsize=8)

def update_bar_graphs(data):
    fig, axs = plt.subplots(2, 2, figsize=(10, 6),
dpi=100)
    fig.tight_layout(pad=5.0)

    # Aggregate data by 'Food Type'
    production_sum = data.groupby('Food Type')
    ['Production (tons)'].sum()
    consumption_sum = data.groupby('Food Type')
    ['Consumption (tons)'].sum()
    export_sum = data.groupby('Food Type')['Export
(tons)'].sum()

    # Plot Production
    bars = axs[0, 0].bar(production_sum.index,
    production_sum.values, color='blue',
    label='Production')
    axs[0, 0].set_title('Production (tons)')
    axs[0, 0].tick_params(axis='x', rotation=45,
    labelsize=8)
    axs[0, 0].tick_params(axis='y', labelsize=8)
    add_values_on_bars(axs[0, 0], bars,
    production_sum.values)

    # Plot Consumption

```

```

        bars = axs[0, 1].bar(consumption_sum.index,
consumption_sum.values, color='green',
label='Consumption')
        axs[0, 1].set_title('Consumption (tons)')
        axs[0, 1].tick_params(axis='x', rotation=45,
labelsize=8)
        axs[0, 1].tick_params(axis='y', labelsize=8)
        add_values_on_bars(axs[0, 1], bars,
consumption_sum.values)

# Plot Export
bars = axs[1, 0].bar(export_sum.index,
export_sum.values, color='orange', label='Export')
axs[1, 0].set_title('Export (tons)')
axs[1, 0].tick_params(axis='x', rotation=45,
labelsize=8)
axs[1, 0].tick_params(axis='y', labelsize=8)
add_values_on_bars(axs[1, 0], bars,
export_sum.values)

# Plot Production Scale
production_scale_counts = data['Production
Scale'].value_counts()
bars = axs[1,
1].bar(production_scale_counts.index,
production_scale_counts.values, color='purple',
label='Production Scale')
axs[1, 1].set_title('Production Scale (High vs
Low)')
axs[1, 1].tick_params(axis='x', rotation=0,
labelsize=8)
axs[1, 1].tick_params(axis='y', labelsize=8)
add_values_on_bars(axs[1, 1], bars,
production_scale_counts.values)

for widget in graph_frame.winfo_children():

```

```

        widget.destroy()

        canvas = FigureCanvasTkAgg(fig,
master=graph_frame)
        canvas.draw()
        canvas.get_tk_widget().pack(side=tk.TOP,
fill=tk.BOTH, expand=True)

# Filter and Refresh Buttons
filter_button = tk.Button(input_frame,
text="Filter", command=filter_data)
filter_button.grid(row=2, column=0, padx=5,
pady=5, sticky='ew')

refresh_button = tk.Button(input_frame,
text="Refresh", command=refresh_data)
refresh_button.grid(row=2, column=1, padx=5,
pady=5, sticky='ew')

# Adjust the grid column weights to ensure buttons
expand properly
input_frame.grid_columnconfigure(0, weight=1)
input_frame.grid_columnconfigure(1, weight=1)

# Initialize Combobox, Treeview, and Graphs with
original data
update_combobox()
update_treeview(df)
update_bar_graphs(df)

root.mainloop()

```

Here's the explanation of how the code works, broken down into its main components and steps:

1. Importing Libraries

- tkinter and ttk: These libraries are used to create the graphical user interface (GUI). tkinter provides basic GUI elements like windows, labels, and buttons, while ttk offers more advanced widgets like comboboxes and treeviews.
- pandas: This library is used for data manipulation and analysis. It helps in reading data from Excel and handling it in a table-like structure called a DataFrame.
- matplotlib: This library is used for creating visualizations such as graphs and charts. The FigureCanvasTkAgg class allows matplotlib figures to be embedded in a Tkinter application.

2. Reading Data

`df = pd.read_excel('world_food_data.xlsx')`: Reads data from an Excel file named world_food_data.xlsx into a DataFrame. This DataFrame will contain information about food production, consumption, and exports.

3. Adding a New Column

`df['Production Scale'] = df['Production (tons)'].apply(lambda x: 'High' if x > 1500000 else 'Low')`: Adds a new column called Production Scale to the DataFrame. This column classifies food production as 'High' or 'Low' based on whether the production amount exceeds 1,500,000 tons.

4. Creating the Main Window

- `root = tk.Tk()`: Initializes the main application window.
- `root.title("World Food Data Analysis")`: Sets the title of the window.

5. Setting Up User Input Area

- Creating an Input Frame: A frame (a container for widgets) is created to hold user input elements like

labels, text boxes, and buttons.

- Country Selector: A label and a combobox (dropdown list) are added for users to select a country. The combobox will be populated with unique country values from the DataFrame.
- Minimum Export Entry: A label and a text entry box are added for users to input the minimum export amount for filtering the data. A default value of 50,000 tons is set.

6. Defining Functions

- `update_combobox()`: Updates the combobox with a list of unique countries from the DataFrame and sets the default selected value.
- `filter_criteria(row)`: Defines the filtering criteria based on user input (selected country and minimum export amount).
- `filter_data()`: Applies the filtering criteria to the DataFrame, updates the table and charts with the filtered data.
- `refresh_data()`: Resets the display to show the original, unfiltered data, updates the combobox, table, and charts.

7. Creating Frames for Display

- Display Frame: A frame is created to hold both the data table and the charts, positioned side by side.
- Table Frame: This frame holds the Treeview widget that will display the data in a tabular format.
- Graph Frame: This frame is used to display the charts.

8. Setting Up the Treeview Widget

- Treeview Configuration: Defines columns for the Treeview based on the DataFrame's columns. Each

column is set up with a heading and a default width.

- `update_treeview(data)`: Updates the Treeview with data. Rows are added with alternating colors for better readability.

9. Creating and Updating Charts

- `add_sum_on_bars(ax, bars, sums)` and `add_values_on_bars(ax, bars, values)`: Helper functions to add labels on top of the bars in the charts.
- `update_bar_graphs(data)`: Creates four bar charts displaying:
 - Total production by food type
 - Total consumption by food type
 - Total export by food type
 - Counts of high vs. low production scale

The charts are embedded in the Tkinter application using FigureCanvasTkAgg.

10. Adding Buttons

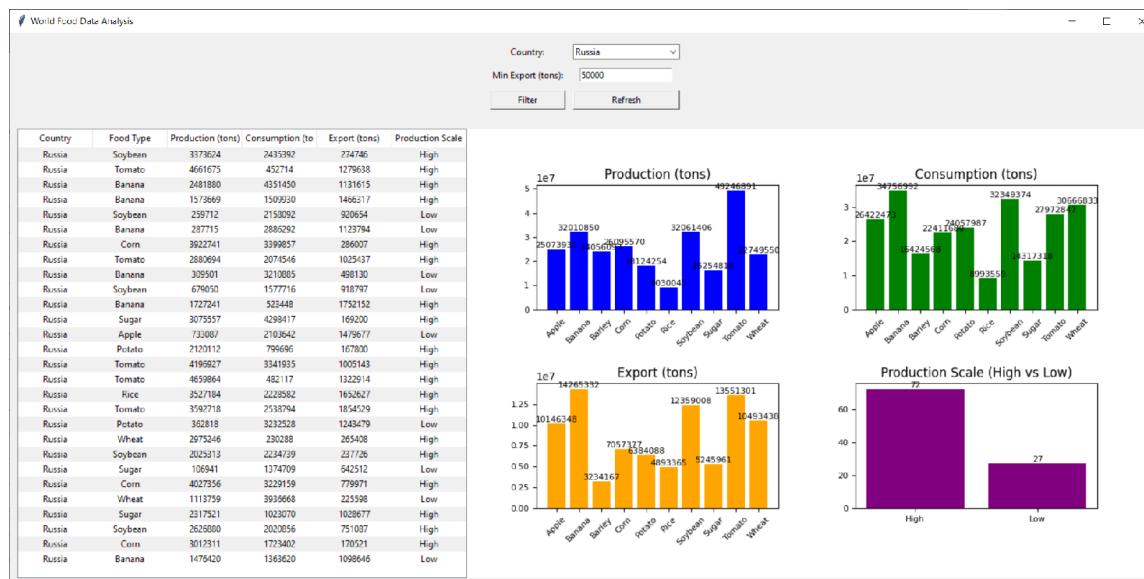
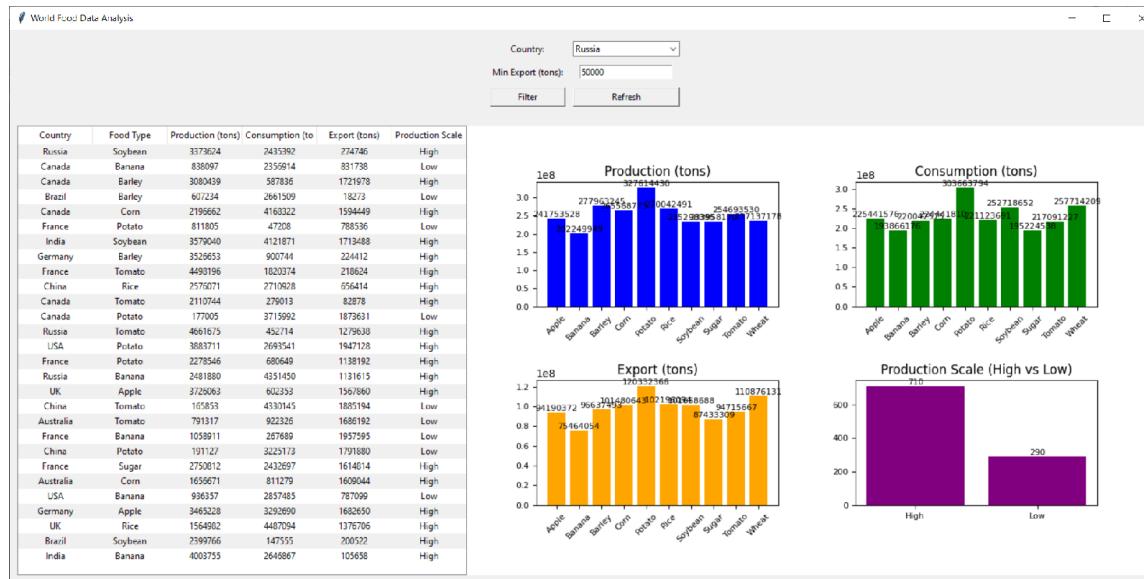
- Filter Button: A button labeled "Filter" triggers the `filter_data()` function to apply the selected filters and update the table and charts.
- Refresh Button: A button labeled "Refresh" triggers the `refresh_data()` function to reset the filters and display the original data.

11. Final Setup

- Grid Configuration: Adjusts the column weights in the input frame to ensure buttons expand properly when the window size changes.
- Initialization: Calls `update_combobox()`, `update_treeview(df)`, and `update_bar_graphs(df)` to populate the initial data in the combobox, table, and charts.

12. Starting the Application

`root.mainloop()`: Starts the Tkinter event loop, which keeps the application running and responsive to user interactions.



COMBINING FILTERING WITH OTHER OPERATIONS

When working with data in a DataFrame, filtering is a powerful operation that can be combined with other data manipulation techniques to perform more complex and insightful analyses. Here's a detailed explanation of how combining filtering with other operations can enhance data analysis:

1. Filtering

Filtering is the process of selecting a subset of data based on specific criteria. For example, you might filter rows in a DataFrame to only include data from a certain country, or where the value in a column exceeds a certain threshold. This allows you to focus on the most relevant data for your analysis.

2. Sorting

Sorting involves ordering your data based on the values in one or more columns. After filtering, you might want to sort the filtered data to identify trends, such as which country has the highest production or which food type is exported the most.

3. Grouping

Grouping is a technique used to split your data into groups based on the values in one or more columns. After filtering the data, you might group it by a specific category, like "Food Type" or "Country," to perform operations on each group separately. This is especially useful for comparing different categories or summarizing data within each group.

4. Aggregation

Aggregation involves applying functions like sum, average, count, etc., to grouped data. After filtering and grouping your data, you can aggregate it to calculate summary statistics. For example, you might want to calculate the total production or

average export for each food type after filtering out certain countries.

Combining These Operations

By combining filtering with sorting, grouping, and aggregation, you can perform sophisticated data analysis tasks. Here are a few examples:

- Example 1: Filtered and Sorted Data: After filtering the data to include only countries with high exports, you might sort this filtered data to identify the top exporters.
- Example 2: Grouped and Aggregated Data: After filtering for a specific food type, you could group the data by country and then aggregate the results to find the total production or average consumption for each country.
- Example 3: Sorted, Grouped, and Aggregated Data: You might filter data for a specific year, group it by food type, aggregate the total production, and then sort the results to identify which food type had the highest production that year.

Benefits

- Focused Analysis: By filtering out irrelevant data, you can concentrate on the specific subset that matters most to your analysis.
- Insights and Trends: Sorting, grouping, and aggregating filtered data help uncover patterns, trends, and outliers that might not be apparent in the raw data.
- Efficiency: Combining these operations allows you to perform complex analyses in a streamlined and efficient manner, reducing the need for manual data manipulation.

In summary, combining filtering with other DataFrame operations like sorting, grouping, and aggregation enhances your ability to perform detailed and sophisticated data analyses, leading to deeper insights and more informed decisions.

EXAMPLE 2.18

Filtering and Aggregating World Food Data

The project aims to provide tools for creating synthetic data, analyzing it, and presenting the results in a meaningful way. It can be used for various purposes, including testing data processing techniques, practicing data analysis skills, or developing data-driven applications.

```
consumption = np.random.randint(40000, 4500000,  
size=num_rows) # in tons  
  
# Create DataFrame  
df = pd.DataFrame({  
    'Country': countries,  
    'Food Type': food_types,  
    'Production (tons)': production,  
    'Consumption (tons)': consumption  
})  
  
# Step 2: Save the DataFrame to an Excel file  
df.to_excel('synthetic_food_data.xlsx',  
index=False)  
  
# Step 3: Filter for a Specific Food Type (e.g.,  
'Wheat')  
filtered_df = df[df['Food Type'] == 'Wheat']  
  
# Step 4: Group by Country and Aggregate the  
Results  
grouped_df = filtered_df.groupby('Country').agg({  
    'Production (tons)': 'sum',  
    'Consumption (tons)': 'mean'  
}).reset_index()  
  
# Step 5: Display the Grouped and Aggregated Data  
print(grouped_df)  
  
# Optionally, save the grouped data to a new Excel  
file  
grouped_df.to_excel('grouped_aggregated_food_data.  
xlsx', index=False)
```

Here's a summary of what the code does, without using the actual code:

1. Imports Libraries: It imports necessary libraries for data handling, numerical operations, and generating random values.
2. Generate Synthetic Data: It creates a dataset with 1000 rows of data, including:
 - Randomly chosen countries from a predefined list.
 - Randomly chosen food types from a predefined list.
 - Random numerical values for production and consumption of these food types, simulating realistic data ranges.
3. Create DataFrame: It organizes this synthetic data into a DataFrame, which is a table-like structure with columns for Country, Food Type, Production (in tons), and Consumption (in tons).
4. Save Data to Excel: It saves the DataFrame to an Excel file, creating a permanent record of the synthetic dataset.
5. Filter Data: It extracts a subset of the data where the food type is 'Wheat', focusing on this specific food type.
6. Group and Aggregate Data: It groups the filtered data by Country and calculates the total production and average consumption for each country.
7. Display Results: It prints the grouped and aggregated data to the console.
8. Optional Save: It can also save the grouped and aggregated data to a new Excel file for further use.

EXAMPLE 2.19

GUI Tkinter for Filtering and Aggregating World Food Data

This code sets up a comprehensive GUI application for managing, analyzing, and visualizing food production and consumption data, providing interactive features like data filtering and graphical representation.

```
import os
import pandas as pd
import numpy as np
import random
import tkinter as tk
from tkinter import ttk
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# File path
file_path = 'synthetic_food_data.xlsx'

# Check if file exists
if os.path.isfile(file_path):
    # Load existing data
    df = pd.read_excel(file_path)
else:
    # Generate Synthetic Dataset
    num_rows = 1000
    countries = [random.choice(['USA', 'China',
'India', 'Brazil', 'Germany', 'Australia',
'Canada', 'Russia', 'France', 'UK']) for _ in
range(num_rows)]
    food_types = [random.choice(['Wheat', 'Rice',
'Corn', 'Soybean', 'Barley', 'Sugar', 'Potato',
'Tomato', 'Apple', 'Banana']) for _ in
range(num_rows)]
    production = np.random.randint(50000, 5000000,
size=num_rows) # in tons
```

```
consumption = np.random.randint(40000,
4500000, size=num_rows) # in tons

# Create DataFrame
df = pd.DataFrame({
    'Country': countries,
    'Food Type': food_types,
    'Production (tons)': production,
    'Consumption (tons)': consumption
})

# Clean up the 'Food Type' column in the
DataFrame
df['Food Type'] = df['Food Type'].str.strip()

# Save to Excel
df.to_excel(file_path, index=False)

# Keep a copy of the original DataFrame
original_df = df.copy()

# Tkinter GUI
class FoodDataGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Food Data GUI")

        # Create a frame for the Listbox,
        Combobox, and Graph
        self.frame = ttk.Frame(root)
        self.frame.pack(side=tk.LEFT,
fill=tk.BOTH, expand=True)

        # Create a frame for the Listbox and
        Combobox
        self.filter_frame = ttk.Frame(self.frame)
```

```
        self.filter_frame.pack(side=tk.TOP,
fill=tk.BOTH, expand=True)

        # Listbox for selecting Country
        self.country_listbox =
tk.Listbox(self.filter_frame,
selectmode=tk.MULTIPLE)
        self.country_listbox.pack(side=tk.LEFT,
fill=tk.BOTH, expand=True)
        self.populate_country_listbox()

        # Combobox for selecting Food Type
        self.food_combobox =
ttk.Combobox(self.filter_frame)
        self.food_combobox.pack(side=tk.LEFT,
fill=tk.X, padx=5, pady=5)
        self.populate_food_combobox()

        # Create a frame for the buttons
        self.button_frame = ttk.Frame(self.frame)
        self.button_frame.pack(side=tk.TOP,
fill=tk.X, padx=5, pady=5)

        # Button to update data
        self.update_button =
ttk.Button(self.button_frame, text="Update Data",
command=self.update_data)
        self.update_button.pack(side=tk.LEFT,
padx=5)

        # Button to refresh data
        self.refresh_button =
ttk.Button(self.button_frame, text="Refresh Data",
command=self.refresh_data)
        self.refresh_button.pack(side=tk.LEFT,
padx=5)
```

```
# Create a frame for the table and scrollbars
    self.table_frame = ttk.Frame(self.frame)
    self.table_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        # Table to display DataFrame
        self.tree = ttk.Treeview(self.table_frame, columns=list(df.columns), show='headings')
        self.tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        # Vertical scrollbar
        self.v_scroll =
ttk.Scrollbar(self.table_frame, orient=tk.VERTICAL, command=self.tree.yview)
        self.v_scroll.pack(side=tk.RIGHT, fill=tk.Y)

        self.tree.configure(yscrollcommand=self.v_scroll.set)

        # Horizontal scrollbar
        self.h_scroll =
ttk.Scrollbar(self.table_frame, orient=tk.HORIZONTAL, command=self.tree.xview)
        self.h_scroll.pack(side=tk.BOTTOM, fill=tk.X)

        self.tree.configure(xscrollcommand=self.h_scroll.set)

    for col in df.columns:
        self.tree.heading(col, text=col)
        self.tree.column(col, width=150)
```

```
# Create a Figure and add a subplot
self.figure = Figure(figsize=(6, 4),
dpi=100)
    self.ax = self.figure.add_subplot(111)
    self.canvas =
FigureCanvasTkAgg(self.figure, master=self.frame)

    self.canvas.get_tk_widget().pack(side=tk.RIGHT,
fill=tk.BOTH, expand=True)
        self.ax.set_title("Production and
Consumption")

# Load initial data
self.refresh_data()

def populate_country_listbox(self):
    countries = df['Country'].unique()
    self.country_listbox.delete(0, tk.END) # Clear previous entries
        for country in countries:
            self.country_listbox.insert(tk.END,
country)

def populate_food_combobox(self):
    food_types = df['Food Type'].unique()
    food_types = [food.strip() for food in
food_types] # Clean up food types
        self.food_combobox['values'] = food_types
        self.food_combobox.set('Wheat') # Default selection

def update_data(self):
    selected_countries =
[self.country_listbox.get(i) for i in
self.country_listbox.curselection()]
```

```
        selected_food_type =
self.food_combobox.get().strip() # Clean up
selected food type

        # Filter data
        filtered_df =
original_df[original_df['Food Type'] ==
selected_food_type]
        if selected_countries:
            filtered_df =
filtered_df[filtered_df['Country'].isin(selected_c
ountries)]

        # Update table
        self.update_table(filtered_df)

        # Group data
        grouped_df =
filtered_df.groupby('Country').agg({
    'Production (tons)': 'sum',
    'Consumption (tons)': 'mean'
}).reset_index()

        # Update graph
        self.plot_graph(grouped_df)

def refresh_data(self):
    # Restore original DataFrame
    self.update_table(original_df)

    # Group data for all countries and food
types
        grouped_df =
original_df.groupby('Country').agg({
    'Production (tons)': 'sum',
    'Consumption (tons)': 'mean'
```

```
}).reset_index()

# Update graph
self.plot_graph(grouped_df)

def update_table(self, data):
    for row in self.tree.get_children():
        self.tree.delete(row)

    for index, (_, row) in enumerate(data.iterrows()):
        # Determine row color based on index
        row_color = '#f0f0f0' if index % 2 == 0 else '#ffffff' # Light gray and white

        # Insert row with color
        self.tree.insert(
            '',
            tk.END,
            values=list(row),
            tags=('oddrow' if index % 2 == 0
else 'evenrow')
        )

    # Define colors for row tags
    self.tree.tag_configure('oddrow',
background='#00f0f0') # Light gray
    self.tree.tag_configure('evenrow',
background='#ffffff') # White

def plot_graph(self, data):
    self.ax.clear()

    if data.empty:
        self.ax.set_title("No data available")
        self.ax.set_xlabel("")
```

```

        self.ax.set_ylabel("")
    else:
        data.plot(kind='bar', x='Country', y=
['Production (tons)', 'Consumption (tons)'],
ax=self.ax)
        # Ensure x-axis labels are horizontal

    self.ax.set_xticklabels(data['Country'],
rotation=0, fontsize=8)
    self.ax.set_title("Production and
Consumption by Country")

    self.canvas.draw()

if __name__ == "__main__":
    root = tk.Tk()
    app = FoodDataGUI(root)
    root.mainloop()

```

Here is a detailed explanation of the provided code, which creates a Tkinter GUI application for managing and analyzing food data:

Libraries and File Path

1. Import Libraries: Various libraries are imported:

- os for file operations.
- pandas (pd) for data manipulation and analysis.
- numpy (np) for numerical operations.
- random for generating random data.
- tkinter (tk) and ttk for creating the GUI.
- matplotlib for plotting graphs.

2. File Path: The path to the Excel file where the data will be stored or read from is set to 'synthetic_food_data.xlsx'.

Data Generation or Loading

1. Check for File Existence: The code checks if the Excel file exists using `os.path.isfile(file_path)`:
 - File Exists: If the file exists, it loads the data into a DataFrame `df` using `pd.read_excel(file_path)`.
 - File Does Not Exist: If the file does not exist, it generates a synthetic dataset.
2. Generate Synthetic Data:
 - `num_rows = 1000` specifies the number of rows in the dataset.
 - Random Data Generation:
 - `countries`: List of 1000 countries selected randomly from a predefined list.
 - `food_types`: List of 1000 food types selected randomly from a predefined list.
 - `production`: List of 1000 random integers between 50,000 and 5,000,000 representing food production in tons.
 - `consumption`: List of 1000 random integers between 40,000 and 4,500,000 representing food consumption in tons.
 - Create DataFrame: A DataFrame `df` is created with columns: Country, Food Type, Production (tons), and Consumption (tons).
3. Clean Data: The Food Type column is stripped of any leading or trailing whitespace.
4. Save Data: The DataFrame is saved to an Excel file '`'synthetic_food_data.xlsx'`'.

GUI Setup and Functionality

1. Keep Original Data: A copy of the original DataFrame is kept for filtering and analysis.
2. Tkinter GUI Class:
 - Class Definition: FoodDataGUI class manages the GUI.
 - Initialization (`__init__` method):
 - Root Window: The main Tkinter window is created with the title "Food Data GUI".
 - Frame Setup: Frames are set up for layout:
 - `self.frame`: Main frame for holding the listbox, combobox, and graph.
 - `self.filter_frame`: Frame for the listbox and combobox.
 - `self.button_frame`: Frame for buttons.
 - `self.table_frame`: Frame for the data table and scrollbars.
 - Listbox: For selecting multiple countries. Populated using `self.populate_country_listbox()`.
 - Combobox: For selecting a food type. Populated using `self.populate_food_combobox()`.
 - Buttons:
 - Update Data: Refreshes the data based on current selections.
 - Refresh Data: Restores and updates the full dataset.
 - Data Table: Uses `ttk.Treeview` to display DataFrame with vertical and horizontal scrollbars.
 - Graph: Uses `matplotlib` to plot data. The figure is displayed using `FigureCanvasTkAgg`.

3. Populate Listbox and Combobox:

- `populate_country_listbox()`: Fills the listbox with unique country names.
- `populate_food_combobox()`: Fills the combobox with unique food types, defaulting to 'Wheat'.

4. Update and Refresh Data:

- `update_data()`: Filters the DataFrame based on user selections and updates the table and graph.
- `refresh_data()`: Restores the original DataFrame and updates the table and graph for all data.

5. Update Table:

`update_table(data)`: Updates the table with the given DataFrame. Alternates row colors for better readability.

6. Plot Graph:

`plot_graph(data)`: Plots a bar graph of production and consumption by country. Ensures x-axis labels are horizontal and set with a fontsize of 8.

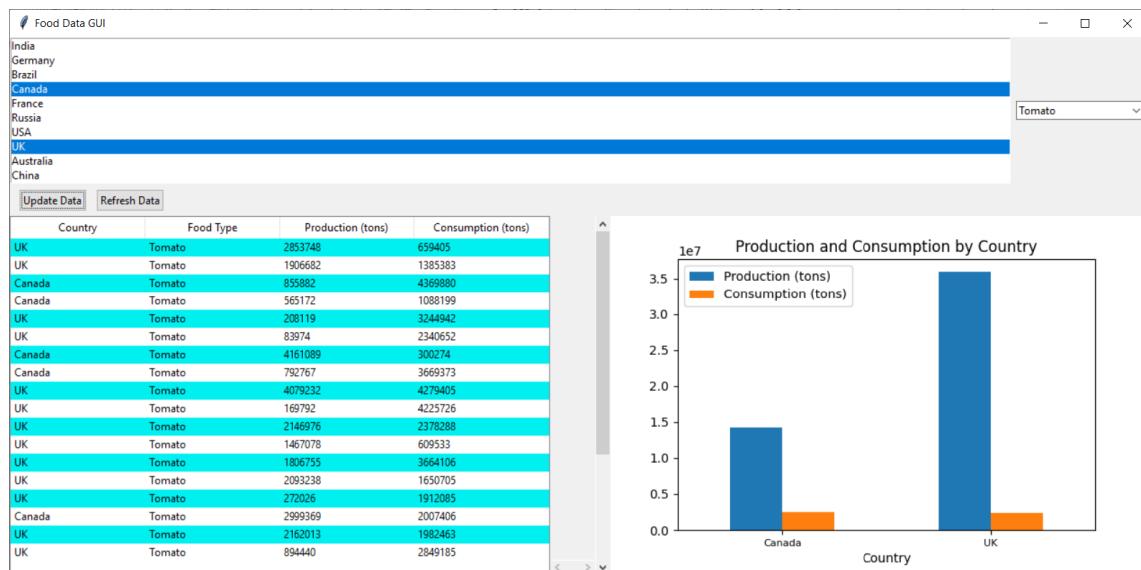
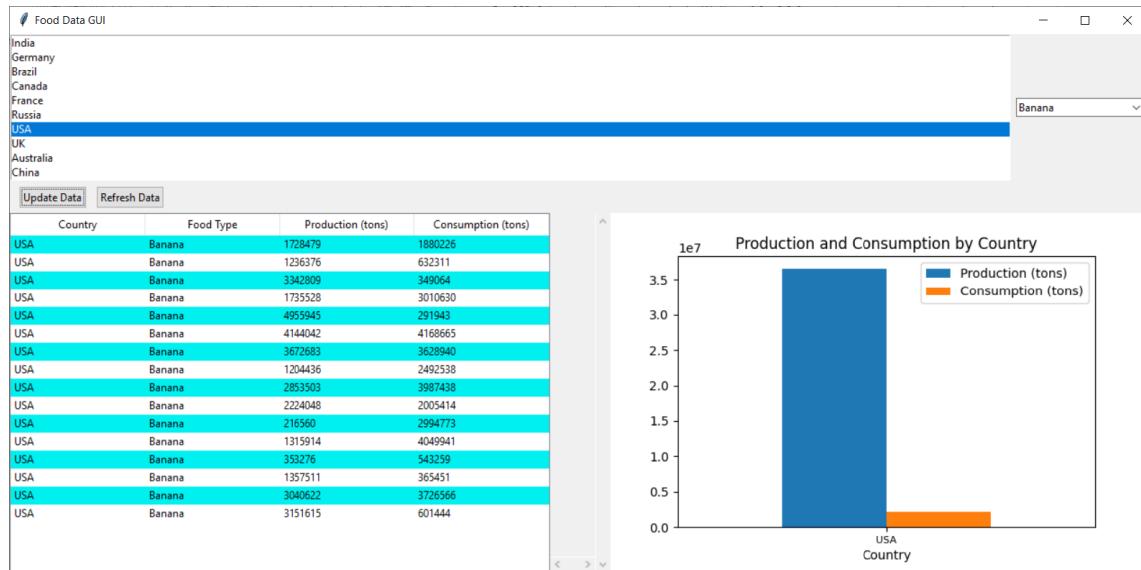


Main Program

Main Execution: Initializes and runs the Tkinter application:

- Creates a root window.

- Instantiates the FoodDataGUI class.
- Starts the Tkinter main event loop.





EXAMPLE 2.20

Filtering and Aggregating Synthetic Forest Data

The purpose of this project is to analyze and manage synthetic forest data for Indonesian provinces.

```
import pandas as pd
import numpy as np
import random

# Step 1: Generate Synthetic Dataset
num_rows = 1000

# Example list of Indonesian provinces
provinces = [random.choice(['West Java', 'Central Java', 'East Java', 'Bali', 'Sumatra', 'Kalimantan', 'Sulawesi', 'Papua']) for _ in range(num_rows)]

# Example list of forest types
forest_types = [random.choice(['Tropical Rainforest', 'Mangrove', 'Monsoon Forest', 'Mountain Forest']) for _ in range(num_rows)]

# Randomly generated data for forest area and tree count
forest_area = np.random.randint(1000, 100000, size=num_rows) # in hectares
tree_count = np.random.randint(10000, 5000000, size=num_rows) # number of trees

# Create DataFrame
df = pd.DataFrame({
    'Province': provinces,
    'Forest Type': forest_types,
    'Forest Area (hectares)': forest_area,
    'Tree Count': tree_count
})
```

```

# Step 2: Save the DataFrame to an Excel file
df.to_excel('indonesian_forest_data.xlsx',
index=False)

# Step 3: Filter for a Specific Forest Type (e.g.,
'Tropical Rainforest')
filtered_df = df[df['Forest Type'] == 'Tropical
Rainforest']

# Step 4: Group by Province and Aggregate the
Results
grouped_df = filtered_df.groupby('Province').agg({
    'Forest Area (hectares)': ['sum', 'min',
'max'],
    'Tree Count': ['mean', 'min', 'max']
}).reset_index()

# Flatten the MultiIndex columns
grouped_df.columns = ['.join(col).strip()' for
col in grouped_df.columns.values]
grouped_df = grouped_df.rename(columns={'Province
': 'Province'})

# Step 5: Display the Grouped and Aggregated Data
print(grouped_df)

# Optionally, save the grouped data to a new Excel
file
grouped_df.to_excel('grouped_aggregated_forest_dat
a.xlsx', index=False)

```

This Python script performs the following tasks:

1. Generate Synthetic Dataset:

- Imports Libraries: The script imports pandas for data manipulation, numpy for numerical operations, and random for random data generation.
- Set Number of Rows: num_rows is set to 1000, indicating the number of rows of data to generate.
- Generate Data:
 - Provinces: Randomly selects a province from a predefined list for each row.
 - Forest Types: Randomly selects a forest type from a predefined list for each row.
 - Forest Area: Generates random values for forest area in hectares between 1,000 and 100,000.
 - Tree Count: Generates random values for tree count between 10,000 and 5,000,000.
- Create DataFrame: Constructs a DataFrame df with columns 'Province', 'Forest Type', 'Forest Area (hectares)', and 'Tree Count' using the generated data.

2. Save the DataFrame to an Excel File:

Save File: Exports the DataFrame to an Excel file named 'indonesian_forest_data.xlsx' without the index column.

3. Filter Data for Specific Forest Type:

Filter Data: Filters the DataFrame to include only rows where the 'Forest Type' is 'Tropical Rainforest'.

4. Group by Province and Aggregate Results:

- Group and Aggregate:
 - Groups the filtered data by 'Province'.
 - Aggregates data for 'Forest Area (hectares)' (sum, min, max) and 'Tree Count' (mean, min, max).
- Flatten MultiIndex Columns: Flattens the MultiIndex columns created by aggregation into

single-level columns and renames the 'Province ' column (with an extra space) to 'Province'.

5. Display and Save Grouped Data:

- Print Grouped Data: Displays the grouped and aggregated DataFrame.
- Save Grouped Data: Optionally saves the grouped data to a new Excel file named 'grouped_aggregated_forest_data.xlsx'.

This script effectively creates a synthetic dataset, processes it by filtering and aggregating, and then saves the results for further analysis.

EXAMPLE 2.21

GUI Tkinter for Filtering and Aggregating Synthetic Forest Data

The script creates a comprehensive GUI application for filtering, aggregating, and visualizing synthetic forest data. It includes functionality for loading and generating data, filtering data based on user input, and displaying results in both a table and graphical format. This setup is useful for analyzing forest data, allowing users to view and interpret various aspects of the data interactively.

```
import os
import pandas as pd
import numpy as np
import random
import tkinter as tk
from tkinter import ttk
import matplotlib.pyplot as plt
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
```

```
# File path
file_path = 'indonesian_forest_data.xlsx'

# Check if file exists
if os.path.isfile(file_path):
    # Load existing data
    df = pd.read_excel(file_path)
else:
    # Generate Synthetic Dataset
    num_rows = 1000
    provinces = [random.choice(['West Java',
    'Central Java', 'East Java', 'Bali', 'Sumatra',
    'Kalimantan', 'Sulawesi', 'Papua']) for _ in
    range(num_rows)]
    forest_types = [random.choice(['Tropical
    Rainforest', 'Mangrove', 'Monsoon Forest',
    'Mountain Forest']) for _ in range(num_rows)]
    forest_area = np.random.randint(1000, 100000,
    size=num_rows) # in hectares
    tree_count = np.random.randint(10000, 5000000,
    size=num_rows) # number of trees

    # Create DataFrame
    df = pd.DataFrame({
        'Province': provinces,
        'Forest Type': forest_types,
        'Forest Area (hectares)': forest_area,
        'Tree Count': tree_count
    })

    # Clean up the 'Forest Type' column in the
    # DataFrame
    df['Forest Type'] = df['Forest
    Type'].str.strip()

    # Save to Excel
```

```
df.to_excel(file_path, index=False)

# Keep a copy of the original DataFrame
original_df = df.copy()

# Tkinter GUI
class ForestDataGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Forest Data GUI")

        # Create a frame for the Listbox,
        Combobox, and Graph
        self.frame = ttk.Frame(root)
        self.frame.pack(side=tk.LEFT,
fill=tk.BOTH, expand=True)

        # Create a frame for the Listbox and
        Combobox
        self.filter_frame = ttk.Frame(self.frame)
        self.filter_frame.pack(side=tk.TOP,
fill=tk.BOTH, expand=True)

        # Listbox for selecting Province
        self.province_listbox =
tk.Listbox(self.filter_frame,
selectmode=tk.MULTIPLE)
        self.province_listbox.pack(side=tk.LEFT,
fill=tk.BOTH, expand=True)
        self.populate_province_listbox()

        # Combobox for selecting Forest Type
        self.forest_combobox =
ttk.Combobox(self.filter_frame)
        self.forest_combobox.pack(side=tk.LEFT,
fill=tk.X, padx=5, pady=5)
```

```
    self.populate_forest_combobox()

    # Create a frame for the buttons
    self.button_frame = ttk.Frame(self.frame)
    self.button_frame.pack(side=tk.TOP,
fill=tk.X, padx=5, pady=5)

        # Button to update data
        self.update_button =
ttk.Button(self.button_frame, text="Update Data",
command=self.update_data)
        self.update_button.pack(side=tk.LEFT,
padx=5)

        # Button to refresh data
        self.refresh_button =
ttk.Button(self.button_frame, text="Refresh Data",
command=self.refresh_data)
        self.refresh_button.pack(side=tk.LEFT,
padx=5)

    # Create a frame for the table and
scrollbars
    self.table_frame = ttk.Frame(self.frame)
    self.table_frame.pack(side=tk.LEFT,
fill=tk.BOTH, expand=True)

    # Table to display DataFrame
    self.tree = ttk.Treeview(self.table_frame,
columns=list(df.columns), show='headings')
    self.tree.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

    # Vertical scrollbar
    self.v_scroll =
ttk.Scrollbar(self.table_frame,
```

```
orient=tk.VERTICAL, command=self.tree.yview)
    self.v_scroll.pack(side=tk.RIGHT,
fill=tk.Y)

    self.tree.configure(yscrollcommand=self.v_scroll.
set)

        # Horizontal scrollbar
        self.h_scroll =
ttk.Scrollbar(self.table_frame,
orient=tk.HORIZONTAL, command=self.tree.xview)
        self.h_scroll.pack(side=tk.BOTTOM,
fill=tk.X)

    self.tree.configure(xscrollcommand=self.h_scroll.
set)

    for col in df.columns:
        self.tree.heading(col, text=col)
        self.tree.column(col, width=150)

        # Create a Figure and add two subplots
        self.figure, (self.ax1, self.ax2) =
plt.subplots(nrows=2, ncols=1, figsize=(10, 12),
dpi=100)
        self.canvas =
FigureCanvasTkAgg(self.figure, master=self.frame)

        self.canvas.get_tk_widget().pack(side=tk.RIGHT,
fill=tk.BOTH, expand=True)

        # Set titles for subplots
        self.ax1.set_title("Forest Area by
Province")
        self.ax2.set_title("Tree Count by
Province")
```

```
# Load initial data
self.refresh_data()

def populate_province_listbox(self):
    provinces = df['Province'].unique()
    self.province_listbox.delete(0, tk.END) # Clear previous entries
    for province in provinces:
        self.province_listbox.insert(tk.END, province)

def populate_forest_combobox(self):
    forest_types = df['Forest Type'].unique()
    forest_types = [forest.strip() for forest
in forest_types] # Clean up forest types
    self.forest_combobox['values'] = forest_types
    self.forest_combobox.set('Tropical Rainforest') # Default selection

def update_data(self):
    selected_provinces =
    [self.province_listbox.get(i) for i in
    self.province_listbox.curselection()]
    selected_forest_type =
    self.forest_combobox.get().strip() # Clean up selected forest type

    # Filter data
    filtered_df =
    original_df[original_df['Forest Type'] ==
    selected_forest_type]
    if selected_provinces:
        filtered_df =
        filtered_df[filtered_df['Province'].isin(selected_
```

```
provinces)]  
  
        # Update table  
        self.update_table(filtered_df)  
  
        # Group data  
        grouped_df =  
filtered_df.groupby('Province').agg({  
            'Forest Area (hectares)': ['median',  
'min', 'max'],  
            'Tree Count': ['mean', 'min', 'max']  
}).reset_index()  
  
        # Flatten MultiIndex columns  
        grouped_df.columns = ['  
.join(col).strip() for col in  
grouped_df.columns.values]  
        grouped_df = grouped_df.rename(columns=  
{'Province ': 'Province'})  
  
        # Update graphs  
        self.plot_graph(grouped_df)  
  
    def refresh_data(self):  
        # Restore original DataFrame  
        self.update_table(original_df)  
  
        # Group data for all provinces and forest  
types  
        grouped_df =  
original_df.groupby('Province').agg({  
            'Forest Area (hectares)': ['median',  
'min', 'max'],  
            'Tree Count': ['mean', 'min', 'max']  
}).reset_index()
```

```

        # Flatten MultiIndex columns
        grouped_df.columns = [
            .join(col).strip() for col in
            grouped_df.columns.values]
        grouped_df = grouped_df.rename(columns=
            {'Province ': 'Province'})

        # Update graphs
        self.plot_graph(grouped_df)

    def update_table(self, data):
        for row in self.tree.get_children():
            self.tree.delete(row)

        for index, (_, row) in
        enumerate(data.iterrows()):
            # Determine row color based on index
            row_color = '#f0f0f0' if index % 2 ==
            0 else '#ffffff' # Light gray and white

            # Insert row with color
            self.tree.insert(
                '',
                tk.END,
                values=list(row),
                tags=( 'oddrow' if index % 2 == 0
            else 'evenrow')
            )

            # Define colors for row tags
            self.tree.tag_configure('oddrow',
background='#f0f0f0') # Light gray
            self.tree.tag_configure('evenrow',
background='#ffffff') # White

    def plot_graph(self, data):

```

```
    self.ax1.clear()
    self.ax2.clear()

    if data.empty:
        self.ax1.set_title("No data
available")
        self.ax1.set_xlabel("")
        self.ax1.set_ylabel("")
        self.ax2.set_title("No data
available")
        self.ax2.set_xlabel("")
        self.ax2.set_ylabel("")
    else:
        # Plot Forest Area
        self.ax1.bar(data['Province'],
data['Forest Area (hectares) median'], color='b',
label='Median Forest Area (hectares)', alpha=0.6)
        self.ax1.plot(data['Province'],
data['Forest Area (hectares) min'], color='r',
marker='o', linestyle='dashed', linewidth=2,
markersize=5, label='Min Forest Area (hectares)')
        self.ax1.plot(data['Province'],
data['Forest Area (hectares) max'], color='g',
marker='o', linestyle='dashed', linewidth=2,
markersize=5, label='Max Forest Area (hectares)')
        self.ax1.set_xlabel('Province')
        self.ax1.set_ylabel('Forest Area
(hectares)')
        self.ax1.set_title('Forest Area by
Province')
        self.ax1.legend()

        self.ax1.set_xticklabels(data['Province'],
rotation=0, ha='right', fontsize=8)

    # Plot Tree Count
```

```

        self.ax2.bar(data['Province'],
data['Tree Count mean'], color='b', label='Average
Tree Count', alpha=0.6)
        self.ax2.plot(data['Province'],
data['Tree Count min'], color='r', marker='o',
linestyle='dashed', linewidth=2, markersize=5,
label='Min Tree Count')
        self.ax2.plot(data['Province'],
data['Tree Count max'], color='y', marker='x',
linestyle='dashed', linewidth=2, markersize=5,
label='Max Tree Count')
        self.ax2.set_xlabel('Province')
        self.ax2.set_ylabel('Tree Count')
        self.ax2.set_title('Tree Count by
Province')
        self.ax2.legend()

self.ax2.set_xticklabels(data['Province'],
rotation=0, ha='right', fontsize=8)

self.canvas.draw()

if __name__ == "__main__":
    root = tk.Tk()
    app = ForestDataGUI(root)
    root.mainloop()

```

Here's a detailed breakdown of each part:

1. Imports and File Path Setup

- Libraries Imported:
 - os: For file path operations.
 - pandas (pd): For data manipulation.
 - numpy (np): For numerical operations.
 - random: For generating synthetic data.
 - tkinter (tk) and ttk: For the GUI.

- matplotlib.pyplot (plt): For plotting graphs.
 - FigureCanvasTkAgg: To embed matplotlib figures in Tkinter.
 - File Path:
 - file_path is set to 'indonesian_forest_data.xlsx'.
2. Check for Existing Data File
- Load Data if Exists: If the file exists, it reads the data into a DataFrame df.
 - Generate Data if Not Exists:
 - Creates a synthetic dataset with 1,000 rows.
 - Generates random values for provinces, forest types, forest areas, and tree counts.
 - Creates and saves a DataFrame df to an Excel file.

3. GUI Setup with Tkinter

- Class Definition: ForestDataGUI initializes the GUI with various widgets and functionalities.
- Constructor (`__init__`):
 - Frames: Sets up frames for layout:
 - `self.frame`: Main frame.
 - `self.filter_frame`: Contains the Listbox and Combobox for filtering.
 - `self.button_frame`: Contains buttons for updating and refreshing data.
 - `self.table_frame`: Contains the table for displaying data.
 - Listbox (`self.province_listbox`): For selecting provinces (supports multiple selections).
 - Combobox (`self.forest_combobox`): For selecting forest types.
 - Buttons:

- Update Data: Filters data based on selections and updates the table and graphs.
- Refresh Data: Restores the original data and updates the table and graphs.
- Table (self.tree): Displays the DataFrame using ttk.Treeview.
- Scrollbars: Vertical and horizontal scrollbars for the table.
- Graphs:
 - Creates a Figure with two subplots using matplotlib.
 - Adds the FigureCanvasTkAgg to display the plots in the Tkinter window.
- Initial Data Loading: Calls self.refresh_data() to load and display initial data.

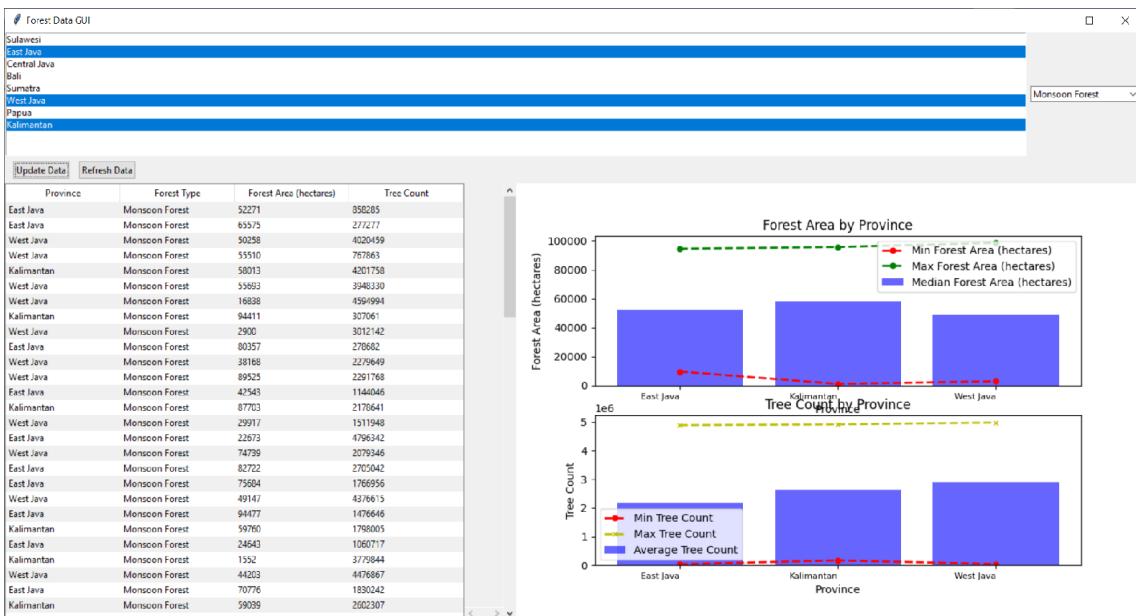
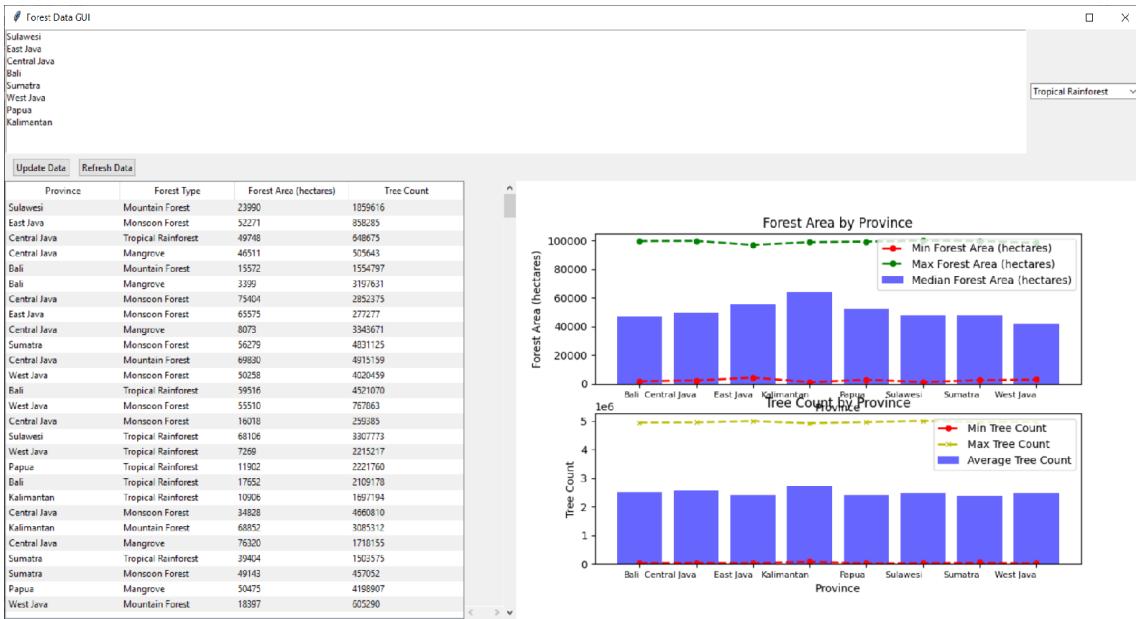
4. GUI Methods

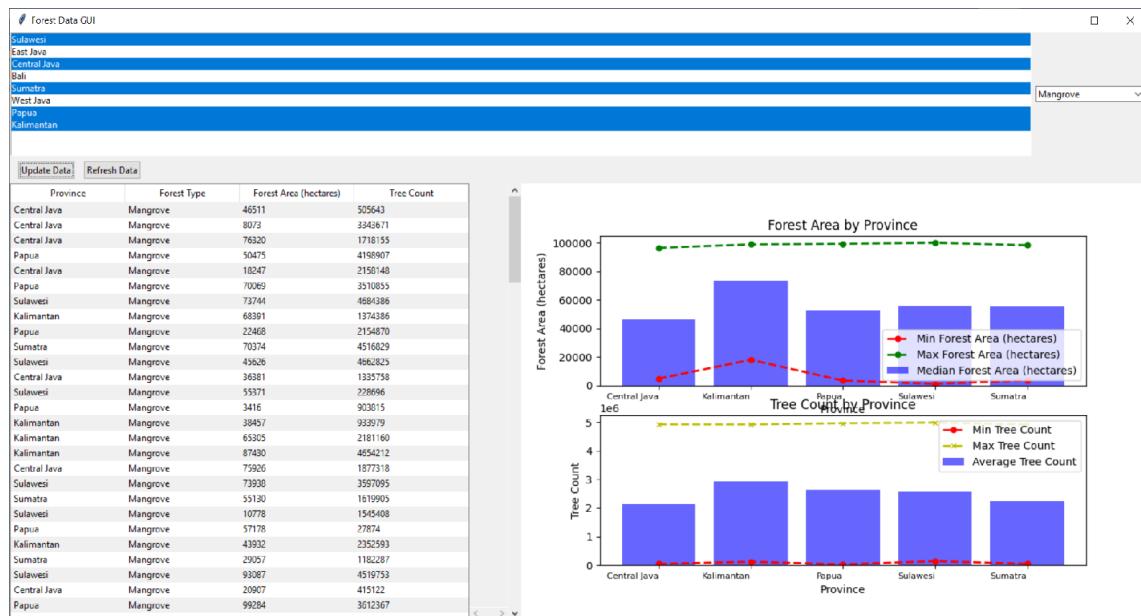
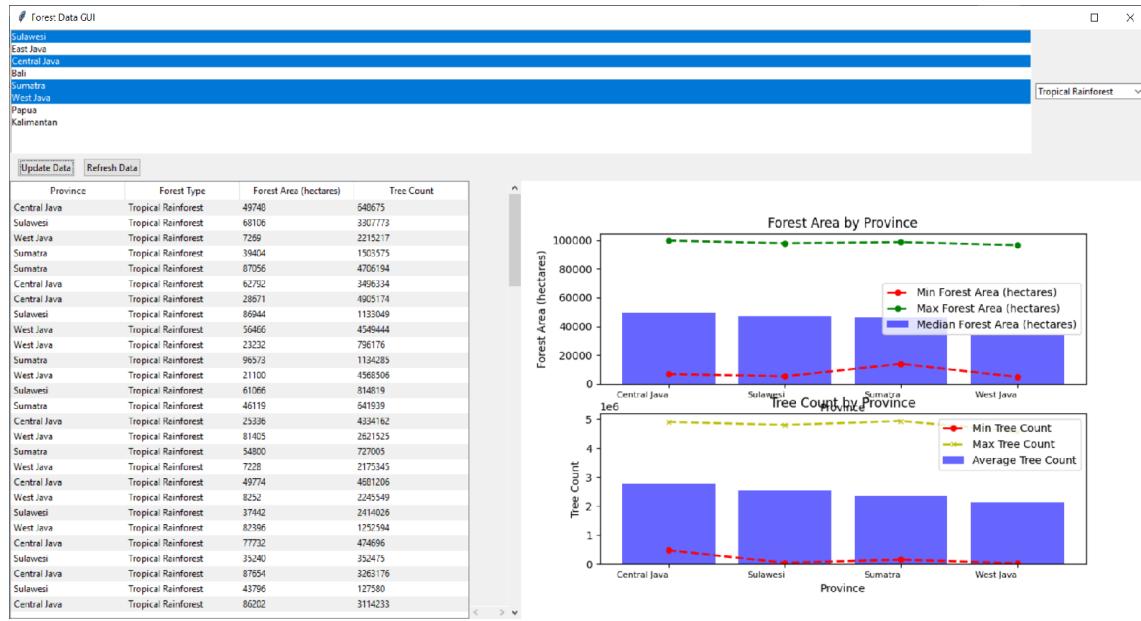
- populate_province_listbox(): Populates the Listbox with unique provinces from the DataFrame.
- populate_forest_combobox(): Populates the Combobox with unique forest types.
- update_data():
 - Filters the DataFrame based on selected provinces and forest types.
 - Updates the table and graphs with the filtered data.
- refresh_data():
 - Restores the original DataFrame.
 - Updates the table and graphs with data grouped by province.
- update_table(data):
 - Clears and updates the table with the given data.
 - Alternates row colors for better readability.

- `plot_graph(data):`
 - Clears and updates the plots based on the data.
 - Plots forest area and tree count with bars and lines showing median, min, and max values.
 - Adjusts x-tick labels for readability and updates the canvas.

5. Main Execution Block

- `__main__:`
 - Initializes the Tkinter root window.
 - Creates an instance of ForestDataGUI.
 - Starts the Tkinter event loop with `root.mainloop()`.





DATAFRAME

SORTING, JOINING, AND MERGING DATAFRAME

SORTING, JOINING, AND MERGING

SORTING

In pandas, a DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It is similar to a spreadsheet or SQL table. You can think of it as a collection of Series objects sharing the same index.

Creating a DataFrame

You can create a DataFrame from various sources, such as dictionaries, lists, or external files (CSV, Excel, etc.). For example:

```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
```

```
'Age': [25, 30, 35],  
'City': ['New York', 'Los Angeles', 'Chicago']  
}  
df = pd.DataFrame(data)
```

Sorting a DataFrame

Sorting a DataFrame in pandas can be done using the `sort_values()` method. This method sorts the DataFrame by the values of one or more columns. You can also sort by index using `sort_index()`.

1. Sorting by Column Values

Syntax:

```
df.sort_values(by='column_name',      ascending=True,  
inplace=False)
```

- `by`: Column or list of columns to sort by.
- `ascending`: Boolean or list of booleans indicating whether to sort in ascending order. Default is True.
- `inplace`: Boolean indicating whether to modify the DataFrame in place. Default is False, which returns a new DataFrame.

Example:

```
# Sorting by Age in ascending order  
df_sorted          =      df.sort_values(by='Age',  
ascending=True)  
  
# Sorting by Age in descending order  
df_sorted_desc     =      df.sort_values(by='Age',  
ascending=False)
```

Sorting by Multiple Columns:

```
# Sorting by City first and then by Age within each City
df_sorted_multiple = df.sort_values(by=['City', 'Age'], ascending=[True, False])
```

2. Sorting by Index

Syntax:

```
df.sort_index(axis=0, ascending=True,
inplace=False)
```

- axis: 0 for rows (default) and 1 for columns.
- ascending: Boolean indicating whether to sort in ascending order. Default is True.

Example:

```
# Sorting by row index
df_sorted_index = df.sort_index(ascending=True)
```

Additional Sorting Features

- na_position: This parameter allows you to specify where NaN values should appear in the sorted result. Options are 'first' or 'last'. Default is 'last'.

```
# Sorting with NaN values placed first
df_sorted_na = df.sort_values(by='Age',
na_position='first')
```

- sort_values with by as list: You can sort by multiple columns. The order of sorting is determined by the order

in the list.

```
df_sorted_multiple      =      df.sort_values(by=['City', 'Age'], ascending=[True, False])
```

- `inplace` parameter: If `True`, sorts the DataFrame in place without returning a new DataFrame.

```
df.sort_values(by='Age',           ascending=True,  
inplace=True)
```

Summary

- DataFrame: A tabular data structure in pandas with labeled axes.
- Sorting by Column Values: Use `sort_values()` to sort by one or more columns.
- Sorting by Index: Use `sort_index()` to sort by the DataFrame's index.
- Additional Features: Control for handling `Nan` values and sorting by multiple columns.
- Sorting helps in organizing and analyzing data more effectively, enabling easier access to the most relevant information.

EXAMPLE 3.1

Sorting Synthetic Financial Data

This code is useful for analyzing and organizing financial data, helping to identify top-performing businesses based on revenue and providing a convenient format for reporting and analysis.

Let's break down this code step-by-step:

Imports and Setup

```
import pandas as pd
import numpy as np
import random

# Set a random seed for reproducibility
random.seed(0)
np.random.seed(0)
```

1. Imports:

- pandas as pd: Imports the pandas library, which is used for data manipulation and analysis.
- numpy as np: Imports numpy, which provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions.
- random: Imports Python's built-in random module to generate random numbers.

2. Setting Random Seed:

`random.seed(0)` and `np.random.seed(0)`: These commands set the seed for random number generation. Setting a seed ensures that the random numbers generated are reproducible, which means running the code multiple times will produce the same random values.

Generating Synthetic Data

```
# Generate synthetic data
num_rows = 10
```

```
businesses = [f'Business {chr(65+i)}' for i in range(num_rows)] # Business names like Business A, Business B, etc.
revenue = np.random.randint(100000, 1000000, size=num_rows) # Monthly revenue in USD
expenses = np.random.randint(50000, 500000, size=num_rows) # Monthly expenses in USD
```

1. Number of Rows:

num_rows = 10: Specifies the number of rows of data to generate.

2. Business Names:

businesses = [f'Business {chr(65+i)}' for i in range(num_rows)]: Generates a list of business names using a list comprehension. chr(65+i) converts the integer values 65, 66, 67, ..., to their corresponding ASCII characters ('A', 'B', 'C', etc.). So, it creates names like Business A, Business B, etc.

3. Monthly Revenue:

revenue = np.random.randint(100000, 1000000, size=num_rows): Generates an array of random integers between 100,000 and 1,000,000. This represents the monthly revenue in USD for each business.

4. Monthly Expenses:

expenses = np.random.randint(50000, 500000, size=num_rows): Generates an array of random integers between 50,000 and 500,000. This represents the monthly expenses in USD for each business.

Creating DataFrame

```
# Create DataFrame
df = pd.DataFrame({
    'Business': businesses,
    'Monthly Revenue (USD)': revenue,
    'Monthly Expenses (USD)': expenses
})

print("Original DataFrame:")
print(df)
```

1. DataFrame Creation:

`df = pd.DataFrame(...)`: Creates a pandas DataFrame using the generated lists and arrays. The DataFrame df will have three columns: 'Business', 'Monthly Revenue (USD)', and 'Monthly Expenses (USD)'.

2. Print DataFrame:

- `print("Original DataFrame:")`: Prints a header for the DataFrame output.
- `print(df)`: Displays the contents of the DataFrame.

Sorting DataFrame

```
# Sort by Monthly Revenue in descending order
df_sorted_by_revenue = df.sort_values(by='Monthly
Revenue (USD)', ascending=False)

print("\nSorted by Monthly Revenue (Descending):")
print(df_sorted_by_revenue)
```

1. Sorting:

`df.sort_values(by='Monthly Revenue (USD)', ascending=False)`: Sorts the DataFrame by the 'Monthly Revenue (USD)' column in descending order (from highest to

lowest revenue). The sorted DataFrame is assigned to df_sorted_by_revenue.

2. Print Sorted DataFrame:

- print("\nSorted by Monthly Revenue (Descending):"): Prints a header for the sorted DataFrame output.
- print(df_sorted_by_revenue): Displays the sorted DataFrame.

Saving to Excel

```
# Save the sorted DataFrame to an Excel file
excel_file_path = 'sorted_monetary_data.xlsx'
df_sorted_by_revenue.to_excel(excel_file_path,
index=False)

print(f"\nSorted DataFrame saved to {excel_file_path}")
```

1. Save to Excel:

df_sorted_by_revenue.to_excel(excel_file_path, index=False): Saves the sorted DataFrame to an Excel file named 'sorted_monetary_data.xlsx'. The index=False parameter ensures that the DataFrame's index is not written to the file.

2. Print Confirmation:

print(f"\nSorted DataFrame saved to {excel_file_path}"): Prints a confirmation message indicating the file path where the DataFrame has been saved.

Summary

- Synthetic Data Generation: Creates a dataset with business names, revenue, and expenses.
- DataFrame Creation: Organizes the data into a pandas DataFrame.
- Sorting: Orders the DataFrame based on monthly revenue in descending order.
- Saving to Excel: Exports the sorted DataFrame to an Excel file for further analysis or sharing.

EXAMPLE 3.2

GUI Tkinter for Sorting Synthetic Financial Data

The purpose of the code is to create a graphical user interface (GUI) that allows users to view and interact with financial data, specifically monthly revenue and expenses for different businesses.

```
import os
import pandas as pd
import numpy as np
import random
import tkinter as tk
from tkinter import ttk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
```

```
import matplotlib.colors as mcolors

# Set a random seed for reproducibility
random.seed(0)
np.random.seed(0)

# Generate synthetic data
num_rows = 20
businesses = [f'Business {chr(65+i)}' for i in
range(num_rows)]
revenue = np.random.randint(100000, 1000000,
size=num_rows)
expenses = np.random.randint(50000, 500000,
size=num_rows)

# Create DataFrame
df = pd.DataFrame({
    'Business': businesses,
    'Monthly Revenue (USD)': revenue,
    'Monthly Expenses (USD)': expenses
})

class MonetaryDataGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Monetary Data GUI")

        # Create a Notebook (tabs)
        self.notebook = ttk.Notebook(root)
        self.notebook.pack(fill=tk.BOTH,
expand=True)

        # Create frames for each tab
        self.revenue_frame =
ttk.Frame(self.notebook)
```

```
        self.expenses_frame =
ttk.Frame(self.notebook)

        # Add frames as tabs
        self.notebook.add(self.revenue_frame,
text="Sort by Revenue")
        self.notebook.add(self.expenses_frame,
text="Sort by Expenses")

        # Create widgets for the "Sort by Revenue"
tab
        self.create_revenue_tab_widgets()

        # Create widgets for the "Sort by
Expenses" tab
        self.create_expenses_tab_widgets()

        # Load initial data
        self.refresh_data()

def create_revenue_tab_widgets(self):
    # Table to display DataFrame
    self.tree_revenue =
ttk.Treeview(self.revenue_frame,
columns=list(df.columns), show='headings')
    self.tree_revenue.pack(side=tk.LEFT,
fill=tk.BOTH, expand=True)

    # Vertical scrollbar
    self.v_scroll_revenue =
ttk.Scrollbar(self.revenue_frame,
orient=tk.VERTICAL,
command=self.tree_revenue.yview)
    self.v_scroll_revenue.pack(side=tk.RIGHT,
fill=tk.Y)
```

```
    self.tree_revenue.configure(yscrollcommand=self.v_
_scroll_revenue.set)

        # Horizontal scrollbar
        self.h_scroll_revenue =
ttk.Scrollbar(self.revenue_frame,
orient=tk.HORIZONTAL,
command=self.tree_revenue.xview)
        self.h_scroll_revenue.pack(side=tk.BOTTOM,
fill=tk.X)

    self.tree_revenue.configure(xscrollcommand=self.h_
_scroll_revenue.set)

    for col in df.columns:
        self.tree_revenue.heading(col,
text=col)
        self.tree_revenue.column(col,
width=150)

        # Create a Figure and add one subplot for
revenue
        self.figure_revenue, self.ax_revenue =
plt.subplots(figsize=(8, 6), dpi=100)
        self.canvas_revenue =
FigureCanvasTkAgg(self.figure_revenue,
master=self.revenue_frame)

        self.canvas_revenue.get_tk_widget().pack(side=tk.
RIGHT, fill=tk.BOTH, expand=True)

def create_expenses_tab_widgets(self):
    # Table to display DataFrame
    self.tree_expenses =
ttk.Treeview(self.expenses_frame,
```

```
columns=list(df.columns), show='headings')
        self.tree_expenses.pack(side=tk.LEFT,
fill=tk.BOTH, expand=True)

        # Vertical scrollbar
        self.v_scroll_expenses =
ttk.Scrollbar(self.expenses_frame,
orient=tk.VERTICAL,
command=self.tree_expenses.yview)
        self.v_scroll_expenses.pack(side=tk.RIGHT,
fill=tk.Y)

        self.tree_expenses.configure(yscrollcommand=self.
v_scroll_expenses.set)

        # Horizontal scrollbar
        self.h_scroll_expenses =
ttk.Scrollbar(self.expenses_frame,
orient=tk.HORIZONTAL,
command=self.tree_expenses.xview)

        self.h_scroll_expenses.pack(side=tk.BOTTOM,
fill=tk.X)

        self.tree_expenses.configure(xscrollcommand=self.
h_scroll_expenses.set)

        for col in df.columns:
            self.tree_expenses.heading(col,
text=col)
            self.tree_expenses.column(col,
width=150)

        # Create a Figure and add one subplot for
expenses
```

```
        self.figure_expenses, self.ax_expenses =
plt.subplots(figsize=(8, 6), dpi=100)
        self.canvas_expenses =
FigureCanvasTkAgg(self.figure_expenses,
master=self.expenses_frame)

self.canvas_expenses.get_tk_widget().pack(side=tk
.RIGHT, fill=tk.BOTH, expand=True)

def refresh_data(self):
    # Initialize or refresh tables and graphs
for both tabs
    self.update_table(self.tree_revenue,
df.sort_values(by='Monthly Revenue (USD)',
ascending=False))
    self.update_table(self.tree_expenses,
df.sort_values(by='Monthly Expenses (USD)',
ascending=False))

    # Plot data for revenue
    self.plot_graph(self.ax_revenue,
df.sort_values(by='Monthly Revenue (USD)',
ascending=False), 'Monthly Revenue (USD)',
'Revenue')

    # Plot data for expenses
    self.plot_graph(self.ax_expenses,
df.sort_values(by='Monthly Expenses (USD)',
ascending=False), 'Monthly Expenses (USD)',
'Expenses')

def update_table(self, tree, data):
    for row in tree.get_children():
        tree.delete(row)
```

```
        for index, (_, row) in
enumerate(data.iterrows()):
    row_color = '#f0f0f0' if index % 2 ==
0 else '#ffffff'

    tree.insert(
        '',
        tk.END,
        values=list(row),
        tags=( 'oddrow' if index % 2 == 0
else 'evenrow')
    )

    tree.tag_configure('oddrow',
background='#f0f0f0')
    tree.tag_configure('evenrow',
background='#ffffff')

    def plot_graph(self, ax, data, value_col,
title):
        ax.clear()

        if data.empty:
            ax.set_title("No data available")
            ax.set_xlabel("")
            ax.set_ylabel("")
        else:
            # Generate a list of colors for the
bars
            colors =
list(mcolors.TABLEAU_COLORS.values())
            num_colors = len(colors)
            bar_colors = [colors[i % num_colors]
for i in range(len(data))]
```

```

        bars = ax.bar(data['Business'],
data[value_col], color=bar_colors, label=title,
alpha=0.8)
        ax.set_xlabel('Business')
        ax.set_ylabel(title)
        ax.set_title(f'{title} by Business')
        ax.legend()
        ax.set_xticklabels(data['Business'],
rotation=45, ha='right', fontsize=8)

# Add y-values on top of each bar
for bar in bars:
    yval = bar.get_height()
    ax.text(
        bar.get_x() + bar.get_width() /
2, yval,
        f'{yval:.0f}', # Format as
integer
        ha='center', va='bottom',
        fontsize=8
    )

    self.canvas_revenue.draw() if ax ==
self.ax_revenue else self.canvas_expenses.draw()

if __name__ == "__main__":
    root = tk.Tk()
    app = MonetaryDataGUI(root)
    root.mainloop()

```

Here's a detailed explanation of the provided code:

1. Imports

The script imports several libraries:

- os: For interacting with the operating system.

- pandas and numpy: For data manipulation and numerical operations.
- random: For generating random numbers.
- tkinter and ttk: For creating the GUI.
- matplotlib.pyplot: For plotting graphs.
- matplotlib.backends.backend_tkagg.FigureCanvasTkAgg: To embed matplotlib plots in the Tkinter GUI.
- matplotlib.colors: For accessing predefined colors in matplotlib.

2. Data Generation

The script generates synthetic data for demonstration purposes:

- A list of 20 business names is created, each labeled "Business A" to "Business T".
- Random revenue and expenses values are generated for each business using numpy.
- The data is stored in a pandas DataFrame df with columns for "Business", "Monthly Revenue (USD)", and "Monthly Expenses (USD)".

3. MonetaryDataGUI Class

This class is the main component of the script and handles the GUI creation and functionality.

- __init__ Method
 - root: The root window of the Tkinter application.
 - A notebook widget (ttk.Notebook) is created to manage multiple tabs within the GUI.
 - Two frames (ttk.Frame) are created: one for "Sort by Revenue" and another for "Sort by

Expenses".

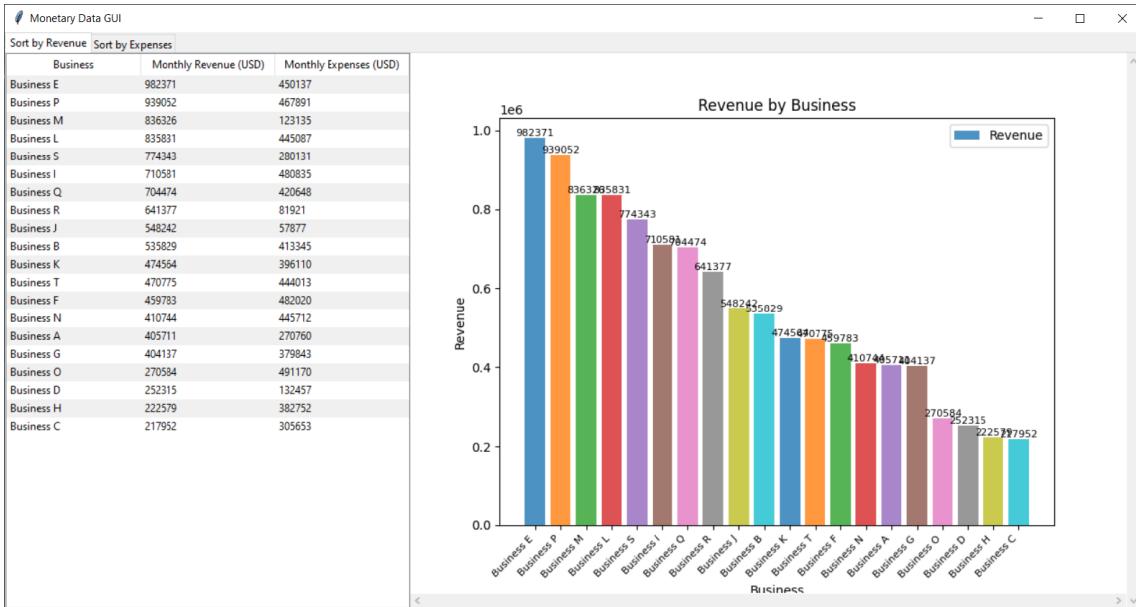
- Each frame is added as a tab to the notebook.
- The method `create_revenue_tab_widgets()` and `create_expenses_tab_widgets()` are called to populate each tab with widgets.
- Finally, `refresh_data()` is called to load and display the initial data.
- `create_revenue_tab_widgets()` Method
 - TreeView Table: A table is created using `ttk.Treeview` to display the DataFrame. This widget supports columns and headings.
 - Scrollbars: Vertical and horizontal scrollbars are added to the TreeView to handle overflow.
 - Plot: A `matplotlib` figure is created with one subplot for the revenue data. The plot is embedded into the Tkinter GUI using `FigureCanvasTkAgg`.
- `create_expenses_tab_widgets()` Method
 - This method is nearly identical to `create_revenue_tab_widgets()`, but it handles the "Sort by Expenses" tab instead.
- `refresh_data()` Method
 - `update_table()`: This method is called to refresh the data in the TreeView tables for both the revenue and expenses tabs.
 - `plot_graph()`: This method is called to plot the revenue and expenses data in their respective graphs.
- `update_table()` Method
 - Clears existing data in the TreeView.
 - Iterates over the DataFrame rows and inserts each row into the TreeView. Alternate rows are

colored differently (light gray and white) for readability.

- `plot_graph()` Method
 - Clears any existing plot on the provided `ax`.
 - If the `DataFrame` is empty, it displays a "No data available" message.
 - Otherwise, it generates a bar plot with the business names on the x-axis and the chosen value column (either revenue or expenses) on the y-axis.
 - Bar Colors: Each bar in the plot is assigned a color from `TABLEAU_COLORS`, ensuring a visually distinct representation.
 - Y-Values: The y-values (i.e., the heights of the bars) are displayed on top of each bar for clarity. This is done using `ax.text()`, which places the value at the center and slightly above the top of each bar.

4. Main Program Execution

- If the script is run directly (as opposed to being imported as a module), a Tkinter root window is created.
- An instance of `MonetaryDataGUI` is created, passing the root window as an argument.
- The Tkinter main loop (`root.mainloop()`) starts, allowing the GUI to function interactively.



Summary of Key Features

- Data Display: The GUI provides an interactive table to view the revenue and expenses data.
- Data Visualization: The GUI also includes bar plots to visualize the monthly revenue and expenses for each business.
- Interactivity: Although no buttons are present, the data is dynamically updated and displayed in both the tables and plots, responding to any changes made within the GUI.



EXAMPLE 3.3

Sorting Synthetic Unemployment Data by Index

This script generates synthetic unemployment data for 10 different regions across 5 years. It first sets up random seeds to ensure that the random data generated will be the same every time the script is run. The regions are labeled alphabetically (e.g., Region A, Region B), and the years range from 2020 to 2024.

The script creates random unemployment rates for each region in each year, then stores this data in a table. After organizing the data alphabetically by region, the script saves the table to an Excel file. Finally, it prints a confirmation message indicating that the data has been successfully saved.

```

import pandas as pd
import numpy as np
import random

# Set a random seed for reproducibility
    
```

```
random.seed(0)
np.random.seed(0)

# Generate synthetic data
num_regions = 10
regions = [f'Region {chr(65+i)}' for i in
range(num_regions)] # Region names like Region A,
Region B, etc.
years = [f'Year {2020+i}' for i in range(5)] # Years from 2020 to 2024

# Generate random unemployment data
unemployment_data = {year: np.random.uniform(3.0,
15.0, size=num_regions) for year in years}

# Create DataFrame
df_unemployment = pd.DataFrame(unemployment_data,
index=regions)

# Sort the DataFrame by index (regions)
df_unemployment_sorted =
df_unemployment.sort_index()

# Save to Excel
file_path = 'synthetic_unemployment_data.xlsx'
df_unemployment_sorted.to_excel(file_path,
index=True)

print(f"Synthetic unemployment data saved to {file_path}.")
```

Here's a step-by-step explanation of the code:

1. Importing Required Libraries:

The script begins by importing three essential Python libraries: pandas, numpy, and random.

- pandas is used for handling and manipulating data in tables (DataFrames).
- numpy is used for numerical operations, especially for generating random numbers.
- random is used to control randomness in the script.

2. Setting Random Seeds:

Random seeds are set using both the random and numpy libraries. This step ensures that the random numbers generated by the script are the same each time it runs, which is crucial for reproducibility. Setting seeds helps in debugging and comparing results.

3. Generating Region Names and Years:

- The script defines the number of regions (10 in this case) and creates a list of region names like "Region A", "Region B", etc., using a combination of chr() and a loop.
- It also generates a list of years from 2020 to 2024.

4. Generating Synthetic Unemployment Data:

For each year in the list, the script generates a list of random unemployment rates for each region. These rates are uniformly distributed between 3.0% and 15.0%, creating realistic but random data. The data is stored in a dictionary, where each key is a year, and the corresponding value is a list of unemployment rates for that year.

5. Creating a DataFrame:

The script then uses the pandas library to create a DataFrame from the dictionary. The rows represent different regions, and the columns represent different years. This DataFrame is a tabular representation of the synthetic unemployment data.

6. Sorting the DataFrame by Region:

The script sorts the DataFrame alphabetically by the region names (which are the index of the DataFrame). Sorting helps

in organizing the data and makes it easier to interpret.

7. Saving the Data to an Excel File:

The sorted DataFrame is then saved to an Excel file using the `to_excel` function. The file is named "synthetic_unemployment_data.xlsx". The `index=True` argument ensures that the region names (the index) are included in the Excel file.

8. Printing a Confirmation Message:

Finally, the script prints a message confirming that the synthetic unemployment data has been successfully saved to the specified Excel file. This provides feedback to the user that the operation was completed.

EXAMPLE 3.4

GUI Tkinter for Sorting Synthetic Unemployment Data by Index

This code generates synthetic unemployment data for 25 regions over a 5-year period and creates a graphical user interface (GUI) using Tkinter. The data is randomly generated, organized into a DataFrame, sorted by region, and then saved to an Excel file. The primary focus of the code is to present this data interactively through a GUI, where users can view the data sorted by either unemployment rate or year. The GUI includes tabs for these sorting options, with each tab displaying the data in a table and accompanied by a corresponding bar chart for visual analysis.

The `UnemploymentDataGUI` class is the core of the application, handling the creation and management of the interface. It provides a user-friendly way to explore the data, allowing users to see how unemployment rates vary across regions and years. The code includes features like scrollable tables, alternating row colors for

readability, and dynamically generated plots embedded in the GUI. This setup is ideal for anyone looking to visualize and interact with multi-year, region-based data in a structured and intuitive environment.

```
import os
import pandas as pd
import numpy as np
import random
import tkinter as tk
from tkinter import ttk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.colors as mcolors

# Set a random seed for reproducibility
random.seed(0)
np.random.seed(0)

# Generate synthetic unemployment data
num_regions = 25
regions = [f'Region {chr(65+i)}' for i in
range(num_regions)] # Region A, Region B, etc.
years = [f'Year {2020+i}' for i in range(5)] # Years from 2020 to 2024

# Generate random unemployment data
unemployment_data = {year: np.random.uniform(3.0,
15.0, size=num_regions) for year in years}

# Create DataFrame
df = pd.DataFrame(unemployment_data,
index=regions)
```

```
# Sort the DataFrame by index (regions)
df_sorted = df.sort_index()

# Save to Excel
file_path = 'synthetic_unemployment_data.xlsx'
df_sorted.to_excel(file_path, index=True)

class UnemploymentDataGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Unemployment Data GUI")

        # Create a Notebook (tabs)
        self.notebook = ttk.Notebook(root)
        self.notebook.pack(fill=tk.BOTH,
                           expand=True)

        # Create frames for each tab
        self.unemployment_frame =
ttk.Frame(self.notebook)
        self.year_frame = ttk.Frame(self.notebook)

        # Add frames as tabs
        self.notebook.add(self.unemployment_frame,
text="Sort by Unemployment Rate")
        self.notebook.add(self.year_frame,
text="Sort by Year")

        # Create widgets for the "Sort by
Unemployment Rate" tab
        self.create_unemployment_tab_widgets()

        # Create widgets for the "Sort by Year"
tab
        self.create_year_tab_widgets()
```

```
# Load initial data
self.refresh_data()

def create_unemployment_tab_widgets(self):
    # Table to display DataFrame sorted by
    average unemployment rate
    self.tree_unemployment =
    ttk.Treeview(self.unemployment_frame,
    columns=list(df.columns), show='headings')
    self.tree_unemployment.pack(side=tk.LEFT,
    fill=tk.BOTH, expand=True)

    # Vertical scrollbar
    self.v_scroll_unemployment =
    ttk.Scrollbar(self.unemployment_frame,
    orient=tk.VERTICAL,
    command=self.tree_unemployment.yview)

    self.v_scroll_unemployment.pack(side=tk.RIGHT,
    fill=tk.Y)

    self.tree_unemployment.configure(yscrollcommand=self.v_scroll_unemployment.set)

    # Horizontal scrollbar
    self.h_scroll_unemployment =
    ttk.Scrollbar(self.unemployment_frame,
    orient=tk.HORIZONTAL,
    command=self.tree_unemployment.xview)

    self.h_scroll_unemployment.pack(side=tk.BOTTOM,
    fill=tk.X)

    self.tree_unemployment.configure(xscrollcommand=self.h_scroll_unemployment.set)
```

```
        for col in df.columns:
            self.tree_unemployment.heading(col,
text=col)
            self.tree_unemployment.column(col,
width=150)

        # Create a Figure and add one subplot for
unemployment data
        self.figure_unemployment,
self.ax_unemployment = plt.subplots(figsize=(8,
6), dpi=100)
        self.canvas_unemployment =
FigureCanvasTkAgg(self.figure_unemployment,
master=self.unemployment_frame)

        self.canvas_unemployment.get_tk_widget().pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

    def create_year_tab_widgets(self):
        # Table to display DataFrame sorted by
year
        self.tree_year =
ttk.Treeview(self.year_frame,
columns=list(df.columns), show='headings')
        self.tree_year.pack(side=tk.LEFT,
fill=tk.BOTH, expand=True)

        # Vertical scrollbar
        self.v_scroll_year =
ttk.Scrollbar(self.year_frame, orient=tk.VERTICAL,
command=self.tree_year.yview)
        self.v_scroll_year.pack(side=tk.RIGHT,
fill=tk.Y)

        self.tree_year.configure(yscrollcommand=self.v_sc
roll_year.set)
```

```
        # Horizontal scrollbar
        self.h_scroll_year =
ttk.Scrollbar(self.year_frame,
orient=tk.HORIZONTAL,
command=self.tree_year.xview)
        self.h_scroll_year.pack(side=tk.BOTTOM,
fill=tk.X)

    self.tree_year.configure(xscrollcommand=self.h_sc
roll_year.set)

    for col in df.columns:
        self.tree_year.heading(col, text=col)
        self.tree_year.column(col, width=150)

        # Create a Figure and add one subplot for
unemployment data by year
        self.figure_year, self.ax_year =
plt.subplots(figsize=(8, 6), dpi=100)
        self.canvas_year =
FigureCanvasTkAgg(self.figure_year,
master=self.year_frame)

    self.canvas_year.get_tk_widget().pack(side=tk.RIG
HT, fill=tk.BOTH, expand=True)

    def refresh_data(self):
        # Initialize or refresh tables and graphs
for both tabs
        avg_unemployment =
df.mean(axis=1).sort_values(ascending=False)
        self.update_table(self.tree_unemployment,
df.loc[avg_unemployment.index])
        self.update_table(self.tree_year,
df.sort_index())
```

```

        # Plot data for unemployment by region
        self.plot_graph(self.ax_unemployment,
df.loc[avg_unemployment.index],
avg_unemployment.index, 'Average Unemployment
Rate', 'Unemployment')

        # Plot data for unemployment by year
        self.plot_graph(self.ax_year, df.T,
df.columns, 'Unemployment Rate', 'Year')

    def update_table(self, tree, data):
        for row in tree.get_children():
            tree.delete(row)

            for index, (_, row) in
enumerate(data.iterrows()):
                row_color = '#f0f0f0' if index % 2 ==
0 else '#ffffff'

                tree.insert(
                    '',
                    tk.END,
                    values=list(row),
                    tags=('oddrow' if index % 2 == 0
else 'evenrow'))
            )

            tree.tag_configure('oddrow',
background='#f0f0f0')
            tree.tag_configure('evenrow',
background='#ffffff')

    def plot_graph(self, ax, data, labels, ylabel,
title):
        ax.clear()

```

```
if data.empty:
    ax.set_title("No data available")
    ax.set_xlabel("")
    ax.set_ylabel("")
else:
    # Generate a list of colors for the
bars
    colors =
list(mcolors.TABLEAU_COLORS.values())
    num_colors = len(colors)
    bar_colors = [colors[i % num_colors]
for i in range(len(data))]

    bars = ax.bar(labels,
data.mean(axis=1), color=bar_colors, label=title,
alpha=0.8)
    ax.set_xlabel('Years' if ylabel ==
'Unemployment Rate' else 'Regions')
    ax.set_ylabel(ylabel)
    ax.set_title(f'{title} by {"Region" if
ylabel == "Unemployment Rate" else "Year"}')
    ax.legend()
    ax.set_xticklabels(labels,
rotation=45, ha='right', fontsize=8)

    # Add y-values on top of each bar
for bar in bars:
    yval = bar.get_height()
    ax.text(
        bar.get_x() + bar.get_width() /
2, yval,
            f'{yval:.1f}', # Format as
one decimal
            ha='center', va='bottom',
            fontsize=8
```

```
        )

    self.canvas_unemployment.draw() if ax ==
self.ax_unemployment else self.canvas_year.draw()

if __name__ == "__main__":
    root = tk.Tk()
    app = UnemploymentDataGUI(root)
    root.mainloop()
```

Here's a step-by-step explanation of the code:

1. Importing Required Libraries

The script starts by importing several Python libraries:

- os: Used for interacting with the operating system (not directly used in this code).
- pandas and numpy: For handling and generating data.
- random: For generating random numbers.
- tkinter and ttk: For creating the graphical user interface (GUI).
- matplotlib and FigureCanvasTkAgg: For creating and embedding plots within the GUI.

2. Setting Random Seeds

The script sets random seeds using both random and numpy to ensure that the generated random data is reproducible every time the script is run.

3. Generating Synthetic Data

- The script defines a list of 25 regions (labeled "Region A" through "Region Y") and a list of 5 years (2020 to 2024).
- For each year, it generates random unemployment rates for each region, which are uniformly distributed between 3.0% and 15.0%.

4. Creating and Sorting a DataFrame

- The random unemployment data is organized into a pandas DataFrame, with regions as the rows and years as the columns.
- The DataFrame is then sorted alphabetically by the region names.

5. Saving Data to an Excel File

The sorted DataFrame is saved to an Excel file named "synthetic_unemployment_data.xlsx". This allows the data to be easily accessed and shared outside of the script.

6. Creating the UnemploymentDataGUI Class

A class called UnemploymentDataGUI is defined to create a GUI for visualizing and interacting with the unemployment data.

a) Initializing the GUI

- The `__init__` method sets up the main window, adds a notebook widget (a tabbed interface), and creates two tabs: one for sorting by unemployment rate and another for sorting by year.
- It calls methods to create widgets for each tab and refreshes the data displayed in the GUI.

b) Creating Widgets for the Unemployment Rate Tab

- The `create_unemployment_tab_widgets` method sets up a table to display the DataFrame, sorted by the average unemployment rate across all years.
- It adds scrollbars for navigating through the data and creates a matplotlib figure to display a bar chart of unemployment rates.

c) Creating Widgets for the Year Tab

- The `create_year_tab_widgets` method is similar to the unemployment tab but focuses on displaying the data sorted by year.
- It also sets up a table and scrollbars, along with a `matplotlib` figure to display the data by year.

d) Refreshing Data

The refresh_data method updates the tables and graphs with the latest data. It calculates the average unemployment rate for each region, sorts the data, and then updates the tables and plots.

e) Updating the Table

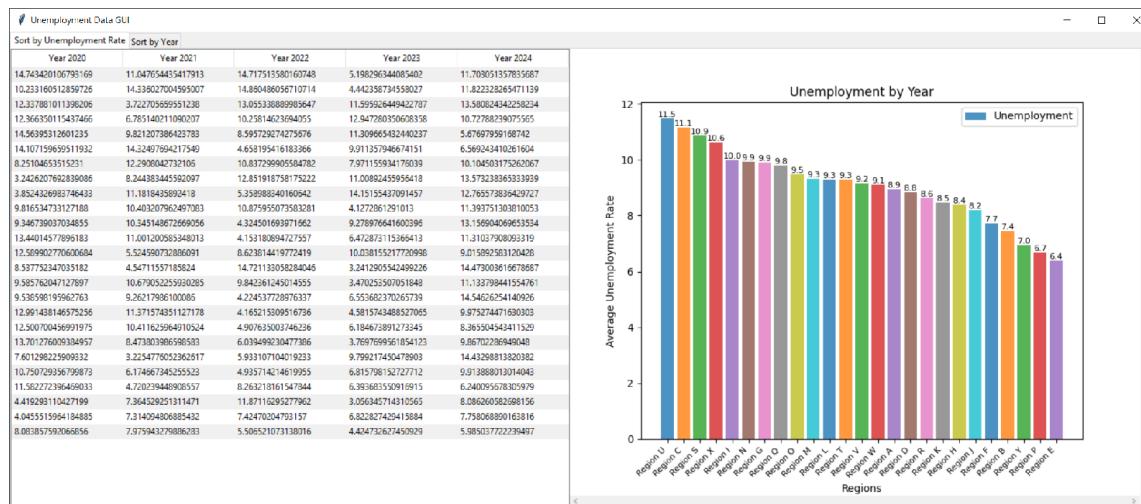
The update_table method clears the existing data in a table and populates it with new data. It alternates row colors for better readability.

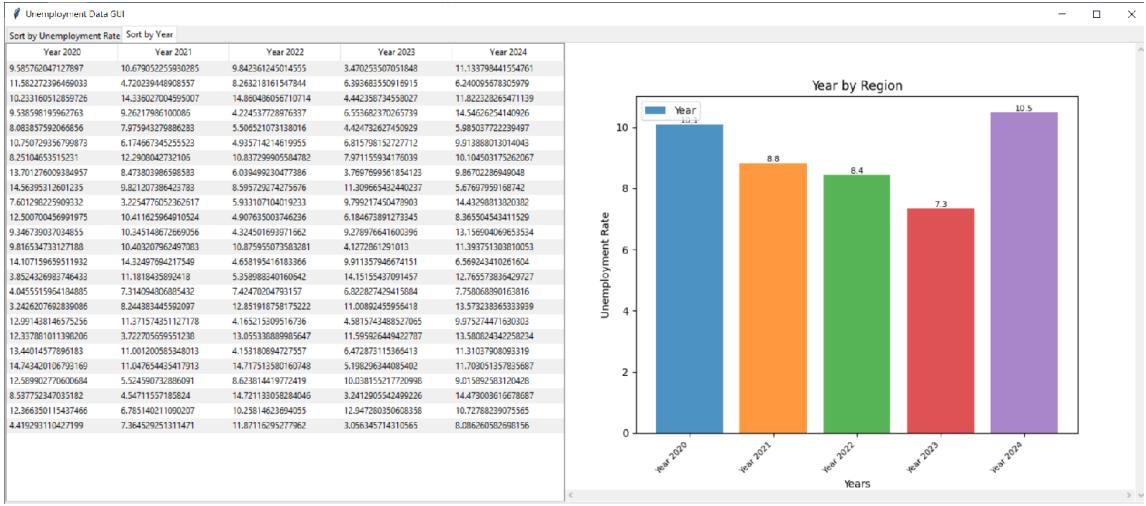
f) Plotting Graphs

The plot_graph method generates bar charts for the unemployment data. It customizes the plot, such as setting labels, adding titles, and displaying the values on top of the bars.

7. Running the GUI

The script checks if it is being run as the main program and then creates an instance of the UnemploymentDataGUI class to start the GUI. The root.mainloop() call enters the main event loop, making the GUI responsive to user interactions.





JOINING AND MERGING

Joining DataFrames in pandas refers to the process of combining two or more DataFrames based on a common key or index. This operation is essential for merging related data stored in different DataFrames into a single DataFrame, enabling more comprehensive analysis. There are several methods for joining DataFrames in pandas, each serving different use cases:

Concatenation (pd.concat):

- Purpose: Concatenation is used when you want to stack DataFrames either vertically (one below the other) or horizontally (side by side).
- Parameters:
 - axis: Determines the direction of concatenation. axis=0 (default) stacks rows, and axis=1 stacks columns.
 - ignore_index: If True, the resulting DataFrame will have a continuous index (useful when concatenating vertically).

- keys: If provided, forms a hierarchical index using the keys.

Example:

```
import pandas as pd
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})
# Vertical concatenation
result1 = pd.concat([df1, df2], axis=0)
print(result1)

# Horizontal concatenation
result2 = pd.concat([df1, df2], axis=1)
print(result2)
```

| | A | B |
|---|---|---|
| 0 | 1 | 3 |
| 1 | 2 | 4 |

| | A | B | A | B |
|---|---|---|---|---|
| 0 | 1 | 3 | 5 | 7 |
| 1 | 2 | 4 | 6 | 8 |

Result:

Vertical concatenation combines the rows, aligning them by columns.

Horizontal concatenation combines the columns, aligning them by rows.

Merging (pd.merge)

- Purpose: Merging is used to combine two DataFrames based on one or more common columns or indexes. It's similar to SQL joins.
- Join Types:
 - Inner Join: Keeps only rows with matching keys in both DataFrames.
 - Outer Join: Keeps all rows from both DataFrames, with NaN for missing matches.
 - Left Join: Keeps all rows from the left DataFrame and matching rows from the right.
 - Right Join: Keeps all rows from the right DataFrame and matching rows from the left.
- Parameters:
 - on: Specifies the column(s) to join on. If not provided, merge uses common column names.
 - how: Specifies the type of join ('inner', 'outer', 'left', 'right').
 - left_on, right_on: Allows joining on different columns from the left and right DataFrames.
 - suffixes: Adds suffixes to overlapping column names.

Example of inner join

```
import pandas as pd
df1 = pd.DataFrame({'key': ['A', 'B', 'C'],
 'value1': [1, 2, 3]})
df2 = pd.DataFrame({'key': ['B', 'C', 'D'],
 'value2': [4, 5, 6]})
result = pd.merge(df1, df2, on='key', how='inner')
print(result)
```

Result:

The result will include only rows with keys present in both DataFrames (in this case, 'B' and 'C').

| | key | value1 | value2 |
|---|-----|--------|--------|
| 0 | B | 2 | 4 |
| 1 | C | 3 | 5 |

Example of outer join

An outer join combines all rows from both DataFrames, and where there is no match, it fills in with NaN.

Let's say you have two DataFrames, df1 and df2, with some overlapping and some unique keys.

DataFrame 1 (df1):

```
import pandas as pd

df1 = pd.DataFrame({
    'key': ['A', 'B', 'C'],
    'value1': [1, 2, 3]
})
print(df1)
```

| | key | value1 |
|---|-----|--------|
| 0 | A | 1 |
| 1 | B | 2 |
| 2 | C | 3 |

DataFrame 2 (df2):

```
df2 = pd.DataFrame({
    'key': ['B', 'C', 'D'],
    'value2': [4, 5, 6]
```

```
)  
print(df2)
```

| | key | value2 |
|---|-----|--------|
| 0 | B | 4 |
| 1 | C | 5 |
| 2 | D | 6 |

Performing an Outer Join

```
result = pd.merge(df1, df2, on='key', how='outer')  
print(result)
```

Output

| | key | value1 | value2 |
|---|-----|--------|--------|
| 0 | A | 1.0 | NaN |
| 1 | B | 2.0 | 4.0 |
| 2 | C | 3.0 | 5.0 |
| 3 | D | NaN | 6.0 |

Explanation:

- Key 'A' is only in df1, so value2 is NaN.
- Key 'B' and 'C' are in both df1 and df2, so both value1 and value2 have values.
- Key 'D' is only in df2, so value1 is NaN.

This outer join ensures that all keys from both DataFrames are included, with missing values filled in with NaN where necessary.

Example of left join

A left join returns all rows from the left DataFrame and the matched rows from the right DataFrame. If there is no match, the

result will have NaN for the columns from the right DataFrame.

Let's create two DataFrames, df1 and df2, to demonstrate a left join.

DataFrame 1 (df1):

```
import pandas as pd

df1 = pd.DataFrame({
    'key': ['A', 'B', 'C'],
    'value1': [1, 2, 3]
})
print(df1)
```

Output:

| | key | value1 |
|---|-----|--------|
| 0 | A | 1 |
| 1 | B | 2 |
| 2 | C | 3 |

DataFrame 2 (df2):

```
df2 = pd.DataFrame({
    'key': ['B', 'C', 'D'],
    'value2': [4, 5, 6]
})
print(df2)
```

Output:

| | key | value2 |
|---|-----|--------|
| 0 | B | 4 |
| 1 | C | 5 |

Performing a Left Join

```
result = pd.merge(df1, df2, on='key', how='left')
print(result)
```

Output

| | key | value1 | value2 |
|---|-----|--------|--------|
| 0 | A | 1 | NaN |
| 1 | B | 2 | 4.0 |
| 2 | C | 3 | 5.0 |

Explanation:

- Key 'A' is only in df1, so it is included in the result with value2 as NaN.
- Key 'B' and 'C' are in both df1 and df2, so both value1 and value2 have values in the result.
- Key 'D' is only in df2, so it is excluded from the result since it has no match in df1.

The left join ensures that all rows from df1 are included in the result, with corresponding values from df2 where a match exists. If there is no match, the columns from df2 will contain NaN.

Example of right join

A right join returns all rows from the right DataFrame and the matched rows from the left DataFrame. If there is no match, the result will have NaN for the columns from the left DataFrame.

Let's create two DataFrames, df1 and df2, to demonstrate a right join.

DataFrame 1 (df1):

```
import pandas as pd
df1 = pd.DataFrame({
    'key': ['A', 'B', 'C'],
    'value1': [1, 2, 3]
})
print(df1)
```

Output:

| | key | value1 |
|---|-----|--------|
| 0 | A | 1 |
| 1 | B | 2 |
| 2 | C | 3 |

DataFrame 2 (df2):

```
df2 = pd.DataFrame({
    'key': ['B', 'C', 'D'],
    'value2': [4, 5, 6]
})
print(df2)
```

Output:

| | key | value2 |
|---|-----|--------|
| 0 | B | 4 |
| 1 | C | 5 |
| 2 | D | 6 |

Performing a Right Join

```
result = pd.merge(df1, df2, on='key', how='right')
print(result)
```

Output

| | key | value1 | value2 |
|---|-----|--------|--------|
| 0 | B | 2.0 | 4 |
| 1 | C | 3.0 | 5 |
| 2 | D | NaN | 6 |

Explanation:

- Key 'B' and 'C' are in both df1 and df2, so both value1 and value2 have values in the result.
- Key 'D' is only in df2, so it is included in the result with value1 as NaN.
- Key 'A' is only in df1, so it is excluded from the result since it has no match in df2.

The right join ensures that all rows from df2 are included in the result, with corresponding values from df1 where a match exists. If there is no match, the columns from df1 will contain NaN.

Joining (df.join)

- Purpose: join is used primarily for combining DataFrames based on their indexes. It's convenient when you need to merge DataFrames with similar structures or when working with time-series data.
- Parameters:
 - on: Specifies a column or index level to join on (if not using index).
 - how: Specifies the type of join ('left', 'right', 'inner', 'outer').

- lsuffix, rsuffix: Suffixes added to overlapping column names.

Example:

```
df1 = pd.DataFrame({'A': [1, 2, 3]}, index=['a', 'b', 'c'])
df2 = pd.DataFrame({'B': [4, 5, 6]}, index=['a', 'b', 'd'])
result = df1.join(df2, how='inner')
```

Output:

| | A | B |
|---|---|---|
| a | 1 | 4 |
| b | 2 | 5 |

Result:

The result will include only rows with matching indexes ('a' and 'b').

Choosing the Right Method

- Concatenation is ideal when your DataFrames are aligned and you need to simply stack them together. It's useful for combining data that comes in chunks or for appending new data.
- Merging provides the most flexibility, especially when dealing with relational data. It allows you to combine DataFrames on specific columns or indexes and supports different types of joins to control how the data is combined.
- Joining is most useful when working with DataFrames that have a common index or when you're working with

hierarchical or multi-level indexes. It's a quick and convenient way to combine DataFrames in a way that maintains their index structure.

Practical Considerations

- Performance: Merging large DataFrames can be computationally expensive, especially with complex joins. It's important to consider the size of the DataFrames and the efficiency of the join operation.
- Data Integrity: When joining, ensure that the key or index you're joining on is clean and consistent across DataFrames. Inconsistent or missing keys can lead to unexpected NaN values in the result.
- Debugging: When results don't look right, check for overlapping column names, missing keys, or mismatched data types in the joining columns. Adding suffixes to overlapping columns can help clarify the resulting DataFrame structure.

By understanding these details, you can effectively combine DataFrames to create a more comprehensive dataset tailored to your analysis needs.

EXAMPLE 3.5

Concatenating Dataframes Using Synthetic Temperature Data for Different Countries

Let's walk through an example of concatenating dataframes, generating synthetic temperature data for different countries, and saving it to an Excel file using Python.

1. Generate Synthetic Temperature Data

First, we'll generate synthetic data for different countries with temperatures for different months.

```
import pandas as pd
import numpy as np

# Define the list of countries
countries = ['USA', 'Canada', 'Brazil',
'Australia', 'India']

# Generate synthetic temperature data for each
country
data = []
for country in countries:
    for month in range(1, 13): # Months from 1 to
12
        data.append({
            'Country': country,
            'Month': month,
            'Temperature': np.random.uniform(low=-10, high=35) # Random
temperature between -10 and 35
        })

# Create a DataFrame
df = pd.DataFrame(data)
```

2. Create Additional DataFrames

Let's create another dataframe with different data to concatenate.

```
# Create another dataframe with similar structure
but different values
additional_data = []
```

```
for country in ['UK', 'Germany', 'France',
'Japan', 'South Korea']:
    for month in range(1, 13):
        additional_data.append({
            'Country': country,
            'Month': month,
            'Temperature': np.random.uniform(low=-5, high=30)
        })

# Create another DataFrame
df_additional = pd.DataFrame(additional_data)
```

3. Concatenate DataFrames

Now, let's concatenate the original and additional dataframes.

```
# Concatenate the dataframes
df_combined = pd.concat([df, df_additional],
ignore_index=True)
```

4. Save to Excel

Finally, we'll save the concatenated dataframe to an Excel file.

```
# Save the dataframe to an Excel file
df_combined.to_excel('combined_temperatures.xlsx',
index=False)
```

Summary

1. Generate Synthetic Data: Created data for temperatures across different countries and months.
2. Create Additional DataFrames: Added another set of synthetic data.
3. Concatenate DataFrames: Combined both dataframes.

4. Save to Excel: Exported the final dataframe to an Excel file.

EXAMPLE 3.6

GUI Tkinter for Concatenating Dataframes Using Synthetic Temperature Data for Different Countries

This code creates a comprehensive Tkinter application that visualizes temperature data, with tabs for viewing raw data, graphical representation, and data filtering. The alternating row colors in tables enhance readability, and filters allow users to view specific subsets of the data.

```
import tkinter as tk
from tkinter import ttk, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Generate synthetic temperature data
def generate_data(countries, months_range):
    data = []
    for country in countries:
        for month in months_range:
            data.append({
                'Country': country,
                'Month': month,
                'Temperature':
np.random.uniform(low=-10, high=35)
            })
    return pd.DataFrame(data)

# Define countries and months
```

```
countries1 = ['USA', 'Canada', 'Brazil',
'Indonesia']
countries2 = ['Australia', 'India', 'UK']
months = range(1, 13)

# Create dataframes
df1 = generate_data(countries1, months)
df2 = generate_data(countries2, months)

# Concatenate dataframes
df_combined = pd.concat([df1, df2],
ignore_index=True)

# Tkinter application
class TemperatureApp(tk.Tk):
    def __init__(self, df):
        super().__init__()
        self.title('Temperature Data Viewer')
        self.geometry('800x600')
        self.df = df

        # Create a Notebook (tabbed interface)
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(expand=True,
fill='both')

        # Create frames for each tab
        self.create_table_tab()
        self.create_graph_tab()
        self.create_filter_tab()

    def create_table_tab(self):
        self.table_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.table_frame,
text='Data Table')
```

```
# Create a Treeview for the table
self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
self.tree.pack(expand=True, fill='both')

# Define columns
for col in self.df.columns:
    self.tree.heading(col, text=col)
    self.tree.column(col, width=100,
anchor='center')

# Define tags for alternating row colors
self.tree.tag_configure('evenrow',
background='#f0f0f0')
self.tree.tag_configure('oddrow',
background='#ffffff')

# Insert data
self.update_table(self.df)

def create_graph_tab(self):
    self.graph_frame =
ttk.Frame(self.notebook)
    self.notebook.add(self.graph_frame,
text='Temperature Graph')

    # Create a Matplotlib figure and axis
    self.fig, self.ax = plt.subplots(figsize=
(6, 4))
        self.ax.set_title('Average Monthly
Temperature by Country')
        self.ax.set_xlabel('Month')
        self.ax.set_ylabel('Temperature')

    # Create a canvas for Matplotlib
```

```
        self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)

    self.canvas.get_tk_widget().pack(expand=True,
fill='both')

    # Plot data
    self.update_plot()

    def create_filter_tab(self):
        self.filter_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.filter_frame,
text='Filters')

        # Create filters for Country and Month
        self.country_label =
ttk.Label(self.filter_frame, text='Select
Country:')
        self.country_label.pack(pady=5)

        self.country_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Country'].unique()))
        self.country_combobox.pack(pady=5)

    self.country_combobox.bind('<>'
, self.apply_filters)

        self.month_label =
ttk.Label(self.filter_frame, text='Select Month:')
        self.month_label.pack(pady=5)

        self.month_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Month'].unique()))
```

```
    self.month_combobox.pack(pady=5)

    self.month_combobox.bind('<>',
self.apply_filters)

        self.reset_button =
ttk.Button(self.filter_frame, text='Reset
Filters', command=self.reset_filters)
        self.reset_button.pack(pady=5)

        # Create a Treeview for filtered data in
the Filters tab
        self.filter_tree =
ttk.Treeview(self.filter_frame,
columns=list(self.df.columns), show='headings')
        self.filter_tree.pack(expand=True,
fill='both')

        # Define columns
        for col in self.df.columns:
            self.filter_tree.heading(col,
text=col)
            self.filter_tree.column(col,
width=100, anchor='center')

        # Define tags for alternating row colors
        self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
        self.filter_tree.tag_configure('oddrow',
background='#ffffff')

        # Insert initial data
        self.update_filter_table(self.df)

def update_table(self, data):
    for row in self.tree.get_children():
```

```

        self.tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else
'oddrow'
            self.tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_plot(self):
        self.ax.clear()
        self.ax.set_title('Average Monthly
Temperature by Country')
        self.ax.set_xlabel('Month')
        self.ax.set_ylabel('Temperature')

        for country in
self.df['Country'].unique():
            country_data =
self.df[self.df['Country'] == country]
            self.ax.plot(country_data['Month'],
country_data['Temperature'], marker='o',
label=country)

        self.ax.legend()
        self.canvas.draw()

    def apply_filters(self, event=None):
        country = self.country_combobox.get()
month = self.month_combobox.get()

        filtered_df = self.df

        if country:
            filtered_df =
filtered_df[filtered_df['Country'] == country]

        if month:

```

```

        try:
            filtered_df =
filtered_df[filtered_df['Month'] == int(month)]
        except ValueError:
            messagebox.showwarning("Invalid
Month", "Please select a valid month.")
        return

    if filtered_df.empty:
        messagebox.showinfo("No Data", "No
data available for the selected filters.")
    filtered_df = self.df

    # Update the table in the Filters tab
    self.update_filter_table(filtered_df)

def reset_filters(self):
    self.country_combobox.set('')
    self.month_combobox.set('')
    self.update_filter_table(self.df)

def update_filter_table(self, data):
    for row in
self.filter_tree.get_children():
        self.filter_tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else
'oddrow'
        self.filter_tree.insert('', 'end',
values=list(row), tags=(tag,))

if __name__ == "__main__":
    app = TemperatureApp(df_combined)
    app.mainloop()

```

Here's a detailed explanation of the provided code for the Tkinter application:

1. Imports

```
import tkinter as tk
from tkinter import ttk, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from    matplotlib.backends.backend_tkagg    import
FigureCanvasTkAgg
```

- tkinter: For creating the GUI.
- ttk: Provides themed widgets for Tkinter.
- messagebox: For displaying dialog boxes.
- pandas: For data manipulation and analysis.
- numpy: For generating random numbers.
- matplotlib: For creating plots.
- FigureCanvasTkAgg: Integrates Matplotlib figures into Tkinter.

2. Data Generation Function

```
def generate_data(countries, months_range):
    data = []
    for country in countries:
        for month in months_range:
            data.append({
                'Country': country,
                'Month': month,
                'Temperature': np.random.uniform(low=-10, high=35)
```

```
        })
return pd.DataFrame(data)
```

- Purpose: Generates synthetic temperature data for given countries and months.
- Data Structure: List of dictionaries, each representing a country-month-temperature combination.
- Output: A Pandas DataFrame containing the generated data.

3. Define Data and Create DataFrames

```
countries1      =      ['USA',      'Canada',      'Brazil',
'Indonesia']
countries2 = ['Australia', 'India', 'UK']
months = range(1, 13)

df1 = generate_data(countries1, months)
df2 = generate_data(countries2, months)

df_combined      =      pd.concat([df1,           df2],
ignore_index=True)
```

- Define Countries and Months: Two lists of countries and a range of months.
- Generate DataFrames: Create df1 and df2 for the defined countries.
- Combine DataFrames: Concatenate df1 and df2 into a single DataFrame, df_combined.

4. Create the Tkinter Application Class

```
class TemperatureApp(tk.Tk):
```

```

def __init__(self, df):
    super().__init__()
    self.title('Temperature Data Viewer')
    self.geometry('800x600')
    self.df = df

    # Create a Notebook (tabbed interface)
    self.notebook = ttk.Notebook(self)
                self.notebook.pack(expand=True,
fill='both')

    # Create frames for each tab
    self.create_table_tab()
    self.create_graph_tab()
    self.create_filter_tab()

```

- Class Inheritance: Inherits from tk.Tk to create the main window.
- Initialization: Sets the window title and size, initializes the DataFrame, and creates tabs for different views.

5. Create Table Tab

```

def create_table_tab(self):
    self.table_frame = ttk.Frame(self.notebook)
    self.notebook.add(self.table_frame, text='DataTable')

    # Create a Treeview for the table
        self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
    self.tree.pack(expand=True, fill='both')

    # Define columns

```

```

        for col in self.df.columns:
            self.tree.heading(col, text=col)
            self.tree.column(col, width=100,
anchor='center')

        # Define tags for alternating row colors
            self.tree.tag_configure('evenrow',
background='#f0f0f0')
            self.tree.tag_configure('oddrow',
background='#ffffff')

        # Insert data
        self.update_table(self.df)
    
```

- Frame Creation: Adds a frame for the table view to the notebook.
- Treeview Widget: Displays the DataFrame in a tabular format.
- Column Definitions: Sets headings and column widths.
- Row Colors: Defines tags for alternating row colors.
- Update Table: Calls update_table to populate the table.

6. Create Graph Tab

```

def create_graph_tab(self):
    self.graph_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.graph_frame,
text='Temperature Graph')

    # Create a Matplotlib figure and axis
    self.fig, self.ax = plt.subplots(figsize=(6,
4))
        self.ax.set_title('Average Monthly Temperature
by Country')
    
```

```

        self.ax.set_xlabel('Month')
        self.ax.set_ylabel('Temperature')

        # Create a canvas for Matplotlib
        self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)
        self.canvas.get_tk_widget().pack(expand=True,
fill='both')

        # Plot data
        self.update_plot()

```

- Frame Creation: Adds a frame for the graph view to the notebook.
- Matplotlib Figure: Creates a figure and axis for plotting.
- Canvas Widget: Embeds the Matplotlib figure into the Tkinter application.
- Update Plot: Calls update_plot to render the graph.

7. Create Filter Tab

```

def create_filter_tab(self):
    self.filter_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.filter_frame,
text='Filters')

        # Create filters for Country and Month
                self.country_label      =
ttk.Label(self.filter_frame,           text='Select
Country:')
        self.country_label.pack(pady=5)

                self.country_combobox     =
ttk.Combobox(self.filter_frame,

```

```
values=sorted(self.df['Country'].unique()))
    self.country_combobox.pack(pady=5)

self.country_combobox.bind('<>',
self.apply_filters)

                self.month_label      =
ttk.Label(self.filter_frame, text='Select Month:')
    self.month_label.pack(pady=5)

                self.month_combobox      =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Month'].unique()))
    self.month_combobox.pack(pady=5)

self.month_combobox.bind('<>',
self.apply_filters)

                self.reset_button      =
ttk.Button(self.filter_frame,           text='Reset
Filters', command=self.reset_filters)
    self.reset_button.pack(pady=5)

# Create a Treeview for filtered data in the
Filters tab
                self.filter_tree      =
ttk.Treeview(self.filter_frame,
columns=list(self.df.columns), show='headings')
    self.filter_tree.pack(expand=True,
fill='both')

# Define columns
for col in self.df.columns:
    self.filter_tree.heading(col, text=col)
    self.filter_tree.column(col, width=100,
anchor='center')
```

```

# Define tags for alternating row colors
    self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
    self.filter_tree.tag_configure('oddrow',
background='#ffffff')

# Insert initial data
self.update_filter_table(self.df)

```

- Frame Creation: Adds a frame for the filter view to the notebook.
- Filters: Creates Combobox widgets for selecting country and month, and a reset button.
- Filtered Data Treeview: Displays filtered data with alternating row colors.
- Update Filter Table: Calls update_filter_table to populate the table with initial data.

8. Update Table Method

```

def update_table(self, data):
    for row in self.tree.get_children():
        self.tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.tree.insert('', 'end',
values=list(row), tags=(tag,))

```

- Delete Existing Rows: Clears the existing rows in the table.
- Insert Rows: Inserts new rows with alternating colors.

9. Update Plot Method

```
def update_plot(self):
    self.ax.clear()
    self.ax.set_title('Average Monthly Temperature
by Country')
    self.ax.set_xlabel('Month')
    self.ax.set_ylabel('Temperature')

    for country in self.df['Country'].unique():
        country_data = self.df[self.df['Country']
== country]
            self.ax.plot(country_data['Month'],
country_data['Temperature'], marker='o',
label=country)

    self.ax.legend()
    self.canvas.draw()
```

- Clear Axis: Clears the previous plot.
- Plot Data: Plots temperature data for each country.
- Update Canvas: Draws the updated plot on the canvas.

10. Apply Filters Method

```
def apply_filters(self, event=None):
    country = self.country_combobox.get()
    month = self.month_combobox.get()

    filtered_df = self.df

    if country:
        filtered_df = filtered_df[filtered_df['Country'] == country]
```

```

if month:
    try:
        filtered_df = filtered_df[filtered_df['Month'] == int(month)]
    except ValueError:
        messagebox.showwarning("Invalid Month", "Please select a valid month.")
    return

if filtered_df.empty:
    messagebox.showinfo("No Data", "No data available for the selected filters.")
    filtered_df = self.df

# Update the table in the Filters tab
self.update_filter_table(filtered_df)

```

- Filter Data: Applies the selected filters to the DataFrame.
- Update Filter Table: Updates the filtered data table with the new data.

11. Reset Filters Method

```

def reset_filters(self):
    self.country_combobox.set('')
    self.month_combobox.set('')
    self.update_filter_table(self.df)

```

Reset Filters: Clears the filter selections and updates the filter table with the full DataFrame.

12. Update Filter Table Method

```
def update_filter_table(self, data):
    for row in self.filter_tree.get_children():
        self.filter_tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else 'oddrow'
        self.filter_tree.insert('', 'end',
values=list(row), tags=(tag,))
```

- Delete Existing Rows: Clears the existing rows in the filtered table.
- Insert Rows: Inserts new rows with alternating colors.

13. Run the Application

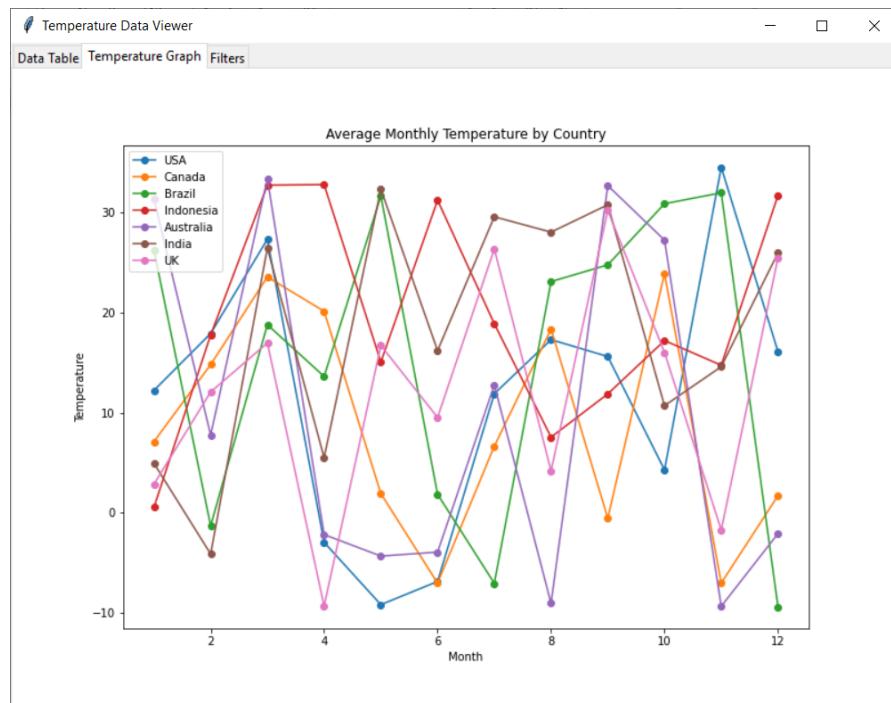
```
if __name__ == "__main__":
    app = TemperatureApp(df_combined)
    app.mainloop()
```

- Create Application: Initializes the TemperatureApp with the combined DataFrame.
- Start Main Loop: Runs the Tkinter event loop to keep the application open and responsive.

Temperature Data Viewer

Data Table Temperature Graph Filters

| Country | Month | Temperature |
|---------|-------|---------------------|
| USA | 1 | 12.219637142092846 |
| USA | 2 | 17.898007327162087 |
| USA | 3 | 27.30414047975804 |
| USA | 4 | -2.9443872409262077 |
| USA | 5 | -9.164070902016572 |
| USA | 6 | -6.849003532634995 |
| USA | 7 | 11.885529992166433 |
| USA | 8 | 17.284825774399867 |
| USA | 9 | 15.59831466889166 |
| USA | 10 | 4.281308419497234 |
| USA | 11 | 34.4877269485602 |
| USA | 12 | 16.08853486606086 |
| Canada | 1 | 7.106352768059768 |
| Canada | 2 | 14.792669860305356 |
| Canada | 3 | 23.540049390792596 |
| Canada | 4 | 20.115480205393308 |
| Canada | 5 | 1.9213800948264232 |
| Canada | 6 | -7.01493245072013 |
| Canada | 7 | 6.653788906134782 |
| Canada | 8 | 18.337287815970402 |
| Canada | 9 | -0.5421695538322187 |
| Canada | 10 | 23.873999918246625 |
| Canada | 11 | -7.005858339064828 |
| Canada | 12 | 1.7141794360343425 |
| Brazil | 1 | 26.21395536845054 |
| Brazil | 2 | -1.2954572819502523 |
| Brazil | 3 | 18.775739639597305 |



Temperature Data Viewer

Data Table | Temperature Graph | Filters

Select Country:

Select Month:

| Country | Month | Temperature |
|---------|-------|---------------------|
| Canada | 1 | 7.106352768059768 |
| Canada | 2 | 14.792669860305356 |
| Canada | 3 | 23.540049390792596 |
| Canada | 4 | 20.115480205393308 |
| Canada | 5 | 1.9213800948264232 |
| Canada | 6 | -7.01493245072013 |
| Canada | 7 | 6.653788906134782 |
| Canada | 8 | 18.337287815970402 |
| Canada | 9 | -0.5421695538322187 |
| Canada | 10 | 23.873999918246625 |
| Canada | 11 | -7.005858339064828 |
| Canada | 12 | 1.7141794360343425 |

Temperature Data Viewer

Data Table | Temperature Graph | Filters

Select Country:

Select Month:

| Country | Month | Temperature |
|---------|-------|--------------------|
| Canada | 4 | 20.115480205393308 |

EXAMPLE 3.7

Performing Inner Join on Dataframes with Synthetic Housing Data

Let's go through a real-world example of performing an inner join on dataframes with synthetic housing data, and then save the results in an Excel file.

Example Scenario

Imagine you have two dataframes:

1. Housing Data: Contains details about houses.
2. Owner Data: Contains details about the owners of the houses.

We'll perform an inner join on these dataframes using a common key (e.g., HouseID), then save the combined data to an Excel file.

Step-by-Step Code

1. Generate Synthetic Data
2. Perform Inner Join
3. Save to Excel

Here's the complete code:

```
import pandas as pd
import numpy as np

# Step 1: Generate Synthetic Data

# Create synthetic housing data
def generate_housing_data(num_records):
    np.random.seed(0) # For reproducibility
    house_ids = np.arange(1, num_records + 1)
    house_prices = np.random.randint(100000,
500000, size=num_records)
```

```

        house_types = np.random.choice(['Detached',
'Semi-Detached', 'Condo'], size=num_records)
locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_records)
return pd.DataFrame({
    'HouseID': house_ids,
    'Price': house_prices,
    'Type': house_types,
    'Location': locations
})

# Create synthetic owner data
def generate_owner_data(num_records):
    np.random.seed(1) # For reproducibility
    house_ids = np.arange(1, num_records + 1)
    owner_names = np.random.choice(['John Doe',
'Jane Smith', 'Alice Johnson', 'Bob Brown', 'Carol
White'], size=num_records)
    owner_contacts = np.random.choice(['555-0100',
'555-0101', '555-0102', '555-0103', '555-0104'],
size=num_records)
    return pd.DataFrame({
        'HouseID': house_ids,
        'OwnerName': owner_names,
        'Contact': owner_contacts
    })

# Generate dataframes
housing_df = generate_housing_data(10)
owner_df = generate_owner_data(10)

# Step 2: Perform Inner Join
combined_df = pd.merge(housing_df, owner_df,
on='HouseID', how='inner')

```

```
# Step 3: Save to Excel
combined_df.to_excel('combined_housing_data.xlsx',
index=False)

print("Combined data has been saved to
'combined_housing_data.xlsx'")
```

Explanation

1. Generate Synthetic Data:

- `generate_housing_data(num_records)`: Creates a DataFrame with house details including HouseID, Price, Type, and Location.
- `generate_owner_data(num_records)`: Creates a DataFrame with owner details including HouseID, OwnerName, and Contact.

2. Perform Inner Join:

`pd.merge(housing_df, owner_df, on='HouseID', how='inner')`: Joins `housing_df` and `owner_df` on HouseID. Only rows with matching HouseID values in both dataframes are included in the result.

3. Save to Excel:

`combined_df.to_excel('combined_housing_data.xlsx', index=False)`: Saves the combined DataFrame to an Excel file named `combined_housing_data.xlsx`. The `index=False` parameter ensures that row indices are not included in the file.

Result

Running this code will create an Excel file `combined_housing_data.xlsx` with the combined data, including details about houses and their respective owners.

EXAMPLE 3.8

GUI Tkinter for Performing Inner Join on Dataframes with Synthetic Housing Data

The purpose of this project is to develop a user-friendly graphical interface for managing and visualizing housing data. By leveraging Python's Tkinter library, the application allows users to interact with a combined dataset of housing and owner information through a tabbed interface. This includes viewing comprehensive tables, applying filters to narrow down data based on location and type, and visualizing house price distributions using Matplotlib. The GUI is designed to simplify data exploration, making it easier for users to analyze trends, identify patterns, and make informed decisions based on the housing dataset.

Furthermore, the application supports essential functionalities such as saving filtered data to an Excel file and resetting filters to revert to the original dataset. This enhances the flexibility and usability of the tool, allowing users to both retain valuable insights and perform multiple analyses. By integrating data visualization and filtering capabilities within a single interface, the project aims to provide a robust solution for housing data analysis, streamlining the process of data examination and presentation for various stakeholders.

```
import tkinter as tk
from tkinter import ttk, messagebox
import pandas as pd
import numpy as np
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.pyplot as plt

# Generate synthetic data functions
def generate_housing_data(num_records):
```

```
np.random.seed(0)
house_ids = np.arange(1, num_records + 1)
house_prices = np.random.randint(100000,
500000, size=num_records)
house_types = np.random.choice(['Detached',
'Semi-Detached', 'Condo'], size=num_records)
locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_records)
return pd.DataFrame({
    'HouseID': house_ids,
    'Price': house_prices,
    'Type': house_types,
    'Location': locations
})

def generate_owner_data(num_records):
    np.random.seed(1)
    house_ids = np.arange(1, num_records + 1)
    owner_names = np.random.choice(['John Doe',
'Jane Smith', 'Alice Johnson', 'Bob Brown', 'Carol
White'], size=num_records)
    owner_contacts = np.random.choice(['555-0100',
'555-0101', '555-0102', '555-0103', '555-0104'],
size=num_records)
    return pd.DataFrame({
        'HouseID': house_ids,
        'OwnerName': owner_names,
        'Contact': owner_contacts
    })

# Generate dataframes
housing_df = generate_housing_data(100)
owner_df = generate_owner_data(100)
combined_df = pd.merge(housing_df, owner_df,
on='HouseID', how='inner')
```

```
class HousingApp(tk.Tk):
    def __init__(self, df):
        super().__init__()
        self.title('Housing Data Viewer')
        self.geometry('1000x800')
        self.df = df

        # Create a Notebook (tabbed interface)
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(expand=True,
fill='both')

        # Create frames for each tab
        self.create_table_tab()
        self.create_filter_tab()
        self.create_graph_tab()

    def create_table_tab(self):
        self.table_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.table_frame,
text='Data Table')

        # Create a Treeview for the table
        self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
        self.tree.pack(expand=True, fill='both')

        # Define columns
        for col in self.df.columns:
            self.tree.heading(col, text=col)
            self.tree.column(col, width=150,
anchor='center')

        # Define tags for alternating row colors
```

```
        self.tree.tag_configure('evenrow',
background='#f0f0f0')
        self.tree.tag_configure('oddrow',
background='#ffffff')

    # Insert data
    self.update_table(self.df)

    def create_filter_tab(self):
        self.filter_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.filter_frame,
text='Filters')

        # Create filters for Location and Type
        self.location_label =
ttk.Label(self.filter_frame, text='Select
Location:')
        self.location_label.pack(pady=5)

        self.location_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Location'].unique()))
        self.location_combobox.pack(pady=5)

        self.location_combobox.bind('<
```

```
    self.type_combobox.bind('<>',
                           self.apply_filters)

        self.reset_button =
ttk.Button(self.filter_frame, text='Reset
Filters', command=self.reset_filters)
        self.reset_button.pack(pady=5)

        self.save_button =
ttk.Button(self.filter_frame, text='Save Filtered
Data', command=self.save_filtered_data)
        self.save_button.pack(pady=5)

        # Create a Treeview for filtered data in
        # the Filters tab
        self.filter_tree =
ttk.Treeview(self.filter_frame,
columns=list(self.df.columns), show='headings')
        self.filter_tree.pack(expand=True,
fill='both')

        # Define columns
        for col in self.df.columns:
            self.filter_tree.heading(col,
text=col)
            self.filter_tree.column(col,
width=150, anchor='center')

        # Define tags for alternating row colors
        self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
        self.filter_tree.tag_configure('oddrow',
background='ffffff')

        # Insert initial data
```

```
    self.update_filter_table(self.df)

    def create_graph_tab(self):
        self.graph_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.graph_frame,
text='Price Distribution')

        # Create a Matplotlib figure and axis
        self.fig, self.ax = plt.subplots(figsize=
(8, 6))
        self.ax.set_title('House Price
Distribution')
        self.ax.set_xlabel('Price')
        self.ax.set_ylabel('Frequency')

        # Create a canvas for Matplotlib
        self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)

        self.canvas.get_tk_widget().pack(expand=True,
fill='both')

        # Plot data
        self.update_plot()

    def update_table(self, data):
        for row in self.tree.get_children():
            self.tree.delete(row)
        for index, row in data.iterrows():
            tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_filter_table(self, data):
```

```

        for row in
self.filter_tree.get_children():
            self.filter_tree.delete(row)
        for index, row in data.iterrows():
            tag = 'evenrow' if index % 2 == 0 else
'oddrow'
            self.filter_tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_plot(self):
        self.ax.clear()
        self.ax.set_title('House Price
Distribution')
        self.ax.set_xlabel('Price')
        self.ax.set_ylabel('Frequency')

        # Plot price distribution
        self.ax.hist(self.df['Price'], bins=10,
edgecolor='black')

        self.canvas.draw()

    def apply_filters(self, event=None):
        location = self.location_combobox.get()
        house_type = self.type_combobox.get()

        filtered_df = self.df

        if location:
            filtered_df =
filtered_df[filtered_df['Location'] == location]

        if house_type:
            filtered_df =
filtered_df[filtered_df['Type'] == house_type]

```

```
    if filtered_df.empty:
        messagebox.showinfo("No Data", "No
data available for the selected filters.")
        filtered_df = self.df

    # Update the table in the Filters tab
    self.update_filter_table(filtered_df)

def reset_filters(self):
    self.location_combobox.set('')
    self.type_combobox.set('')
    self.update_filter_table(self.df)

def save_filtered_data(self):
    filtered_df = self.get_filtered_data()
    if filtered_df.empty:
        messagebox.showinfo("No Data", "No
data to save.")
        return
    file_path = 'filtered_housing_data.xlsx'
    filtered_df.to_excel(file_path,
index=False)
    messagebox.showinfo("Saved", f"Filtered
data has been saved to '{file_path}'")

def get_filtered_data(self):
    location = self.location_combobox.get()
    house_type = self.type_combobox.get()

    filtered_df = self.df

    if location:
        filtered_df =
filtered_df[filtered_df['Location'] == location]

    if house_type:
```

```
        filtered_df =  
filtered_df[filtered_df['Type'] == house_type]  
  
    return filtered_df  
  
if __name__ == "__main__":  
    app = HousingApp(combined_df)  
    app.mainloop()
```

Let's dive deeper into each part of the script, providing more granular details on how it all fits together.

1. Importing Modules

```
import tkinter as tk  
from tkinter import ttk, messagebox  
import pandas as pd  
import numpy as np  
from matplotlib.backends.backend_tkagg import  
FigureCanvasTkAgg  
import matplotlib.pyplot as plt
```

- **tkinter:** The standard GUI toolkit for Python. It provides classes for creating windows, dialogs, buttons, and other GUI components.
- **ttk:** A module within tkinter that provides access to the Tk themed widget set, which has a more modern look compared to the standard tkinter widgets.
- **messagebox:** A module within tkinter for displaying message boxes with alerts or prompts.
- **pandas:** A library for data manipulation and analysis. It's used here to create and manage dataframes.
- **numpy:** A library for numerical operations, used here to generate synthetic data.

- `matplotlib`: A plotting library for creating static, interactive, and animated visualizations in Python.
- `FigureCanvasTkAgg`: A class from `matplotlib.backends.backend_tkagg` that integrates Matplotlib figures with Tkinter.

2. Data Generation Functions

```
def generate_housing_data(num_records):
    np.random.seed(0)
    house_ids = np.arange(1, num_records + 1)
        house_prices = np.random.randint(100000,
500000, size=num_records)
        house_types = np.random.choice(['Detached',
'Semi-Detached', 'Condo'], size=num_records)
        locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_records)
        return pd.DataFrame({
            'HouseID': house_ids,
            'Price': house_prices,
            'Type': house_types,
            'Location': locations
        })
```

- `np.random.seed(0)`: Sets the random seed for reproducibility, ensuring the same random numbers are generated every time.
- `house_ids`: Creates an array of house IDs from 1 to `num_records`.
- `house_prices`: Generates random house prices between \$100,000 and \$500,000.
- `house_types`: Randomly assigns house types from a predefined list.

- locations: Randomly assigns locations from a predefined list.
- pd.DataFrame(): Constructs a DataFrame from the generated data.

```
def generate_owner_data(num_records):
    np.random.seed(1)
    house_ids = np.arange(1, num_records + 1)
    owner_names = np.random.choice(['John Doe',
'Jane Smith', 'Alice Johnson', 'Bob Brown', 'Carol
White'], size=num_records)
    owner_contacts = np.random.choice(['555-0100',
'555-0101', '555-0102', '555-0103', '555-0104'],
size=num_records)
    return pd.DataFrame({
        'HouseID': house_ids,
        'OwnerName': owner_names,
        'Contact': owner_contacts
    })
```

- np.random.seed(1): Sets the seed for reproducibility with a different seed than for housing data.
- owner_names: Randomly assigns owner names from a predefined list.
- owner_contacts: Randomly assigns contact numbers from a predefined list.

3. Combining DataFrames

```
housing_df = generate_housing_data(100)
owner_df = generate_owner_data(100)
combined_df = pd.merge(housing_df, owner_df,
on='HouseID', how='inner')
```

- `generate_housing_data(100)` and `generate_owner_data(100)`: Generate 100 records each for housing and owner data.
- `pd.merge(housing_df, owner_df, on='HouseID', how='inner')`: Combines the two DataFrames using an inner join on the HouseID column, resulting in a DataFrame that includes data from both original DataFrames where HouseID matches.

4. Tkinter Application Class

```
class HousingApp(tk.Tk):
    def __init__(self, df):
        super().__init__()
        self.title('Housing Data Viewer')
        self.geometry('1000x800')
        self.df = df

        # Create a Notebook (tabbed interface)
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(expand=True,
                           fill='both')

        # Create frames for each tab
        self.create_table_tab()
        self.create_filter_tab()
        self.create_graph_tab()
```

- `HousingApp(tk.Tk)`: Inherits from `tk.Tk`, making `HousingApp` a Tkinter window.
- `self.title('Housing Data Viewer')`: Sets the window title.

- `self.geometry('1000x800')`: Sets the initial size of the window.
- `self.df`: Stores the combined DataFrame for use within the class.
- `self.notebook`: Creates a notebook widget (tabbed interface) for organizing different views (tabs).

5. Creating Tabs

Data Table Tab

```
def create_table_tab(self):
    self.table_frame = ttk.Frame(self.notebook)
    self.notebook.add(self.table_frame, text='Data
Table')

    self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
    self.tree.pack(expand=True, fill='both')

    for col in self.df.columns:
        self.tree.heading(col, text=col)
            self.tree.column(col, width=150,
anchor='center')

            self.tree.tag_configure('evenrow',
background='#f0f0f0')
            self.tree.tag_configure('oddrow',
background='#ffffff')

    self.update_table(self.df)
```

- `ttk.Frame(self.notebook)`: Creates a new frame within the notebook for the Data Table tab.

- `ttk.Treeview(self.table_frame, columns=list(self.df.columns), show='headings')`: Creates a treeview widget for displaying tabular data.
- `self.tree.heading(col, text=col)`: Sets the column headers in the treeview.
- `self.tree.column(col, width=150, anchor='center')`: Sets the width and alignment of columns.
- `tag_configure('evenrow', background='#f0f0f0')`: Sets the background color for even rows to improve readability.
- `self.update_table(self.df)`: Updates the treeview with data from the DataFrame.

Filter Tab

```
def create_filter_tab(self):
    self.filter_frame = ttk.Frame(self.notebook)
    self.notebook.add(self.filter_frame,
text='Filters')

    self.location_label      =
ttk.Label(self.filter_frame,           text='Select
Location:')
    self.location_label.pack(pady=5)

    self.location_combobox   =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Location'].unique()))
    self.location_combobox.pack(pady=5)

    self.location_combobox.bind('<<ComboboxSelected>>'
, self.apply_filters)
```

```
    self.type_label = ttk.Label(self.filter_frame,
text='Select Type:')
    self.type_label.pack(pady=5)

                self.type_combobox      =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Type'].unique()))
    self.type_combobox.pack(pady=5)

self.type_combobox.bind('<>',
self.apply_filters)

                self.reset_button      =
ttk.Button(self.filter_frame,           text='Reset
Filters', command=self.reset_filters)
    self.reset_button.pack(pady=5)

                self.save_button      =
ttk.Button(self.filter_frame,  text='Save Filtered
Data', command=self.save_filtered_data)
    self.save_button.pack(pady=5)

                self.filter_tree      =
ttk.Treeview(self.filter_frame,
columns=list(self.df.columns), show='headings')
    self.filter_tree.pack(expand=True,
fill='both')

    for col in self.df.columns:
        self.filter_tree.heading(col, text=col)
        self.filter_tree.column(col, width=150,
anchor='center')

        self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
```

```
        self.filter_tree.tag_configure('oddrow',
background='#ffffff')

    self.update_filter_table(self.df)
```

- `ttk.Label(self.filter_frame, text='Select Location:')`: Creates a label for the location filter.
- `ttk.Combobox(self.filter_frame, values=sorted(self.df['Location'].unique()))`: Creates a dropdown for selecting locations.
- `self.location_combobox.bind('<<ComboboxSelected>>', self.apply_filters)`: Binds the selection event to the filter application method.
- `ttk.Button(self.filter_frame, text='Reset Filters', command=self.reset_filters)`: Creates a button to reset filters.
- `ttk.Button(self.filter_frame, text='Save Filtered Data', command=self.save_filtered_data)`: Creates a button to save filtered data.
- `self.filter_tree`: A treeview widget similar to the one in the Data Table tab, but used to display filtered data.
- `self.update_filter_table(self.df)`: Updates the filter treeview with the initial data.

Graph Tab

```
def create_graph_tab(self):
    self.graph_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.graph_frame,
text='Price Distribution')

    self.fig, self.ax = plt.subplots(figsize=(8,
6))
```

```

        self.ax.set_title('House Price Distribution')
        self.ax.set_xlabel('Price')
        self.ax.set_ylabel('Frequency')

        self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)
        self.canvas.get_tk_widget().pack(expand=True,
fill='both')

        self.update_plot()

```

- plt.subplots(figsize=(8, 6)): Creates a Matplotlib figure and axis for plotting.
- self.ax.set_title('House Price Distribution'): Sets the title of the plot.
- self.canvas = FigureCanvasTkAgg(self.fig, master=self.graph_frame): Integrates the Matplotlib figure with the Tkinter GUI.
- self.canvas.get_tk_widget().pack(expand=True, fill='both'): Adds the Matplotlib canvas to the Tkinter frame.
- self.update_plot(): Updates the plot with the latest data.

6. Update Methods

```

def update_table(self, data):
    for row in self.tree.get_children():
        self.tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else
'oddrow'

```

```
        self.tree.insert('', 'end',
values=list(row), tags=(tag,))
```

- `self.tree.get_children()`: Gets existing rows from the treeview.
- `self.tree.delete(row)`: Deletes existing rows to clear the table.
- `self.tree.insert("", 'end', values=list(row), tags=(tag,))`: Inserts new rows into the treeview with alternating row colors.

```
def update_filter_table(self, data):
    for row in self.filter_tree.get_children():
        self.filter_tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else
'oddrow'
        self.filter_tree.insert('', 'end',
values=list(row), tags=(tag,))
```

- `self.filter_tree`: Similar to `update_table`, but updates the filtered data treeview.

```
def update_plot(self):
    self.ax.clear()
    self.ax.set_title('House Price Distribution')
    self.ax.set_xlabel('Price')
    self.ax.set_ylabel('Frequency')

    self.ax.hist(self.df['Price'], bins=10,
edgecolor='black')

    self.canvas.draw()
```

- `self.ax.clear()`: Clears the previous plot.
- `self.ax.hist(self.df['Price'], bins=10, edgecolor='black')`: Plots a histogram of house prices.
- `self.canvas.draw()`: Redraws the canvas with the updated plot.

7. Filter Management

```
def apply_filters(self, event=None):
    location = self.location_combobox.get()
    house_type = self.type_combobox.get()

    filtered_df = self.df

    if location:
        filtered_df = filtered_df[filtered_df['Location'] == location]

    if house_type:
        filtered_df = filtered_df[filtered_df['Type'] == house_type]

    if filtered_df.empty:
        messagebox.showinfo("No Data", "No data available for the selected filters.")
        filtered_df = self.df

    self.update_filter_table(filtered_df)
```

- `self.location_combobox.get()`: Retrieves the selected location filter.
- `self.type_combobox.get()`: Retrieves the selected type filter.

- `filtered_df = self.df[filtered_df['Location'] == location]`: Filters data based on selected location.
- `filtered_df = self.df[filtered_df['Type'] == house_type]`: Filters data based on selected type.
- `messagebox.showinfo("No Data", "No data available for the selected filters.")`: Shows an alert if no data matches the filters.

```
def reset_filters(self):
    self.location_combobox.set('')
    self.type_combobox.set('')
    self.update_filter_table(self.df)
```

- `self.location_combobox.set("")`: Resets the location filter.
- `self.type_combobox.set("")`: Resets the type filter.
- `self.update_filter_table(self.df)`: Refreshes the table with the original data.

8. Data Saving

```
def save_filtered_data(self):
    filtered_df = self.get_filtered_data()
    if filtered_df.empty:
        messagebox.showinfo("No Data", "No data to save.")
        return
    file_path = 'filtered_housing_data.xlsx'
    filtered_df.to_excel(file_path, index=False)
    messagebox.showinfo("Saved", f"Filtered data has been saved to '{file_path}'")
```

- `filtered_df = self.get_filtered_data()`: Retrieves the currently filtered data.

- `filtered_df.to_excel(file_path, index=False)`: Saves the filtered data to an Excel file.
- `messagebox.showinfo("Saved", f"Filtered data has been saved to '{file_path}'")`: Shows a confirmation message.

```
def get_filtered_data(self):
    location = self.location_combobox.get()
    house_type = self.type_combobox.get()

    filtered_df = self.df

    if location:
        filtered_df = filtered_df[filtered_df['Location'] == location]

    if house_type:
        filtered_df = filtered_df[filtered_df['Type'] == house_type]

    return filtered_df
```

- `self.get_filtered_data()`: Same as `apply_filters`, but returns the filtered DataFrame instead of updating the table.

9. Running the Application

```
if __name__ == "__main__":
    app = HousingApp(combined_df)
    app.mainloop()
```

- `app = HousingApp(combined_df)`: Creates an instance of `HousingApp` with the combined data.

- `app.mainloop()`: Starts the Tkinter event loop, making the application responsive to user interactions.

This script creates a comprehensive Tkinter application for displaying, filtering, and visualizing housing data. It includes features like dynamic table updates, data visualization, and file saving, making it a powerful tool for data analysis.

Housing Data Viewer

Data Table | Filters | Price Distribution

| HouseID | Price | Type | Location | OwnerName | Contact |
|---------|--------|---------------|-------------|---------------|----------|
| 1 | 405711 | Condo | Los Angeles | Bob Brown | 555-0101 |
| 2 | 217952 | Detached | Los Angeles | Carol White | 555-0100 |
| 3 | 252315 | Condo | Phoenix | John Doe | 555-0102 |
| 4 | 459083 | Detached | New York | Jane Smith | 555-0104 |
| 5 | 459783 | Semi-Detached | New York | Bob Brown | 555-0104 |
| 6 | 404137 | Condo | Houston | John Doe | 555-0100 |
| 7 | 222579 | Condo | Chicago | John Doe | 555-0104 |
| 8 | 186293 | Semi-Detached | Houston | Jane Smith | 555-0101 |
| 9 | 474564 | Detached | Chicago | Carol White | 555-0104 |
| 10 | 311543 | Semi-Detached | Phoenix | Carol White | 555-0101 |
| 11 | 312038 | Semi-Detached | Houston | Jane Smith | 555-0100 |
| 12 | 410744 | Detached | Houston | Alice Johnson | 555-0102 |
| 13 | 270584 | Condo | New York | Carol White | 555-0103 |
| 14 | 414764 | Condo | Houston | Alice Johnson | 555-0101 |
| 15 | 180186 | Condo | New York | Carol White | 555-0102 |
| 16 | 117089 | Condo | Phoenix | Bob Brown | 555-0104 |
| 17 | 250055 | Semi-Detached | Chicago | Carol White | 555-0104 |
| 18 | 470775 | Condo | Houston | Alice Johnson | 555-0102 |
| 19 | 320760 | Condo | Phoenix | Carol White | 555-0102 |
| 20 | 463345 | Condo | New York | Alice Johnson | 555-0100 |
| 21 | 355653 | Condo | Chicago | Carol White | 555-0101 |
| 22 | 182457 | Condo | Houston | Jane Smith | 555-0102 |
| 23 | 429843 | Detached | Houston | Jane Smith | 555-0102 |
| 24 | 432752 | Semi-Detached | Los Angeles | John Doe | 555-0100 |
| 25 | 107877 | Condo | Houston | Jane Smith | 555-0101 |
| 26 | 446110 | Condo | Phoenix | Jane Smith | 555-0102 |
| 27 | 495087 | Semi-Detached | Houston | Jane Smith | 555-0104 |
| 28 | 173135 | Condo | Houston | Jane Smith | 555-0100 |
| 29 | 495712 | Semi-Detached | Los Angeles | John Doe | 555-0101 |
| 30 | 470648 | Detached | Los Angeles | Carol White | 555-0102 |
| 31 | 131921 | Condo | Los Angeles | Jane Smith | 555-0101 |
| 32 | 330131 | Condo | Houston | John Doe | 555-0104 |

Housing Data Viewer

Data Table | Filters | Price Distribution

Select Location:

Select Type:

| HouseID | Price | Type | Location | OwnerName | Contact |
|---------|--------|---------------|----------|---------------|----------|
| 6 | 404137 | Condo | Houston | John Doe | 555-0100 |
| 8 | 186293 | Semi-Detached | Houston | Jane Smith | 555-0101 |
| 11 | 312038 | Semi-Detached | Houston | Jane Smith | 555-0100 |
| 12 | 410744 | Detached | Houston | Alice Johnson | 555-0102 |
| 14 | 414764 | Condo | Houston | Alice Johnson | 555-0101 |
| 18 | 470775 | Condo | Houston | Alice Johnson | 555-0102 |
| 22 | 182457 | Condo | Houston | Jane Smith | 555-0102 |
| 23 | 429843 | Detached | Houston | Jane Smith | 555-0102 |
| 25 | 107877 | Condo | Houston | Jane Smith | 555-0101 |
| 27 | 495087 | Semi-Detached | Houston | Jane Smith | 555-0104 |
| 28 | 173135 | Condo | Houston | Jane Smith | 555-0100 |
| 32 | 330131 | Condo | Houston | John Doe | 555-0104 |
| 36 | 321607 | Detached | Houston | Jane Smith | 555-0101 |
| 39 | 184665 | Condo | Houston | Jane Smith | 555-0103 |
| 49 | 290372 | Condo | Houston | John Doe | 555-0101 |
| 56 | 145444 | Detached | Houston | Alice Johnson | 555-0101 |
| 58 | 146522 | Condo | Houston | Jane Smith | 555-0100 |
| 64 | 454361 | Detached | Houston | Jane Smith | 555-0100 |
| 66 | 374278 | Semi-Detached | Houston | John Doe | 555-0103 |
| 77 | 445135 | Condo | Houston | Jane Smith | 555-0100 |
| 78 | 162079 | Semi-Detached | Houston | Bob Brown | 555-0103 |
| 82 | 238084 | Condo | Houston | Alice Johnson | 555-0102 |

Housing Data Viewer

Data Table Filters Price Distribution

Select Location:

Houston

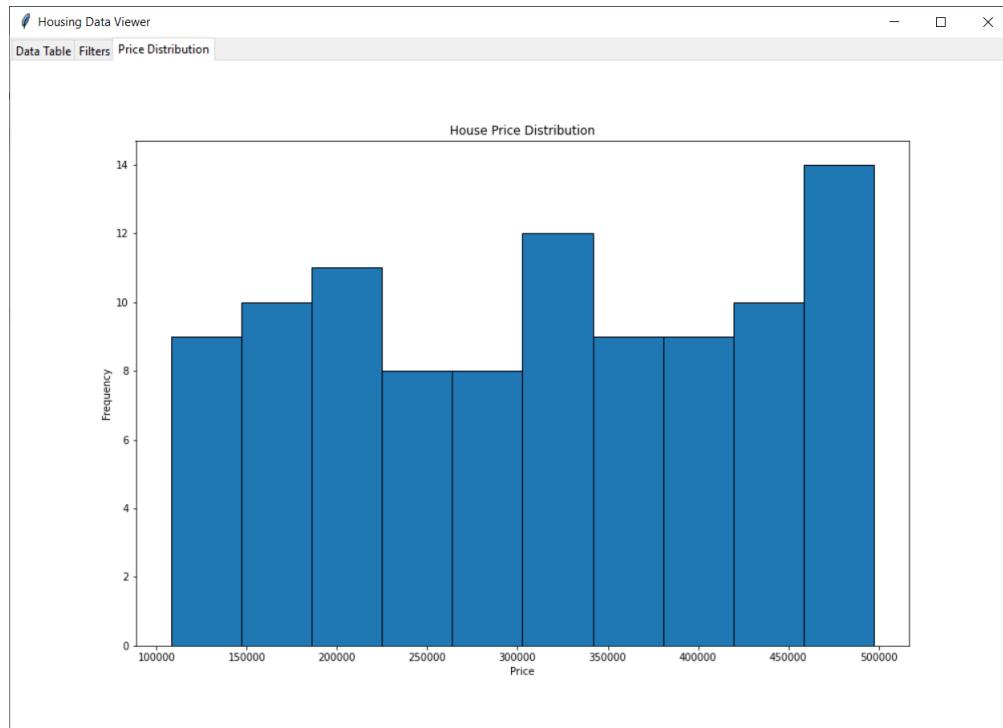
Select Type:

Detached

Reset Filters

Save Filtered Data

| HouseID | Price | Type | Location | OwnerName | Contact |
|---------|--------|----------|----------|---------------|----------|
| 12 | 410744 | Detached | Houston | Alice Johnson | 555-0102 |
| 23 | 429843 | Detached | Houston | Jane Smith | 555-0102 |
| 36 | 321607 | Detached | Houston | Jane Smith | 555-0101 |
| 56 | 145444 | Detached | Houston | Alice Johnson | 555-0101 |
| 64 | 454361 | Detached | Houston | Jane Smith | 555-0100 |
| 93 | 208101 | Detached | Houston | Jane Smith | 555-0103 |



EXAMPLE 3.9

Performing Outer Join on Dataframes with Synthetic Medical Data

To demonstrate an outer join of DataFrames with a real-world example, we can generate a synthetic medical dataset. This dataset will include information about patients and their medical records. The outer join will combine patient information with their respective medical records, ensuring that all patients and records are included even if there is no direct match.

Generating the Synthetic Medical Dataset

Let's create two DataFrames:

1. Patient Data: Contains information about patients.
2. Medical Records Data: Contains information about medical records, with some records possibly missing for patients.

We'll then perform an outer join to combine these DataFrames and save the result in an Excel file.

Here's the code:

```
import pandas as pd
import numpy as np

# Generate synthetic patient data
def generate_patient_data(num_records):
    np.random.seed(0)
    patient_ids = np.arange(1, num_records + 1)
    names = np.random.choice(['Alice Johnson',
    'Bob Brown', 'Carol White', 'David Black', 'Eva Green'],
    size=num_records)
    ages = np.random.randint(20, 80,
    size=num_records)
    return pd.DataFrame({
```

```
'PatientID': patient_ids,
'Name': names,
'Age': ages
})

# Generate synthetic medical records data
def generate_medical_records(num_records):
    np.random.seed(1)
    record_ids = np.arange(1, num_records + 1)
    patient_ids = np.random.choice(np.arange(1, 101), size=num_records) # Random patient IDs (may have duplicates)
    conditions = np.random.choice(['Diabetes', 'Hypertension', 'Asthma', 'Cancer', 'Flu'], size=num_records)
    return pd.DataFrame({
        'RecordID': record_ids,
        'PatientID': patient_ids,
        'Condition': conditions
    })

# Generate dataframes
num_patients = 100
num_records = 120
patient_df = generate_patient_data(num_patients)
medical_records_df = generate_medical_records(num_records)

# Perform outer join
combined_df = pd.merge(patient_df, medical_records_df, on='PatientID', how='outer')

# Save to Excel
file_path = 'outer_join_medical_data.xlsx'
combined_df.to_excel(file_path, index=False)
```

```
print(f"Outer joined medical data has been saved  
to '{file_path}'")
```

Explanation

1. Generate Synthetic Data:

- `generate_patient_data(num_records)`: Creates a DataFrame with patient IDs, names, and ages.
- `generate_medical_records(num_records)`: Creates a DataFrame with medical records, including record IDs, patient IDs, and conditions.

2. Outer Join:

`pd.merge(patient_df, medical_records_df, on='PatientID', how='outer')`: Combines the two DataFrames using an outer join on the PatientID column. This includes all patients and medical records, filling in missing values where there is no match.

3. Save to Excel:

`combined_df.to_excel(file_path, index=False)`: Saves the combined DataFrame to an Excel file named `outer_join_medical_data.xlsx`.

The resulting Excel file will contain a comprehensive dataset with all patients and their corresponding medical records, providing a complete view of the data even if some records are not linked to patients.

EXAMPLE 3.10

GUI Tkinter for Performing Outer Join on Dataframes with Synthetic Medical Data

The purpose of the project is to develop a comprehensive desktop application for visualizing and interacting with medical data using a graphical user interface (GUI). This application aims to facilitate

the exploration and analysis of patient records and medical conditions by providing various tools to display, filter, and visualize the data. By leveraging the Tkinter library for the GUI and Matplotlib for data visualization, the application offers an intuitive interface where users can view detailed patient information, apply filters to narrow down the dataset, and generate visual representations of medical conditions.

In its core functionality, the application allows users to view a combined dataset of patient information and their medical records in a tabular format. The outer join operation merges patient details with their medical conditions, ensuring that all patients and records are included, even if there are discrepancies or missing information. This comprehensive dataset is displayed in a user-friendly table, where users can scroll, sort, and interact with the data. The application also provides filtering options, enabling users to refine their view based on specific criteria such as patient ID or medical condition, which is particularly useful for analyzing subsets of data or focusing on particular conditions.

Beyond data display and filtering, the application incorporates a graphical tab that visualizes the distribution of medical conditions across the dataset. This visualization helps users quickly understand the prevalence of various conditions and identify trends or anomalies. Additionally, the application offers functionality for saving filtered data to an Excel file, allowing users to export and further analyze the data outside of the application. Overall, the project aims to provide a powerful yet accessible tool for healthcare professionals, researchers, and analysts to explore and interpret medical data effectively.

```
import tkinter as tk
```

```
from tkinter import ttk, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Generate synthetic data functions
def generate_patient_data(num_records):
    np.random.seed(0)
    patient_ids = np.arange(1, num_records + 1)
    names = np.random.choice(['Alice Johnson',
    'Bob Brown', 'Carol White', 'David Black', 'Eva
    Green'], size=num_records)
    ages = np.random.randint(20, 80,
size=num_records)
    return pd.DataFrame({
        'PatientID': patient_ids,
        'Name': names,
        'Age': ages
    })

def generate_medical_records(num_records):
    np.random.seed(1)
    record_ids = np.arange(1, num_records + 1)
    patient_ids = np.random.choice(np.arange(1,
101), size=num_records) # Random patient IDs (may
have duplicates)
    conditions = np.random.choice(['Diabetes',
'Hypertension', 'Asthma', 'Cancer', 'Flu'],
size=num_records)
    return pd.DataFrame({
        'RecordID': record_ids,
        'PatientID': patient_ids,
        'Condition': conditions
    })
```

```
# Generate dataframes
num_patients = 100
num_records = 120
patient_df = generate_patient_data(num_patients)
medical_records_df =
generate_medical_records(num_records)
combined_df = pd.merge(patient_df,
medical_records_df, on='PatientID', how='outer')

# Tkinter application
class MedicalApp(tk.Tk):
    def __init__(self, df):
        super().__init__()
        self.title('Medical Data Viewer')
        self.geometry('1200x800')
        self.df = df

        # Create a Notebook (tabbed interface)
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(expand=True,
fill='both')

        # Create frames for each tab
        self.create_table_tab()
        self.create_filter_tab()
        self.create_graph_tab()

    def create_table_tab():
        self.table_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.table_frame,
text='Data Table')

        # Create a Treeview for the table
```

```
        self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
        self.tree.pack(expand=True, fill='both')

        # Define columns
        for col in self.df.columns:
            self.tree.heading(col, text=col)
            self.tree.column(col, width=150,
anchor='center')

        # Define tags for alternating row colors
        self.tree.tag_configure('evenrow',
background='#f0f0f0')
        self.tree.tag_configure('oddrow',
background='#ffffff')

        # Insert data
        self.update_table(self.df)

    def create_filter_tab(self):
        self.filter_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.filter_frame,
text='Filters')

        # Ensure 'Condition' column is treated as
strings
        self.df['Condition'] =
self.df['Condition'].astype(str).fillna('Unknown')

        # Create filters for Patient ID and
Condition
        self.patient_label =
ttk.Label(self.filter_frame, text='Select Patient
ID:')
        self.patient_label.pack(pady=5)
```

```
        self.patient_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['PatientID'].unique())))
        self.patient_combobox.pack(pady=5)

    self.patient_combobox.bind('<>'
, self.apply_filters)

        self.condition_label =
ttk.Label(self.filter_frame, text='Select
Condition:')
        self.condition_label.pack(pady=5)

        self.condition_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Condition'].unique()))
        self.condition_combobox.pack(pady=5)

    self.condition_combobox.bind('<
```

```
columns=list(self.df.columns), show='headings')
        self.filter_tree.pack(expand=True,
fill='both')

        # Define columns
        for col in self.df.columns:
            self.filter_tree.heading(col,
text=col)
                self.filter_tree.column(col,
width=150, anchor='center')

        # Define tags for alternating row colors
        self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
        self.filter_tree.tag_configure('oddrow',
background='#ffffff')

        # Insert initial data
        self.update_filter_table(self.df)

    def create_graph_tab(self):
        self.graph_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.graph_frame,
text='Condition Distribution')

        # Create a Matplotlib figure and axis
        self.fig, self.ax = plt.subplots(figsize=
(8, 6))
        self.ax.set_title('Condition
Distribution')
        self.ax.set_xlabel('Condition')
        self.ax.set_ylabel('Frequency')

        # Create a canvas for Matplotlib
```

```

        self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)

    self.canvas.get_tk_widget().pack(expand=True,
fill='both')

    # Plot data
    self.update_plot()

    def update_table(self, data):
        for row in self.tree.get_children():
            self.tree.delete(row)
        for index, row in data.iterrows():
            tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_filter_table(self, data):
        for row in
self.filter_tree.get_children():
            self.filter_tree.delete(row)
        for index, row in data.iterrows():
            tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.filter_tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_plot(self):
        self.ax.clear()
        self.ax.set_title('Condition
Distribution')
        self.ax.set_xlabel('Condition')
        self.ax.set_ylabel('Frequency')

    # Plot condition distribution

```

```
        condition_counts =
self.df['Condition'].value_counts()
    self.ax.bar(condition_counts.index,
condition_counts.values, color='skyblue',
edgecolor='black')

    self.canvas.draw()

def apply_filters(self, event=None):
    patient_id = self.patient_combobox.get()
    condition = self.condition_combobox.get()

    filtered_df = self.df

    if patient_id:
        filtered_df =
filtered_df[filtered_df['PatientID'] ==
int(patient_id)]

    if condition:
        filtered_df =
filtered_df[filtered_df['Condition'] == condition]

    if filtered_df.empty:
        messagebox.showinfo("No Data", "No
data available for the selected filters.")
        filtered_df = self.df

    # Update the table in the Filters tab
    self.update_filter_table(filtered_df)

def reset_filters(self):
    self.patient_combobox.set('')
    self.condition_combobox.set('')
    self.update_filter_table(self.df)
```

```

def save_filtered_data(self):
    filtered_df = self.get_filtered_data()
    if filtered_df.empty:
        messagebox.showinfo("No Data", "No
data to save.")
    return
    file_path = 'filtered_medical_data.xlsx'
    filtered_df.to_excel(file_path,
index=False)
    messagebox.showinfo("Saved", f"Filtered
data has been saved to '{file_path}'")

def get_filtered_data(self):
    patient_id = self.patient_combobox.get()
    condition = self.condition_combobox.get()

    filtered_df = self.df

    if patient_id:
        filtered_df =
filtered_df[filtered_df['PatientID'] ==
int(patient_id)]

    if condition:
        filtered_df =
filtered_df[filtered_df['Condition'] == condition]

    return filtered_df

if __name__ == "__main__":
    app = MedicalApp(combined_df)
    app.mainloop()

```

Let's break down the code with detailed explanations of each part:

Imports and Data Generation

1. Imports:

```
import tkinter as tk
from tkinter import ttk, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from    matplotlib.backends.backend_tkagg    import
FigureCanvasTkAgg
```

- tkinter: A Python library used for creating graphical user interfaces (GUIs). tk is the main module.
- ttk: A module within tkinter that provides access to themed widgets, such as ttk.Combobox and ttk.Treeview.
- messagebox: A tkinter module that allows you to display standard dialogs, such as information and error messages.
- pandas: A powerful data manipulation library that provides data structures like DataFrames for working with structured data.
- numpy: A library for numerical operations, including generating random numbers and arrays.
- matplotlib: A plotting library used for creating static, interactive, and animated visualizations in Python.
- FigureCanvasTkAgg: A matplotlib class that integrates plots with tkinter widgets, allowing plots to be displayed within a tkinter window.

2. Generate Synthetic Data:

```
def generate_patient_data(num_records):
    np.random.seed(0)
    patient_ids = np.arange(1, num_records + 1)
```

```

        names = np.random.choice(['Alice Johnson',
'Bob Brown', 'Carol White', 'David Black', 'Eva
Green'], size=num_records)
        ages = np.random.randint(20, 80,
size=num_records)
        return pd.DataFrame({
            'PatientID': patient_ids,
            'Name': names,
            'Age': ages
        })

def generate_medical_records(num_records):
    np.random.seed(1)
    record_ids = np.arange(1, num_records + 1)
    patient_ids = np.random.choice(np.arange(1,
101), size=num_records) # Random patient IDs (may
have duplicates)
    conditions = np.random.choice(['Diabetes',
'Hypertension', 'Asthma', 'Cancer', 'Flu'],
size=num_records)
    return pd.DataFrame({
        'RecordID': record_ids,
        'PatientID': patient_ids,
        'Condition': conditions
    })

```

- generate_patient_data(num_records):
 - Purpose: Generates synthetic patient data.
 - Parameters: num_records - the number of patient records to generate.
 - Process:
 - Seed Setting: np.random.seed(0) ensures reproducibility.

- Patient IDs: Creates a sequence of unique IDs starting from 1.
- Names: Randomly selects names from a predefined list.
- Ages: Generates random ages between 20 and 80.
- Output: Returns a DataFrame with columns: PatientID, Name, and Age.
- generate_medical_records(num_records):
 - Purpose: Generates synthetic medical records.
 - Parameters: num_records - the number of medical records to generate.
- Process:
 - Seed Setting: np.random.seed(1) ensures reproducibility.
 - Record IDs: Creates a sequence of unique record IDs.
 - Patient IDs: Randomly selects patient IDs from a range, allowing for duplicates.
 - Conditions: Randomly selects medical conditions from a predefined list.
- Output: Returns a DataFrame with columns: RecordID, PatientID, and Condition.

3. Generate DataFrames and Merge:

```

num_patients = 100
num_records = 120
patient_df = generate_patient_data(num_patients)
medical_records_df =
generate_medical_records(num_records)
combined_df = pd.merge(patient_df,
medical_records_df, on='PatientID', how='outer')

```

- Purpose: Creates and merges the synthetic data for the application.
- Process:
 - Generate DataFrames:
 - patient_df: Contains synthetic patient data.
 - medical_records_df: Contains synthetic medical records.
 - Merge DataFrames:
 - combined_df: Performs an outer join on the PatientID column, ensuring all patient and record data is included. This will include all patients and records, filling missing values with NaN where no match is found.

Tkinter Application Class

4. Main Application Class:

```
class MedicalApp(tk.Tk):
    def __init__(self, df):
        super().__init__()
        self.title('Medical Data Viewer')
        self.geometry('1200x800')
        self.df = df
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(expand=True,
fill='both')
        self.create_table_tab()
        self.create_filter_tab()
        self.create_graph_tab()
```

- Purpose: Defines the main application window for viewing and interacting with medical data.

- Parameters:
 - df: The combined DataFrame to be used in the application.
- Process:
 - Initialization: Sets the title and size of the main window.
 - Notebook: Creates a tabbed interface using ttk.Notebook.
 - Tab Creation: Calls methods to create and configure tabs for displaying the table, applying filters, and showing graphs.

5. Table Tab:

```
def create_table_tab(self):
    self.table_frame = ttk.Frame(self.notebook)
    self.notebook.add(self.table_frame, text='DataTable')
    self.tree = ttk.Treeview(self.table_frame,
                           columns=list(self.df.columns), show='headings')
    self.tree.pack(expand=True, fill='both')
    for col in self.df.columns:
        self.tree.heading(col, text=col)
        self.tree.column(col, width=150,
                         anchor='center')
        self.tree.tag_configure('evenrow',
                               background='#f0f0f0')
        self.tree.tag_configure('oddrow',
                               background='#ffffff')
    self.update_table(self.df)
```

- Purpose: Creates a tab to display the full dataset in a table format.

- Components:
 - Frame: self.table_frame holds the table widget.
 - Treeview Widget: ttk.Treeview is used to display data in a tabular format.
 - Columns: Configured to show the columns from the DataFrame.
 - Headings: Set the column headings.
 - Column Widths: Adjusted to ensure readability.
 - Row Colors: Alternating row colors for better visibility.
 - Update Data: Calls self.update_table(self.df) to populate the table with the data.

6. Filter Tab:

```
def create_filter_tab(self):
    self.filter_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.filter_frame,
text='Filters')
                self.df['Condition'] =
self.df['Condition'].astype(str).fillna('Unknown')
                    self.patient_label =
ttk.Label(self.filter_frame,  text='Select Patient
ID:')
            self.patient_label.pack(pady=5)
                    self.patient_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['PatientID'].unique()))
            self.patient_combobox.pack(pady=5)

self.patient_combobox.bind('<>',
self.apply_filters)
```

```
                self.condition_label      =
ttk.Label(self.filter_frame,                  text='Select
Condition:')
    self.condition_label.pack(pady=5)
                self.condition_combobox     =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Condition'].unique())))
    self.condition_combobox.pack(pady=5)

self.condition_combobox.bind( '<<ComboboxSelected>>'
', self.apply_filters)
                self.reset_button      =
ttk.Button(self.filter_frame,                 text='Reset
Filters', command=self.reset_filters)
    self.reset_button.pack(pady=5)
                self.save_button      =
ttk.Button(self.filter_frame,   text='Save Filtered
Data', command=self.save_filtered_data)
    self.save_button.pack(pady=5)
                self.filter_tree      =
ttk.Treeview(self.filter_frame,
columns=list(self.df.columns), show='headings')
    self.filter_tree.pack(expand=True,
fill='both')
    for col in self.df.columns:
        self.filter_tree.heading(col, text=col)
        self.filter_tree.column(col, width=150,
anchor='center')
        self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
        self.filter_tree.tag_configure('oddrow',
background='#ffffff')
    self.update_filter_table(self.df)
```

- Purpose: Provides a tab for filtering data and displaying the filtered results.
- Components:
 - Frame: self.filter_frame holds all filtering widgets.
 - Condition Column Handling: Ensures Condition column values are strings and fills NaN values with 'Unknown'.
 - Filter Controls:
 - Labels and Comboboxes: Allow users to select filters for PatientID and Condition.
 - Buttons: For resetting filters and saving filtered data.
 - Treeview Widget: Displays the filtered data.
 - Event Binding: <<ComboboxSelected>> event triggers self.apply_filters() to update the filtered data.

7. Graph Tab:

```
def create_graph_tab(self):
    self.graph_frame = ttk.Frame(self.notebook)
    self.notebook.add(self.graph_frame,
text='Condition Distribution')
    self.fig, self.ax = plt.subplots(figsize=(8, 6))
    self.ax.set_title('Condition Distribution')
    self.ax.set_xlabel('Condition')
    self.ax.set_ylabel('Frequency')
    self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)
    self.canvas.get_tk_widget().pack(expand=True,
fill='both')
    self.update_plot()
```

- Purpose: Creates a tab to visualize the distribution of medical conditions.
- Components:
 - Frame: self.graph_frame holds the Matplotlib plot.
 - Figure and Axis: self.fig and self.ax are used to create and configure the plot.
 - Canvas: FigureCanvasTkAgg integrates the Matplotlib plot into the Tkinter interface.
 - Plot Initialization: Calls self.update_plot() to generate and display the initial plot.

8. Updating the Table:

```
def update_table(self, data):
    for row in self.tree.get_children():
        self.tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else 'oddrow'
                    self.tree.insert('', 'end',
values=list(row), tags=(tag,))
```

- Purpose: Refreshes the data displayed in the Treeview widget.
- Process:
 - Clear Existing Rows: Deletes all existing rows from the Treeview.
 - Insert New Rows: Adds rows from the DataFrame, with alternating colors for better readability.

9. Updating the Filter Table:

```

def update_filter_table(self, data):
    for row in self.filter_tree.get_children():
        self.filter_tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else 'oddrow'
        self.filter_tree.insert('', 'end',
values=list(row), tags=(tag,))

```

- Purpose: Updates the filtered data table in the Filters tab.
- Process:
 - Clear Existing Rows: Deletes all existing rows from the filter_tree.
 - Insert New Rows: Adds rows from the filtered DataFrame, with alternating row colors.

10. Updating the Plot:

```

def update_plot(self):
    self.ax.clear()
    self.ax.set_title('Condition Distribution')
    self.ax.set_xlabel('Condition')
    self.ax.set_ylabel('Frequency')
                condition_counts =
self.df['Condition'].value_counts()
                self.ax.bar(condition_counts.index,
condition_counts.values,           color='skyblue',
edgecolor='black')
    self.canvas.draw()

```

- Purpose: Updates the Matplotlib plot with the latest data.
- Process:
 - Clear Existing Plot: Clears previous plot data.

- Set Titles and Labels: Configures the plot's title and axis labels.
- Plot Data: Creates a bar chart of condition frequencies.
- Draw Canvas: Refreshes the canvas to display the updated plot.

11. Applying Filters:

```
def apply_filters(self, event=None):
    patient_id = self.patient_combobox.get()
    condition = self.condition_combobox.get()
    filtered_df = self.df
    if patient_id:
        filtered_df      =
    filtered_df[filtered_df['PatientID'] == int(patient_id)]
    if condition:
        filtered_df      =
    filtered_df[filtered_df['Condition'] == condition]
    if filtered_df.empty:
        messagebox.showinfo("No Data", "No data available for the selected filters.")
    filtered_df = self.df
    self.update_filter_table(filtered_df)
```

- Purpose: Filters the data based on user selections and updates the filtered table.
- Process:
 - Get Filter Values: Retrieves selected values from comboboxes.
 - Apply Filters: Filters the DataFrame based on selected PatientID and Condition.

- Handle Empty Results: Displays a message if no data matches the filters.
- Update Filter Table: Refreshes the filtered data display.

12. Resetting Filters:

```
def reset_filters(self):
    self.patient_combobox.set('')
    self.condition_combobox.set('')
    self.update_filter_table(self.df)
```

- Purpose: Resets the filter controls and shows the unfiltered data.
- Process:
 - Clear Selections: Resets the comboboxes to their default values.
 - Update Filter Table: Refreshes the table with the full dataset.

13. Saving Filtered Data:

```
def save_filtered_data(self):
    filtered_df = self.get_filtered_data()
    if filtered_df.empty:
        messagebox.showinfo("No Data", "No data to save.")
        return
    file_path = 'filtered_medical_data.xlsx'
    filtered_df.to_excel(file_path, index=False)
    messagebox.showinfo("Saved", f"Filtered data has been saved to '{file_path}'")
```

- Purpose: Saves the currently filtered data to an Excel file.
- Process:
 - Get Filtered Data: Retrieves the data based on current filter settings.
 - Check Data: Displays a message if there is no data to save.
 - Save to Excel: Writes the filtered DataFrame to an Excel file.
 - Show Confirmation: Displays a message indicating the data has been saved.

14. Getting Filtered Data:

```
def get_filtered_data(self):
    patient_id = self.patient_combobox.get()
    condition = self.condition_combobox.get()
    filtered_df = self.df
    if patient_id:
        filtered_df = filtered_df[filtered_df['PatientID'] == int(patient_id)]
    if condition:
        filtered_df = filtered_df[filtered_df['Condition'] == condition]
    return filtered_df
```

- Purpose: Returns the DataFrame filtered by the current filter settings.
- Process:
 - Get Filter Values: Retrieves selected values from comboboxes.
 - Apply Filters: Filters the DataFrame based on selected PatientID and Condition.

Application Execution

15. Main Block:

```
if __name__ == "__main__":
    app = MedicalApp(combined_df)
    app.mainloop()
```

- Purpose: Runs the application if the script is executed directly.
- Process:
 - Create Application: Instantiates the MedicalApp class with the combined DataFrame.
 - Start Mainloop: Enters the Tkinter event loop, making the application responsive to user actions.

Medical Data Viewer

Data Table | Filters | Condition Distribution

| PatientID | Name | Age | RecordID | Condition |
|-----------|---------------|-----|----------|--------------|
| 1 | Eva Green | 78 | 38.0 | Hypertension |
| 1 | Eva Green | 78 | 101.0 | Cancer |
| 2 | Alice Johnson | 43 | 10.0 | Hypertension |
| 2 | Alice Johnson | 43 | 37.0 | Diabetes |
| 3 | David Black | 79 | 107.0 | Cancer |
| 4 | David Black | 22 | 48.0 | Cancer |
| 4 | David Black | 22 | 53.0 | Cancer |
| 4 | David Black | 22 | 100.0 | Flu |
| 5 | David Black | 77 | nan | nan |
| 6 | Bob Brown | 54 | 6.0 | Diabetes |
| 7 | David Black | 55 | 13.0 | Flu |
| 7 | David Black | 55 | 103.0 | Cancer |
| 8 | Carol White | 50 | 32.0 | Asthma |
| 8 | Carol White | 50 | 69.0 | Asthma |
| 8 | Carol White | 50 | 112.0 | Flu |
| 9 | Eva Green | 79 | 41.0 | Flu |
| 9 | Eva Green | 79 | 81.0 | Diabetes |
| 10 | Alice Johnson | 23 | 4.0 | Flu |
| 10 | Alice Johnson | 23 | 31.0 | Asthma |
| 10 | Alice Johnson | 23 | 73.0 | Diabetes |
| 11 | Alice Johnson | 38 | 84.0 | Flu |
| 12 | Eva Green | 66 | 19.0 | Hypertension |
| 13 | Carol White | 55 | 2.0 | Hypertension |
| 14 | Bob Brown | 40 | 30.0 | Hypertension |
| 14 | Bob Brown | 40 | 43.0 | Diabetes |
| 15 | Alice Johnson | 37 | 22.0 | Cancer |
| 16 | Bob Brown | 70 | 63.0 | Cancer |
| 16 | Bob Brown | 70 | 86.0 | Diabetes |
| 17 | Bob Brown | 47 | 9.0 | Hypertension |
| 18 | Alice Johnson | 34 | nan | nan |

Medical Data Viewer

Data Table | Filters | Condition Distribution

Select Patient ID:

Select Condition:

| PatientID | Name | Age | RecordID | Condition |
|-----------|-------------|-----|----------|-----------|
| 4 | David Black | 22 | 48.0 | Cancer |
| 4 | David Black | 22 | 53.0 | Cancer |
| 4 | David Black | 22 | 100.0 | Flu |

Medical Data Viewer

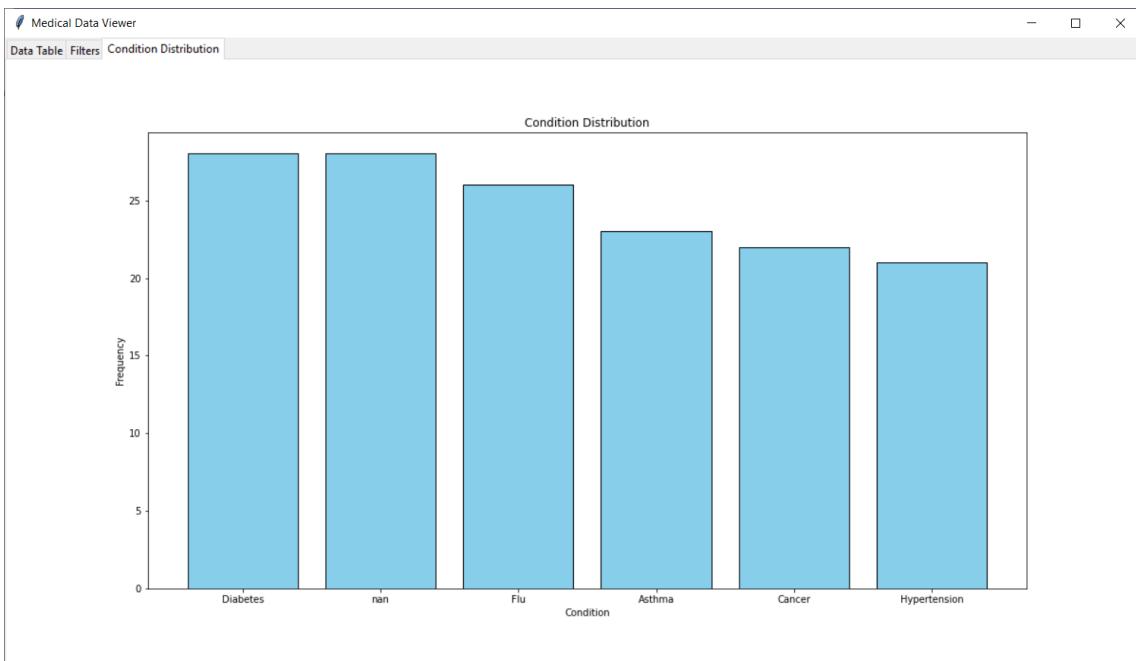
Data Table Filters Condition Distribution

Select Patient ID:
4

Select Condition:
Cancer

Reset Filters
Save Filtered Data

| PatientID | Name | Age | RecordID | Condition |
|-----------|-------------|-----|----------|-----------|
| 4 | David Black | 22 | 48.0 | Cancer |
| 4 | David Black | 22 | 53.0 | Cancer |



EXAMPLE 3.11

Performing Left Join on Dataframes with Synthetic Hospital Data

To demonstrate how a left join works and how it can result in NaN values for columns from the right DataFrame, let's generate synthetic hospital and doctor datasets, perform a left join on them, and then save the result to an Excel file.

Steps:

1. Generate Synthetic Data:

- hospitals_df: Contains hospital information such as Hospital ID, Name, and Location.
- doctors_df: Contains doctor information such as Doctor ID, Name, Specialization, and Hospital ID (as a foreign key).

2. Perform a Left Join:

We will merge the hospitals_df (left DataFrame) and doctors_df (right DataFrame) on the HospitalID column. This will include all rows from the hospitals_df, and rows from doctors_df where the HospitalID matches. If there's no match, the resulting columns from doctors_df will have NaN values.

3. Save to Excel:

Save the resulting DataFrame to an Excel file.

Python Code:

```
import pandas as pd
import numpy as np

# Function to generate hospital data
def generate_hospital_data(num_hospitals):
    np.random.seed(0)
    hospital_ids = np.arange(1, num_hospitals + 1)
```

```

        hospital_names = np.random.choice(['City
Hospital', 'Green Valley Medical', 'Sunrise
Clinic', 'Lakeside Healthcare', 'Mountainview
Health'], size=num_hospitals)
    locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_hospitals)
    return pd.DataFrame({
        'HospitalID': hospital_ids,
        'HospitalName': hospital_names,
        'Location': locations
    })
}

# Function to generate doctor data
def generate_doctor_data(num_doctors,
num_hospitals):
    np.random.seed(1)
    doctor_ids = np.arange(1, num_doctors + 1)
    doctor_names = np.random.choice(['Dr. Alice',
'Dr. Bob', 'Dr. Carol', 'Dr. David', 'Dr. Eva'],
size=num_doctors)
    specializations =
np.random.choice(['Cardiology', 'Neurology',
'Pediatrics', 'Oncology', 'Orthopedics'],
size=num_doctors)
    # Intentionally use fewer hospital IDs to
create NaN values when joined
    hospital_ids = np.random.choice(np.arange(1,
num_hospitals - 10), size=num_doctors) # some
hospitals won't match
    return pd.DataFrame({
        'DoctorID': doctor_ids,
        'DoctorName': doctor_names,
        'Specialization': specializations,
        'HospitalID': hospital_ids
    })

```

```

# Generate the datasets
num_hospitals = 50
num_doctors = 150
hospitals_df = generate_hospital_data(num_hospitals)
doctors_df = generate_doctor_data(num_doctors,
num_hospitals)

# Perform a left join on 'HospitalID'
combined_df = pd.merge(hospitals_df, doctors_df,
on='HospitalID', how='left')

# Save the result to an Excel file
file_path = 'hospital_doctor_left_join.xlsx'
combined_df.to_excel(file_path, index=False)

print(f"Data has been saved to '{file_path}'")

```

Explanation:

1. Hospital Dataset: hospitals_df contains HospitalID, HospitalName, and Location. Each hospital is identified by a unique HospitalID.
2. Doctor Dataset: doctors_df contains DoctorID, DoctorName, Specialization, and HospitalID. We deliberately generate fewer unique HospitalID values than exist in hospitals_df to ensure some hospitals won't have matching doctors, leading to NaN values when the datasets are merged.
3. Left Join:
 - The pd.merge() function performs the left join between hospitals_df and doctors_df on the HospitalID column.

- All rows from hospitals_df are included in the result. If a hospital in hospitals_df doesn't have a corresponding HospitalID in doctors_df, the columns from doctors_df will show NaN values.
4. Save to Excel: The combined_df DataFrame is saved to an Excel file named hospital_doctor_left_join.xlsx.

Result:

After running the code, you'll have an Excel file that shows the results of a left join operation. The file will include rows where some hospitals don't have corresponding doctors, resulting in NaN values in the doctor-related columns. This is a common scenario when merging datasets where not all entries have a match in both tables.

EXAMPLE 3.12

GUI Tkinter for Performing Left Join on Dataframes with Synthetic Hospital Data

This code creates a graphical user interface (GUI) application using Tkinter to manage and analyze synthetic data for hospitals and doctors. The application first generates two datasets: one representing hospitals with attributes like HospitalID, HospitalName, and Location, and another representing doctors with attributes like DoctorID, DoctorName, Specialization, and the hospital they are associated with (HospitalID). These datasets are then merged using a left join, ensuring that all hospital records are preserved, even if some hospitals have no associated doctors. The merged dataset is saved to an Excel file for further use.

The GUI is designed with three main tabs: a data table, filters, and a graph. The data table tab allows users to view the entire merged dataset in a tabular format, with alternating row colors for better readability. The filters tab enables users to filter the data based on HospitalID and Location using dropdown menus, displaying the filtered results in a separate table. Users can reset the filters or save the filtered data to a new Excel file. The graph tab provides a visual representation of the distribution of doctor specializations across hospitals, using a bar chart that is dynamically updated based on the data.

Overall, the purpose of this code is to provide a user-friendly interface for exploring and analyzing synthetic hospital and doctor data. It allows users to interact with the data by viewing it in different formats, applying filters, and visualizing trends, making it a versatile tool for data analysis in a simulated healthcare environment.

```
import tkinter as tk
from tkinter import ttk, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Generate synthetic hospital data
def generate_hospital_data(num_hospitals):
    np.random.seed(0)
    hospital_ids = np.arange(1, num_hospitals + 1)
    hospital_names = np.random.choice(['City
Hospital', 'Green Valley Medical', 'Sunrise
Clinic', 'Lakeside Healthcare', 'Mountainview
Health'], size=num_hospitals)
```

```
locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_hospitals)
return pd.DataFrame({
    'HospitalID': hospital_ids,
    'HospitalName': hospital_names,
    'Location': locations
})

# Generate synthetic doctor data
def generate_doctor_data(num_doctors,
num_hospitals):
    np.random.seed(1)
    doctor_ids = np.arange(1, num_doctors + 1)
    doctor_names = np.random.choice(['Dr. Alice',
'Dr. Bob', 'Dr. Carol', 'Dr. David', 'Dr. Eva'],
size=num_doctors)
    specializations =
np.random.choice(['Cardiology', 'Neurology',
'Pediatrics', 'Oncology', 'Orthopedics'],
size=num_doctors)
    hospital_ids = np.random.choice(np.arange(1,
num_hospitals - 10), size=num_doctors) # some
hospitals won't match
return pd.DataFrame({
    'DoctorID': doctor_ids,
    'DoctorName': doctor_names,
    'Specialization': specializations,
    'HospitalID': hospital_ids
})

# Generate the datasets
num_hospitals = 100
num_doctors = 200
hospitals_df =
generate_hospital_data(num_hospitals)
```

```
doctors_df = generate_doctor_data(num_doctors,
num_hospitals)

# Perform a left join on 'HospitalID'
combined_df = pd.merge(hospitals_df, doctors_df,
on='HospitalID', how='left')

# Save the result to an Excel file
file_path = 'hospital_doctor_left_join.xlsx'
combined_df.to_excel(file_path, index=False)

class HospitalApp(tk.Tk):
    def __init__(self, df):
        super().__init__()
        self.title('Hospital and Doctor Data
Viewer')
        self.geometry('1200x800')
        self.df = df

        # Create a Notebook (tabbed interface)
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(expand=True,
fill='both')

        # Create frames for each tab
        self.create_table_tab()
        self.create_filter_tab()
        self.create_graph_tab()

    def create_table_tab(self):
        self.table_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.table_frame,
text='Data Table')

        # Create a Treeview for the table
```

```
        self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
        self.tree.pack(expand=True, fill='both')

        # Define columns
        for col in self.df.columns:
            self.tree.heading(col, text=col)
            self.tree.column(col, width=150,
anchor='center')

        # Define tags for alternating row colors
        self.tree.tag_configure('evenrow',
background='#f0f0f0')
        self.tree.tag_configure('oddrow',
background='#ffffff')

        # Insert data
        self.update_table(self.df)

    def create_filter_tab(self):
        self.filter_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.filter_frame,
text='Filters')

        # Create filters for Hospital ID and
Location
        self.hospital_label =
ttk.Label(self.filter_frame, text='Select Hospital
ID:')
        self.hospital_label.pack(pady=5)

        self.hospital_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['HospitalID'].unique()))
        self.hospital_combobox.pack(pady=5)
```

```
    self.hospital_combobox.bind('<>'
', self.apply_filters)

        self.location_label =
ttk.Label(self.filter_frame, text='Select
Location:')
        self.location_label.pack(pady=5)

        self.location_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Location'].unique()))
        self.location_combobox.pack(pady=5)

    self.location_combobox.bind('<>'
', self.apply_filters)

        self.reset_button =
ttk.Button(self.filter_frame, text='Reset
Filters', command=self.reset_filters)
        self.reset_button.pack(pady=5)

        self.save_button =
ttk.Button(self.filter_frame, text='Save Filtered
Data', command=self.save_filtered_data)
        self.save_button.pack(pady=5)

    # Create a Treeview for filtered data in
the Filters tab
        self.filter_tree =
ttk.Treeview(self.filter_frame,
columns=list(self.df.columns), show='headings')
        self.filter_tree.pack(expand=True,
fill='both')

    # Define columns
```

```
        for col in self.df.columns:
            self.filter_tree.heading(col,
text=col)
                self.filter_tree.column(col,
width=150, anchor='center')

        # Define tags for alternating row colors
        self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
        self.filter_tree.tag_configure('oddrow',
background='#ffffff')

        # Insert initial data
        self.update_filter_table(self.df)

    def create_graph_tab(self):
        self.graph_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.graph_frame,
text='Doctor Specialization Distribution')

        # Create a Matplotlib figure and axis
        self.fig, self.ax = plt.subplots(figsize=
(8, 6))
        self.ax.set_title('Doctor Specialization
Distribution')
        self.ax.set_xlabel('Specialization')
        self.ax.set_ylabel('Number of Doctors')

        # Create a canvas for Matplotlib
        self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)

        self.canvas.get_tk_widget().pack(expand=True,
fill='both')
```

```

        # Plot data
        self.update_plot()

    def update_table(self, data):
        for row in self.tree.get_children():
            self.tree.delete(row)
        for index, row in data.iterrows():
            tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_filter_table(self, data):
        for row in
self.filter_tree.get_children():
            self.filter_tree.delete(row)
        for index, row in data.iterrows():
            tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.filter_tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_plot(self):
        self.ax.clear()
        self.ax.set_title('Doctor Specialization
Distribution')
        self.ax.set_xlabel('Specialization')
        self.ax.set_ylabel('Number of Doctors')

        # Plot specialization distribution
        specialization_counts =
self.df['Specialization'].value_counts()
        self.ax.bar(specialization_counts.index,
specialization_counts.values, color='skyblue',
edgecolor='black')

```

```
    self.canvas.draw()

def apply_filters(self, event=None):
    hospital_id = self.hospital_combobox.get()
    location = self.location_combobox.get()

    filtered_df = self.df

    if hospital_id:
        filtered_df =
    filtered_df[filtered_df['HospitalID'] ==
    int(hospital_id)]

    if location:
        filtered_df =
    filtered_df[filtered_df['Location'] == location]

    if filtered_df.empty:
        messagebox.showinfo("No Data", "No
data available for the selected filters.")
        filtered_df = self.df

    # Update the table in the Filters tab
    self.update_filter_table(filtered_df)

def reset_filters(self):
    self.hospital_combobox.set('')
    self.location_combobox.set('')
    self.update_filter_table(self.df)

def save_filtered_data(self):
    filtered_df = self.get_filtered_data()
    if filtered_df.empty:
        messagebox.showinfo("No Data", "No
data to save.")
    return
```

```

        file_path = 'filtered_hospital_data.xlsx'
        filtered_df.to_excel(file_path,
index=False)
        messagebox.showinfo("Saved", f"Filtered
data has been saved to '{file_path}'")

def get_filtered_data(self):
    hospital_id = self.hospital_combobox.get()
    location = self.location_combobox.get()

    filtered_df = self.df

    if hospital_id:
        filtered_df =
filtered_df[filtered_df['HospitalID'] ==
int(hospital_id)]

    if location:
        filtered_df =
filtered_df[filtered_df['Location'] == location]

    return filtered_df

if __name__ == "__main__":
    app = HospitalApp(combined_df)
    app.mainloop()

```

Below is a detailed explanation of each part of the code:

1. Importing Necessary Libraries

```

import tkinter as tk
from tkinter import ttk, messagebox
import pandas as pd
import numpy as np

```

```
import matplotlib.pyplot as plt
from    matplotlib.backends.backend_tkagg      import
FigureCanvasTkAgg
```

- tkinter: The standard Python library for creating GUI applications.
- ttk: A themed widget set that provides a more modern look than the standard Tkinter widgets.
- messagebox: Provides simple message boxes for user interaction.
- pandas: A powerful data manipulation and analysis library.
- numpy: A library for numerical operations, used here to generate random data.
- matplotlib: A plotting library used to create visualizations in the GUI.
- FigureCanvasTkAgg: A helper class that integrates Matplotlib plots into Tkinter applications.

2. Generating Synthetic Hospital Data

```
def generate_hospital_data(num_hospitals):
    np.random.seed(0)
    hospital_ids = np.arange(1, num_hospitals + 1)
    hospital_names = np.random.choice(['City
Hospital', 'Green Valley Medical', 'Sunrise
Clinic', 'Lakeside Healthcare', 'Mountainview
Health'], size=num_hospitals)
    locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_hospitals)
    return pd.DataFrame({
        'HospitalID': hospital_ids,
```

```

        'HospitalName': hospital_names,
        'Location': locations
    })

```

- `generate_hospital_data(num_hospitals)`: This function creates a synthetic dataset for hospitals.
 - `np.random.seed(0)`: Ensures reproducibility by setting the random number generator's seed.
 - `hospital_ids`: Generates unique IDs for hospitals.
 - `hospital_names` and `locations`: Randomly assigns names and locations to hospitals.
 - Returns a DataFrame with columns: HospitalID, HospitalName, and Location.

3. Generating Synthetic Doctor Data

```

def generate_doctor_data(num_doctors,
                        num_hospitals):
    np.random.seed(1)
    doctor_ids = np.arange(1, num_doctors + 1)
    doctor_names = np.random.choice(['Dr. Alice',
                                    'Dr. Bob', 'Dr. Carol', 'Dr. David', 'Dr. Eva'],
                                    size=num_doctors)
    specializations = np.random.choice(['Cardiology', 'Neurology',
                                        'Pediatrics', 'Oncology', 'Orthopedics'],
                                        size=num_doctors)
    hospital_ids = np.random.choice(np.arange(1, num_hospitals - 10), size=num_doctors) # some
    # hospitals won't match
    return pd.DataFrame({
        'DoctorID': doctor_ids,
        'DoctorName': doctor_names,
        'Specialization': specializations,
    })

```

```
        'HospitalID': hospital_ids
    })
```

- `generate_doctor_data(num_doctors, num_hospitals)`: This function creates a synthetic dataset for doctors.
 - Similar to the hospital data, but with additional columns for DoctorID, DoctorName, and Specialization.
 - `hospital_ids`: Randomly assigns doctors to hospitals. Some doctors may not match a hospital, demonstrating how NaN values appear after a left join.

4. Generating and Merging Datasets

```
num_hospitals = 100
num_doctors = 200
hospitals_df = generate_hospital_data(num_hospitals)
doctors_df = generate_doctor_data(num_doctors,
num_hospitals)

combined_df = pd.merge(hospitals_df, doctors_df,
on='HospitalID', how='left')
```

- `hospitals_df` and `doctors_df`: DataFrames containing the synthetic hospital and doctor data, respectively.
- `combined_df`: Merges the two datasets on HospitalID using a left join. This ensures that all hospitals appear in the final DataFrame, with NaN values for doctors if there's no match.

5. Saving the Combined Data to an Excel File

```
file_path = 'hospital_doctor_left_join.xlsx'  
combined_df.to_excel(file_path, index=False)
```

- `to_excel(file_path, index=False)`: Saves the combined DataFrame to an Excel file without including the index column.

6. Creating the Tkinter GUI

```
class HospitalApp(tk.Tk):  
    def __init__(self, df):  
        super().__init__()  
        self.title('Hospital and Doctor Data  
Viewer')  
        self.geometry('1200x800')  
        self.df = df  
        self.notebook = ttk.Notebook(self)  
        self.notebook.pack(expand=True,  
fill='both')  
  
        self.create_table_tab()  
        self.create_filter_tab()  
        self.create_graph_tab()
```

- `HospitalApp`: A class that inherits from `tk.Tk`, creating the main window of the application.
- `init(self, df)`: Initializes the GUI, setting the window title, size, and storing the DataFrame (`df`).
- `ttk.Notebook(self)`: Creates a tabbed interface where different views (data table, filters, graphs) are added.

7. Creating the Data Table Tab

```
def create_table_tab(self):
    self.table_frame = ttk.Frame(self.notebook)
    self.notebook.add(self.table_frame, text='Data
Table')

    self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
    self.tree.pack(expand=True, fill='both')

    for col in self.df.columns:
        self.tree.heading(col, text=col)
        self.tree.column(col, width=150,
anchor='center')

        self.tree.tag_configure('evenrow',
background='#f0f0f0')
        self.tree.tag_configure('oddrow',
background='#ffffff')

    self.update_table(self.df)
```

- `create_table_tab(self)`: Sets up the "Data Table" tab.
 - `ttk.Frame(self.notebook)`: Creates a new frame within the notebook.
 - `ttk.Treeview(self.table_frame, columns=...)`: Displays the DataFrame as a table in the GUI.
 - `tree.heading(col, text=col)` and `tree.column(col, ...)`: Set the column headings and properties.
 - `tree.tag_configure`: Configures alternating row colors for better readability.

8. Creating the Filters Tab

```
def create_filter_tab(self):
    self.filter_frame = ttk.Frame(self.notebook)
    self.notebook.add(self.filter_frame,
text='Filters')

        self.hospital_label      =
ttk.Label(self.filter_frame, text='Select Hospital
ID:')
    self.hospital_label.pack(pady=5)

        self.hospital_combobox     =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['HospitalID'].unique()))
    self.hospital_combobox.pack(pady=5)

self.hospital_combobox.bind('<<ComboboxSelected>>'
, self.apply_filters)

        self.location_label       =
ttk.Label(self.filter_frame,           text='Select
Location:')
    self.location_label.pack(pady=5)

        self.location_combobox    =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Location'].unique()))
    self.location_combobox.pack(pady=5)

self.location_combobox.bind('<<ComboboxSelected>>'
, self.apply_filters)

        self.reset_button         =
ttk.Button(self.filter_frame,           text='Reset
Filters', command=self.reset_filters)
    self.reset_button.pack(pady=5)
```

```

                self.save_button      =
ttk.Button(self.filter_frame,  text='Save Filtered
Data', command=self.save_filtered_data)
self.save_button.pack(pady=5)

                self.filter_tree      =
ttk.Treeview(self.filter_frame,
columns=list(self.df.columns), show='headings')
self.filter_tree.pack(expand=True,
fill='both')

for col in self.df.columns:
    self.filter_tree.heading(col, text=col)
    self.filter_tree.column(col, width=150,
anchor='center')

    self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
    self.filter_tree.tag_configure('oddrow',
background='#ffffff')

self.update_filter_table(self.df)

```

- `create_filter_tab(self)`: Sets up the "Filters" tab.
 - `ttk.Combobox`: Dropdown menus allow users to filter data by HospitalID and Location.
 - `apply_filters`: Called when a filter is selected, updating the displayed data.
 - `reset_button`: Resets all filters.
 - `save_button`: Saves the filtered data to an Excel file.
 - `filter_tree`: Displays the filtered data in a table.

9. Creating the Graph Tab

```

def create_graph_tab(self):
    self.graph_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.graph_frame,
text='Doctor Specialization Distribution')

    self.fig, self.ax = plt.subplots(figsize=(8,
6))
        self.ax.set_title('Doctor Specialization
Distribution')
    self.ax.set_xlabel('Specialization')
    self.ax.set_ylabel('Number of Doctors')

    self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)
    self.canvas.get_tk_widget().pack(expand=True,
fill='both')

    self.update_plot()

```

- `create_graph_tab(self)`: Sets up the "Graph" tab to display a bar chart of doctor specializations.
 - `plt.subplots(figsize=(8, 6))`: Creates a Matplotlib figure and axis for plotting.
 - `FigureCanvasTkAgg(self.fig, master=self.graph_frame)`: Embeds the plot into the Tkinter window.

10. Helper Functions to Update GUI Components

```

def update_table(self, data):
    for row in self.tree.get_children():
        self.tree.delete(row)
    for index, row in data.iterrows():

```

```

        tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.tree.insert('',    'end',
values=list(row), tags=(tag,))

def update_filter_table(self, data):
    for row in self.filter_tree.get_children():
        self.filter_tree.delete(row)
    for index, row in data.iterrows():
        tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.filter_tree.insert('',    'end',
values=list(row), tags=(tag,))

def update_plot(self):
    self.ax.clear()
        self.ax.set_title('Doctor Specialization
Distribution')
    self.ax.set_xlabel('Specialization')
    self.ax.set_ylabel('Number of Doctors')

        specialization_counts      =
self.df['Specialization'].value_counts()
        self.ax.bar(specialization_counts.index,
specialization_counts.values,      color='skyblue',
edgecolor='black')

    self.canvas.draw()

```

- `update_table` and `update_filter_table`: Update the data displayed in the tables.
- `update_plot`: Updates the bar chart with the latest data.

11. Running the Application

```

if __name__ == "__main__":
    app = HospitalApp(combined_df)
    app.mainloop()

```

- if name == "main": This ensures the code only runs if the script is executed directly.
- app = HospitalApp(combined_df): Creates an instance of the HospitalApp class.
- app.mainloop(): Starts the Tkinter event loop, displaying the GUI.

This code provides a comprehensive GUI for exploring, filtering, and visualizing the synthetic hospital and doctor datasets. The interface is designed to be user-friendly and offers multiple ways to interact with the data.

Hospital and Doctor Data Viewer

| Data Table | | Doctor Specialization Distribution | | | |
|------------|----------------------|------------------------------------|----------|------------|----------------|
| HospitalID | HospitalName | Location | DoctorID | DoctorName | Specialization |
| 1 | Mountainview Health | Chicago | 143.0 | Dr. Eva | Cardiology |
| 2 | City Hospital | Houston | 150.0 | Dr. Carol | Orthopedics |
| 2 | City Hospital | Houston | 183.0 | Dr. Bob | Pediatrics |
| 2 | City Hospital | Houston | 191.0 | Dr. Eva | Neurology |
| 3 | Lakeside Healthcare | Chicago | 11.0 | Dr. Bob | Pediatrics |
| 4 | Lakeside Healthcare | Los Angeles | 8.0 | Dr. Bob | Oncology |
| 4 | Lakeside Healthcare | Los Angeles | 62.0 | Dr. Eva | Oncology |
| 5 | Lakeside Healthcare | Chicago | 3.0 | Dr. Alice | Orthopedics |
| 5 | Lakeside Healthcare | Chicago | 33.0 | Dr. Alice | Cardiology |
| 5 | Lakeside Healthcare | Chicago | 174.0 | Dr. Carol | Cardiology |
| 6 | Green Valley Medical | Houston | 20.0 | Dr. Carol | Orthopedics |
| 6 | Green Valley Medical | Houston | 44.0 | Dr. Bob | Oncology |
| 6 | Green Valley Medical | Houston | 48.0 | Dr. Eva | Orthopedics |
| 6 | Green Valley Medical | Houston | 102.0 | Dr. Alice | Pediatrics |
| 6 | Green Valley Medical | Houston | 175.0 | Dr. Eva | Cardiology |
| 7 | Lakeside Healthcare | Houston | nan | nan | nan |
| 8 | Sunrise Clinic | Houston | 49.0 | Dr. Alice | Neurology |
| 8 | Sunrise Clinic | Houston | 92.0 | Dr. Eva | Orthopedics |
| 8 | Sunrise Clinic | Houston | 168.0 | Dr. Carol | Cardiology |
| 8 | Sunrise Clinic | Houston | 169.0 | Dr. Eva | Oncology |
| 8 | Sunrise Clinic | Houston | 172.0 | Dr. Alice | Cardiology |
| 9 | Mountainview Health | Chicago | 45.0 | Dr. David | Cardiology |
| 10 | City Hospital | Houston | 60.0 | Dr. Bob | Pediatrics |
| 11 | City Hospital | Phoenix | 75.0 | Dr. Alice | Oncology |
| 12 | Mountainview Health | Los Angeles | 128.0 | Dr. Alice | Pediatrics |
| 12 | Mountainview Health | Los Angeles | 160.0 | Dr. Carol | Orthopedics |
| 12 | Mountainview Health | Los Angeles | 161.0 | Dr. David | Pediatrics |
| 13 | Sunrise Clinic | Chicago | 10.0 | Dr. Eva | Cardiology |
| 13 | Sunrise Clinic | Chicago | 16.0 | Dr. David | Neurology |
| 13 | Sunrise Clinic | Chicago | 89.0 | Dr. David | Orthopedics |

Hospital and Doctor Data Viewer

Data Table | Filters | Doctor Specialization Distribution

Select Hospital ID:

Select Location:

| HospitalID | HospitalName | Location | DoctorID | DoctorName | Specialization |
|------------|----------------------|-------------|----------|------------|----------------|
| 1 | Mountainview Health | Chicago | 143.0 | Dr. Eva | Cardiology |
| 2 | City Hospital | Houston | 150.0 | Dr. Carol | Orthopedics |
| 2 | City Hospital | Houston | 183.0 | Dr. Bob | Pediatrics |
| 2 | City Hospital | Houston | 191.0 | Dr. Eva | Neurology |
| 3 | Lakeside Healthcare | Chicago | 11.0 | Dr. Bob | Pediatrics |
| 4 | Lakeside Healthcare | Los Angeles | 8.0 | Dr. Bob | Oncology |
| 4 | Lakeside Healthcare | Los Angeles | 62.0 | Dr. Eva | Oncology |
| 5 | Lakeside Healthcare | Chicago | 3.0 | Dr. Alice | Orthopedics |
| 5 | Lakeside Healthcare | Chicago | 33.0 | Dr. Alice | Cardiology |
| 5 | Lakeside Healthcare | Chicago | 174.0 | Dr. Carol | Cardiology |
| 6 | Green Valley Medical | Houston | 20.0 | Dr. Carol | Orthopedics |
| 6 | Green Valley Medical | Houston | 44.0 | Dr. Bob | Oncology |
| 6 | Green Valley Medical | Houston | 48.0 | Dr. Eva | Orthopedics |
| 6 | Green Valley Medical | Houston | 102.0 | Dr. Alice | Pediatrics |
| 6 | Green Valley Medical | Houston | 175.0 | Dr. Eva | Cardiology |
| 7 | Lakeside Healthcare | Houston | nan | nan | nan |
| 8 | Sunrise Clinic | Houston | 49.0 | Dr. Alice | Neurology |
| 8 | Sunrise Clinic | Houston | 92.0 | Dr. Eva | Orthopedics |
| 8 | Sunrise Clinic | Houston | 168.0 | Dr. Carol | Cardiology |
| 8 | Sunrise Clinic | Houston | 169.0 | Dr. Eva | Oncology |
| 8 | Sunrise Clinic | Houston | 172.0 | Dr. Alice | Cardiology |

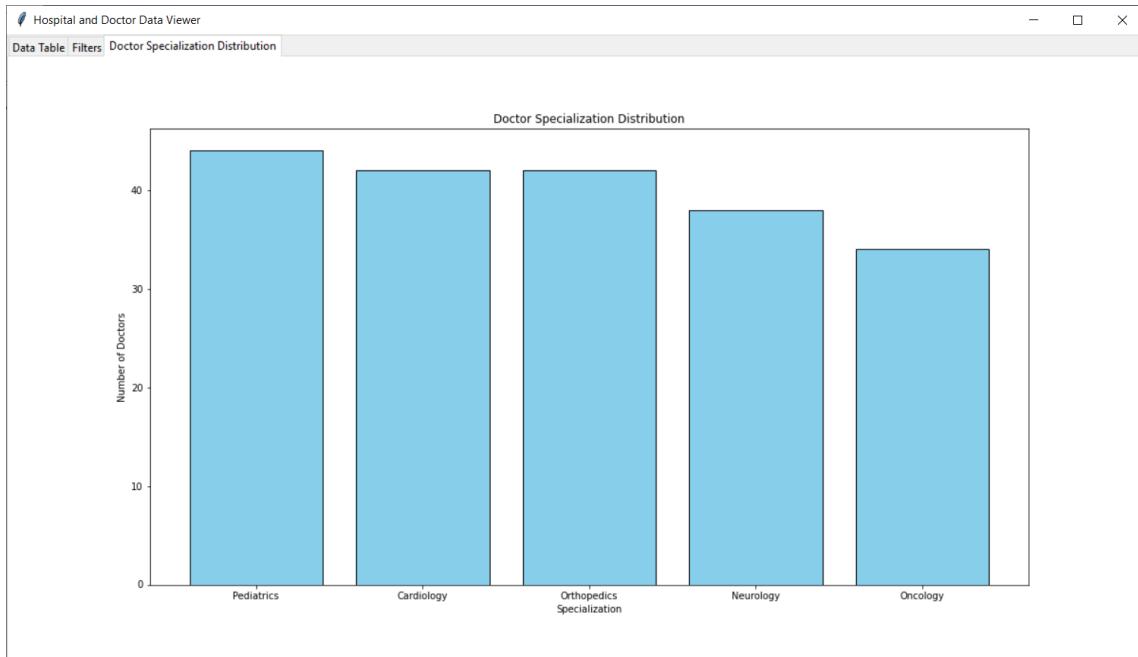
Hospital and Doctor Data Viewer

Data Table | Filters | Doctor Specialization Distribution

Select Hospital ID:

Select Location:

| HospitalID | HospitalName | Location | DoctorID | DoctorName | Specialization |
|------------|----------------|----------|----------|------------|----------------|
| 8 | Sunrise Clinic | Houston | 49.0 | Dr. Alice | Neurology |
| 8 | Sunrise Clinic | Houston | 92.0 | Dr. Eva | Orthopedics |
| 8 | Sunrise Clinic | Houston | 168.0 | Dr. Carol | Cardiology |
| 8 | Sunrise Clinic | Houston | 169.0 | Dr. Eva | Oncology |
| 8 | Sunrise Clinic | Houston | 172.0 | Dr. Alice | Cardiology |



EXAMPLE 3.13

Performing Right Join on Dataframes with Synthetic University Data

Here's how you can perform a right join on two synthetic DataFrames, generate a university dataset, and save the resulting data (including demonstration of NaN values) to an Excel file.

Step-by-Step Implementation

1. Generate the Synthetic Datasets:

- University Data: This will contain information about different universities with attributes like UniversityID, UniversityName, and Location.
- Student Data: This will contain information about students with attributes like StudentID, StudentName, Major, and UniversityID.

2. Perform a Right Join:

- We'll join the Student DataFrame with the University DataFrame using UniversityID as the key.
- Since it's a right join, all rows from the Student DataFrame will be retained, and matching rows from the University DataFrame will be included. If a student's UniversityID doesn't exist in the University DataFrame, the corresponding university fields will show NaN.

3. Save to Excel:

The resulting DataFrame will be saved as an Excel file, demonstrating the NaN values where there were mismatches in the UniversityID between the two DataFrames.

Python Code

```
import pandas as pd
import numpy as np

# Generate synthetic university data
def generate_university_data(num_universities):
    np.random.seed(0)
    university_ids = np.arange(1, num_universities
+ 1)
    university_names = np.random.choice(['Tech
University', 'Liberal Arts College', 'Medical
School', 'Business Institute', 'Engineering
Academy'], size=num_universities)
    locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_universities)
    return pd.DataFrame({
        'UniversityID': university_ids,
```

```
'UniversityName': university_names,
'Location': locations
})

# Generate synthetic student data
def generate_student_data(num_students,
num_universities):
    np.random.seed(1)
    student_ids = np.arange(1, num_students + 1)
    student_names = np.random.choice(['John Doe',
'Jane Smith', 'Emily Davis', 'Michael Brown',
'Sarah Wilson'], size=num_students)
    majors = np.random.choice(['Computer Science',
'Biology', 'Business', 'Engineering', 'History'],
size=num_students)
    university_ids = np.random.choice(np.arange(1,
num_universities + 100), size=num_students) # some students will have no matching university
    return pd.DataFrame({
        'StudentID': student_ids,
        'StudentName': student_names,
        'Major': majors,
        'UniversityID': university_ids
```

```
})

# Generate the datasets
num_universities = 100
num_students = 500
universities_df =
generate_university_data(num_universities)
students_df = generate_student_data(num_students,
num_universities)

# Perform a right join on 'UniversityID'
combined_df = pd.merge(students_df,
universities_df, on='UniversityID', how='right')

# Save the result to an Excel file
file_path = 'university_student_right_join.xlsx'
combined_df.to_excel(file_path, index=False)

print(f"Right join DataFrame saved to
{file_path}")
```

Explanation

- University DataFrame (universities_df): This contains 10 universities with unique UniversityID, their names, and locations.
- Student DataFrame (students_df): This contains 15 students. Some of the students have UniversityIDs that do not match any university in the universities_df, which will result in NaN values for the UniversityName and Location columns after the right join.
- Right Join: The pd.merge() function with how='right' ensures that all entries in the students_df are preserved, and any corresponding data from universities_df is filled

in. Where there is no matching university, the UniversityName and Location fields will be NaN.

- Excel File: The resulting combined_df will be saved to an Excel file named university_student_right_join.xlsx, containing all students, with NaN values for universities that do not match.

This code gives a clear demonstration of how NaN values appear in a right join operation when there are missing matches in the join key between the two DataFrames.

EXAMPLE 3.14

GUI Tkinter for Performing Right Join on Dataframes with Synthetic University Data

The purpose of this code is to create an interactive Tkinter application for visualizing and analyzing synthetic university and student data. The application uses data generated through synthetic data generation functions to populate a variety of views within a graphical user interface (GUI). This includes displaying a comprehensive data table, filtering the data based on user-selected criteria, and providing visual insights through charts and graphs.

In addition to presenting the data in a tabular format, the application allows users to apply filters based on university IDs and majors, which updates a filtered data table within the GUI. The filtering functionality helps users to drill down into specific subsets of the data, making it easier to analyze trends or particular attributes. Additionally, the application features buttons for resetting filters and saving the filtered data to an Excel file, which provides flexibility in managing and exporting the data.

The visual aspects of the application are handled through embedded Matplotlib charts that display student distributions by major, university, and location. This graphical representation helps users to quickly grasp the distribution patterns and relative sizes within the data. The combination of data tables and graphical visualizations provides a robust tool for exploring and understanding the synthetic data, making it a useful resource for both analysis and presentation.

```
import tkinter as tk
from tkinter import ttk, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Generate synthetic university data
def generate_university_data(num_universities):
    np.random.seed(0)
    university_ids = np.arange(1, num_universities
+ 1)
    university_names = np.random.choice(['Tech
University', 'Liberal Arts College', 'Medical
School', 'Business Institute', 'Engineering
Academy'], size=num_universities)
    locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_universities)
    return pd.DataFrame({
        'UniversityID': university_ids,
        'UniversityName': university_names,
        'Location': locations
    })
```

```
# Generate synthetic student data
def generate_student_data(num_students,
num_universities):
    np.random.seed(1)
    student_ids = np.arange(1, num_students + 1)
    student_names = np.random.choice(['John Doe',
'Jane Smith', 'Emily Davis', 'Michael Brown',
'Sarah Wilson'], size=num_students)
    majors = np.random.choice(['Computer Science',
'Biology', 'Business', 'Engineering', 'History'],
size=num_students)
    university_ids = np.random.choice(np.arange(1,
num_universities + 100), size=num_students) # Ensure IDs are within range
    locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_students) # Ensure locations match
universities
    return pd.DataFrame({
        'StudentID': student_ids,
        'StudentName': student_names,
        'Major': majors,
        'UniversityID': university_ids,
        'Location': locations
    })

# Generate the datasets
num_universities = 100
num_students = 5000
universities_df =
generate_university_data(num_universities)
students_df = generate_student_data(num_students,
num_universities)

# Perform a right join on 'UniversityID'
```

```
combined_df = pd.merge(students_df,
universities_df, on='UniversityID', how='right')

# Save the result to an Excel file
file_path = 'university_student_right_join.xlsx'
combined_df.to_excel(file_path, index=False)

# Tkinter Application
class UniversityApp(tk.Tk):
    def __init__(self, df):
        super().__init__()
        self.title('University and Student Data
Viewer')
        self.geometry('1200x900') # Increased
height to accommodate additional tabs
        self.df = df

    # Print columns for debugging
    print(self.df.columns)

    # Create a Notebook (tabbed interface)
    self.notebook = ttk.Notebook(self)
    self.notebook.pack(expand=True,
fill='both')

    # Create frames for each tab
    self.create_table_tab()
    self.create_filter_tab()
    self.create_graph_tab()
    self.create_university_tab()
    self.create_location_tab()

    def create_table_tab(self):
        self.table_frame =
ttk.Frame(self.notebook)
```

```
        self.notebook.add(self.table_frame,
text='Data Table')

        # Create a Treeview for the table
        self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
        self.tree.pack(expand=True, fill='both')

        # Define columns
        for col in self.df.columns:
            self.tree.heading(col, text=col)
            self.tree.column(col, width=150,
anchor='center')

        # Define tags for alternating row colors
        self.tree.tag_configure('evenrow',
background='#f0f0f0')
        self.tree.tag_configure('oddrow',
background='#ffffff')

        # Insert data
        self.update_table(self.df)

    def create_filter_tab(self):
        self.filter_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.filter_frame,
text='Filters')

        # Create filters for University ID and
Major
        self.university_label =
ttk.Label(self.filter_frame, text='Select
University ID:')
        self.university_label.pack(pady=5)
```

```
        self.university_combobox =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['UniversityID'].dropna().unique()))
        self.university_combobox.pack(pady=5)

    self.university_combobox.bind('<
```

```
        self.filter_tree.pack(expand=True,
fill='both')

        # Define columns
        for col in self.df.columns:
            self.filter_tree.heading(col,
text=col)
            self.filter_tree.column(col,
width=150, anchor='center')

        # Define tags for alternating row colors
        self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
        self.filter_tree.tag_configure('oddrow',
background='#ffffff')

        # Insert initial data
        self.update_filter_table(self.df)

    def create_graph_tab(self):
        self.graph_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.graph_frame,
text='Student Major Distribution')

        # Create a Matplotlib figure and axis
        self.fig, self.ax = plt.subplots(figsize=
(8, 6))
        self.ax.set_title('Student Major
Distribution')
        self.ax.set_xlabel('Major')
        self.ax.set_ylabel('Number of Students')

        # Create a canvas for Matplotlib
        self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)
```

```
    self.canvas.get_tk_widget().pack(expand=True,
fill='both')

        # Plot data
        self.update_plot()

    def create_university_tab(self):
        self.university_frame =
ttk.Frame(self.notebook)
        self.notebook.add(self.university_frame,
text='Students by University')

        # Create a Matplotlib figure and axis
        self.fig_uni, self.ax_uni =
plt.subplots(figsize=(8, 6))
        self.ax_uni.set_title('Number of Students
per University')
        self.ax_uni.set_xlabel('University')
        self.ax_uni.set_ylabel('Number of
Students')

        # Create a canvas for Matplotlib
        self.canvas_uni =
FigureCanvasTkAgg(self.fig_uni,
master=self.university_frame)

        self.canvas_uni.get_tk_widget().pack(expand=True,
fill='both')

        # Plot data
        self.update_university_plot()

    def create_location_tab(self):
        self.location_frame =
ttk.Frame(self.notebook)
```

```
        self.notebook.add(self.location_frame,
text='Students by Location')

        # Create a Matplotlib figure and axis
        self.fig_loc, self.ax_loc =
plt.subplots(figsize=(8, 6))
        self.ax_loc.set_title('Number of Students
per Location')
        self.ax_loc.set_xlabel('Location')
        self.ax_loc.set_ylabel('Number of
Students')

        # Create a canvas for Matplotlib
        self.canvas_loc =
FigureCanvasTkAgg(self.fig_loc,
master=self.location_frame)

        self.canvas_loc.get_tk_widget().pack(expand=True,
fill='both')

        # Plot data
        self.update_location_plot()

    def update_table(self, data):
        for row in self.tree.get_children():
            self.tree.delete(row)
        for index, row in data.iterrows():
            tag = 'evenrow' if index % 2 == 0 else
'oddrow'
            self.tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_filter_table(self, data):
        for row in
self.filter_tree.get_children():
            self.filter_tree.delete(row)
```

```
        for index, row in data.iterrows():
            tag = 'evenrow' if index % 2 == 0 else
'oddrow'
                self.filter_tree.insert('', 'end',
values=list(row), tags=(tag,))

    def update_plot(self):
        self.ax.clear()
        self.ax.set_title('Student Major
Distribution')
        self.ax.set_xlabel('Major')
        self.ax.set_ylabel('Number of Students')

        # Plot major distribution
        major_counts =
self.df['Major'].value_counts()
        self.ax.bar(major_counts.index,
major_counts.values, color='skyblue',
edgecolor='black')

        self.canvas.draw()

    def update_university_plot(self):
        self.ax_uni.clear()
        self.ax_uni.set_title('Number of Students
per University')
        self.ax_uni.set_xlabel('University')
        self.ax_uni.set_ylabel('Number of
Students')

        # Plot university distribution
        university_counts =
self.df['UniversityName'].value_counts()
        self.ax_uni.bar(university_counts.index,
university_counts.values, color='lightgreen',
edgecolor='black')
```

```
    self.canvas_uni.draw()

def update_location_plot(self):
    self.ax_loc.clear()
    self.ax_loc.set_title('Number of Students
per Location')
    self.ax_loc.set_xlabel('Location')
    self.ax_loc.set_ylabel('Number of
Students')

    # Plot location distribution
    location_counts =
self.df['Location_y'].value_counts()
    self.ax_loc.bar(location_counts.index,
location_counts.values, color='salmon',
edgecolor='black')

    self.canvas_loc.draw()

def apply_filters(self, event=None):
    university_id =
self.university_combobox.get()
    major = self.major_combobox.get()

    filtered_df = self.df

    if university_id:
        filtered_df =
filtered_df[filtered_df['UniversityID'] ==
int(university_id)]

    if major:
        filtered_df =
filtered_df[filtered_df['Major'] == major]
```

```
        if filtered_df.empty:
            messagebox.showinfo("No Data", "No
data available for the selected filters.")
            filtered_df = self.df

        # Update the table in the Filters tab
        self.update_filter_table(filtered_df)

    def reset_filters(self):
        self.university_combobox.set('')
        self.major_combobox.set('')
        self.update_filter_table(self.df)

    def save_filtered_data(self):
        filtered_df = self.get_filtered_data()
        if filtered_df.empty:
            messagebox.showinfo("No Data", "No
data to save.")
            return
        file_path =
'filtered_university_data.xlsx'
        filtered_df.to_excel(file_path,
index=False)
        messagebox.showinfo("Saved", f"Filtered
data has been saved to '{file_path}'")

    def get_filtered_data(self):
        university_id =
self.university_combobox.get()
        major = self.major_combobox.get()

        filtered_df = self.df

        if university_id:
            filtered_df =
filtered_df[filtered_df['UniversityID'] ==
```

```
int(university_id)]  
  
    if major:  
        filtered_df =  
filtered_df[filtered_df['Major'] == major]  
  
    return filtered_df  
  
if __name__ == "__main__":  
    app = UniversityApp(combined_df)  
    app.mainloop()
```

Here's a detailed explanation of the code:

1. Import Statements

```
import tkinter as tk  
from tkinter import ttk, messagebox  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.backends.backend_tkagg import  
FigureCanvasTkAgg
```

- **tkinter**: A standard GUI toolkit for Python used to create graphical user interfaces.
- **ttk**: Provides access to Tk themed widgets, which offer a more modern look than standard Tkinter widgets.
- **messagebox**: Part of tkinter, used to display message boxes.
- **pandas**: A powerful data analysis library used for handling and manipulating structured data.
- **numpy**: A library for numerical computations in Python, used here to generate synthetic data.

- `matplotlib.pyplot`: A plotting library used for creating visualizations.
- `FigureCanvasTkAgg`: A Matplotlib backend that allows Matplotlib plots to be embedded in Tkinter applications.

2. Data Generation Functions

`generate_university_data(num_universities)`

```
def generate_university_data(num_universities):
    np.random.seed(0)
    university_ids = np.arange(1, num_universities
+ 1)
        university_names = np.random.choice(['Tech
University', 'Liberal Arts College', 'Medical
School', 'Business Institute', 'Engineering
Academy'], size=num_universities)
    locations = np.random.choice(['New York', 'Los
Angeles', 'Chicago', 'Houston', 'Phoenix'],
size=num_universities)
    return pd.DataFrame({
        'UniversityID': university_ids,
        'UniversityName': university_names,
        'Location': locations
    })
```

- Purpose: Generates a DataFrame containing synthetic university data.
- Parameters:
 - `num_universities`: Number of universities to generate.
- Functionality:
 - `np.random.seed(0)`: Ensures reproducibility of the random data.

- `np.arange(1, num_universities + 1)`: Creates a range of university IDs.
- `np.random.choice`: Randomly selects names and locations for universities.
- `pd.DataFrame`: Constructs a DataFrame with the generated data.

generate_student_data(num_students, num_universities)

```
def generate_student_data(num_students,
    num_universities):
    np.random.seed(1)
    student_ids = np.arange(1, num_students + 1)
    student_names = np.random.choice(['John Doe',
        'Jane Smith', 'Emily Davis', 'Michael Brown',
        'Sarah Wilson'], size=num_students)
    majors = np.random.choice(['Computer Science',
        'Biology', 'Business', 'Engineering', 'History'],
        size=num_students)
    university_ids = np.random.choice(np.arange(1,
        num_universities + 100), size=num_students) # Ensure IDs are within range
    locations = np.random.choice(['New York', 'Los Angeles',
        'Chicago', 'Houston', 'Phoenix'],
        size=num_students) # Ensure locations match universities
    return pd.DataFrame({
        'StudentID': student_ids,
        'StudentName': student_names,
        'Major': majors,
        'UniversityID': university_ids,
        'Location': locations
    })
```

- Purpose: Generates a DataFrame containing synthetic student data.
- Parameters:
 - num_students: Number of students to generate.
 - num_universities: Number of universities to reference.
- Functionality:
 - np.random.seed(1): Ensures reproducibility of the random data.
 - np.arange(1, num_students + 1): Creates a range of student IDs.
 - np.random.choice: Randomly selects student names, majors, university IDs, and locations.
 - pd.DataFrame: Constructs a DataFrame with the generated data.

3. Data Combination and Export

```

num_universities = 100
num_students = 5000
universities_df
= generate_university_data(num_universities)
students_df = generate_student_data(num_students,
num_universities)

combined_df = pd.merge(students_df,
universities_df, on='UniversityID', how='right')

file_path = 'university_student_right_join.xlsx'
combined_df.to_excel(file_path, index=False)

```

- Purpose: Generates and combines the university and student datasets, and then saves the result to an Excel file.
- Functionality:
 - pd.merge: Merges student and university DataFrames on the UniversityID column using a right join, keeping all universities and matching student records.
 - to_excel: Exports the combined DataFrame to an Excel file.

4. Tkinter Application Class: UniversityApp Initialization

```
class UniversityApp(tk.Tk):
    def __init__(self, df):
        super().__init__()
        self.title('University and Student Data
Viewer')
        self.geometry('1200x900')    # Increased
height to accommodate additional tabs
        self.df = df

        print(self.df.columns)

        self.notebook = ttk.Notebook(self)
        self.notebook.pack(expand=True,
fill='both')

        self.create_table_tab()
        self.create_filter_tab()
        self.create_graph_tab()
        self.create_university_tab()
        self.create_location_tab()
```

- Purpose: Sets up the main Tkinter window and initializes various tabs for displaying data.
- Parameters:
 - df: DataFrame containing the combined data.
- Functionality:
 - super().__init__(): Initializes the Tkinter Tk class.
 - self.notebook = ttk.Notebook(self): Creates a tabbed interface.
 - self.notebook.pack(expand=True, fill='both'): Packs the notebook widget to expand and fill the window.
 - Tab Methods: Calls methods to create different tabs for data display.

create_table_tab()

```
def create_table_tab(self):
    self.table_frame = ttk.Frame(self.notebook)
    self.notebook.add(self.table_frame, text='DataTable')

        self.tree = ttk.Treeview(self.table_frame,
columns=list(self.df.columns), show='headings')
        self.tree.pack(expand=True, fill='both')

    for col in self.df.columns:
        self.tree.heading(col, text=col)
            self.tree.column(col, width=150,
anchor='center')

                self.tree.tag_configure('evenrow',
background='#f0f0f0')
                    self.tree.tag_configure('oddrow',
background='#ffffff')
```

```
self.update_table(self.df)
```

- Purpose: Creates a tab displaying the entire dataset in a table format.
- Functionality:
 - ttk.Treeview: Creates a table-like widget for displaying data.
 - self.tree.heading and self.tree.column: Configures column headings and widths.
 - tag_configure: Sets alternating row colors for readability.
 - update_table: Updates the table with data from the DataFrame.

create_filter_tab()

```
def create_filter_tab(self):  
    self.filter_frame = ttk.Frame(self.notebook)  
        self.notebook.add(self.filter_frame,  
text='Filters')  
  
                self.university_label      =  
ttk.Label(self.filter_frame,           text='Select  
University ID:')  
    self.university_label.pack(pady=5)  
  
                self.university_combobox      =  
ttk.Combobox(self.filter_frame,  
values=sorted(self.df['UniversityID'].dropna().uni  
que()))  
    self.university_combobox.pack(pady=5)
```

```
self.university_combobox.bind('<<ComboboxSelected>>', self.apply_filters)

                self.major_label      =
ttk.Label(self.filter_frame, text='Select Major:')
    self.major_label.pack(pady=5)

                self.major_combobox      =
ttk.Combobox(self.filter_frame,
values=sorted(self.df['Major'].dropna().unique()))
    self.major_combobox.pack(pady=5)

self.major_combobox.bind('<<ComboboxSelected>>', self.apply_filters)

                self.reset_button      =
ttk.Button(self.filter_frame,           text='Reset
Filters', command=self.reset_filters)
    self.reset_button.pack(pady=5)

                self.save_button      =
ttk.Button(self.filter_frame,   text='Save Filtered
Data', command=self.save_filtered_data)
    self.save_button.pack(pady=5)

                self.filter_tree      =
ttk.Treeview(self.filter_frame,
columns=list(self.df.columns), show='headings')
    self.filter_tree.pack(expand=True,
fill='both')

for col in self.df.columns:
    self.filter_tree.heading(col, text=col)
        self.filter_tree.column(col, width=150,
anchor='center')
```

```

        self.filter_tree.tag_configure('evenrow',
background='#f0f0f0')
        self.filter_tree.tag_configure('oddrow',
background='#ffffff')

    self.update_filter_table(self.df)

```

- Purpose: Creates a tab with filters for University ID and Major, and displays filtered data in a table.
- Functionality:
 - ttk.Combobox: Dropdown menus for selecting filters.
 - bind: Binds selection events to the apply_filters method.
 - ttk.Button: Buttons for resetting filters and saving filtered data.
 - update_filter_table: Updates the table based on applied filters.

create_graph_tab()

```

def create_graph_tab(self):
    self.graph_frame = ttk.Frame(self.notebook)
        self.notebook.add(self.graph_frame,
text='Student Major Distribution')

    self.fig, self.ax = plt.subplots(figsize=(8,
6))
        self.ax.set_title('Student      Major
Distribution')
    self.ax.set_xlabel('Major')
    self.ax.set_ylabel('Number of Students')

```

```
        self.canvas = FigureCanvasTkAgg(self.fig,
master=self.graph_frame)
        self.canvas.get_tk_widget().pack(expand=True,
fill='both')

    self.update_plot()
```

- Purpose: Creates a tab for displaying a bar chart of student major distribution.
- Functionality:
 - plt.subplots: Creates a Matplotlib figure and axis.
 - FigureCanvasTkAgg: Embeds the Matplotlib figure in the Tkinter window.
 - update_plot: Updates the plot with data.

create_university_tab()

```
def create_university_tab(self):
    self.university_frame =
ttk.Frame(self.notebook)
    self.notebook.add(self.university_frame,
text='Students by University')

    self.fig_uni, self.ax_uni =
plt.subplots(figsize=(8, 6))
    self.ax_uni.set_title('Number of Students per
University')
    self.ax_uni.set_xlabel('University')
    self.ax_uni.set_ylabel('Number of Students')

    self.canvas_uni =
FigureCanvasTkAgg(self.fig_uni,
master=self.university_frame)
```

```
self.canvas_uni.get_tk_widget().pack(expand=True,  
fill='both')  
  
    self.update_university_plot()
```

- Purpose: Creates a tab for displaying a bar chart of students by university.
- Functionality:
 - plt.subplots: Creates a Matplotlib figure and axis.
 - FigureCanvasTkAgg: Embeds the Matplotlib figure in the Tkinter window.
 - update_university_plot: Updates the plot with data.

create_location_tab()

```
def create_location_tab(self):  
    self.location_frame = ttk.Frame(self.notebook)  
        self.notebook.add(self.location_frame,  
text='Students by Location')  
  
                self.fig_loc,      self.ax_loc      =  
plt.subplots(figsize=(8, 6))  
    self.ax_loc.set_title('Number of Students per  
Location')  
    self.ax_loc.set_xlabel('Location')  
    self.ax_loc.set_ylabel('Number of Students')  
  
                self.canvas_loc      =  
FigureCanvasTkAgg(self.fig_loc,  
master=self.location_frame)  
  
self.canvas_loc.get_tk_widget().pack(expand=True,  
fill='both')
```

```
self.update_location_plot()
```

- Purpose: Creates a tab for displaying a bar chart of students by location.
- Functionality:
 - plt.subplots: Creates a Matplotlib figure and axis.
 - FigureCanvasTkAgg: Embeds the Matplotlib figure in the Tkinter window.
 - update_location_plot: Updates the plot with data.

Data Updating Methods

- update_table(data): Updates the data table in the "Data Table" tab.
- update_filter_table(data): Updates the filtered data table in the "Filters" tab.
- update_plot(): Updates the student major distribution plot.
- update_university_plot(): Updates the number of students per university plot.
- update_location_plot(): Updates the number of students per location plot.

Filter and Save Methods

- apply_filters(event=None): Filters data based on user selections and updates the filtered table.
- reset_filters(): Resets the filters and updates the table to show all data.
- save_filtered_data(): Saves the currently filtered data to an Excel file.
- get_filtered_data(): Retrieves data based on the currently applied filters.

5. Running the Application

```
if __name__ == "__main__":
    app = UniversityApp(combined_df)
    app.mainloop()
```

- Purpose: Starts the Tkinter application.
- Functionality:
 - UniversityApp(combined_df): Creates an instance of the UniversityApp class, passing the combined data.
 - app.mainloop(): Starts the Tkinter event loop, making the application responsive to user actions.

The screenshot shows a window titled "University and Student Data Viewer". At the top, there are tabs for "Data Table", "Filters", "Student Major Distribution", "Students by University", and "Students by Location". The main area is a table with the following columns: StudentID, StudentName, Major, UniversityID, Location_x, UniversityName, and Location_y. The data consists of approximately 30 rows of student information, including names like Emily Davis, Sarah Wilson, John Doe, Jane Smith, Michael Brown, and their respective majors (Computer Science, Engineering, Business, History, Biology) and locations (Phoenix, Houston, New York, Los Angeles, Chicago). The table has a light gray background with alternating row colors for readability.

| StudentID | StudentName | Major | UniversityID | Location_x | UniversityName | Location_y |
|-----------|---------------|------------------|--------------|-------------|---------------------|------------|
| 435 | Emily Davis | Computer Science | 1 | Phoenix | Engineering Academy | Chicago |
| 541 | Sarah Wilson | Engineering | 1 | Houston | Engineering Academy | Chicago |
| 661 | John Doe | Business | 1 | Phoenix | Engineering Academy | Chicago |
| 716 | John Doe | Business | 1 | New York | Engineering Academy | Chicago |
| 923 | John Doe | Engineering | 1 | Phoenix | Engineering Academy | Chicago |
| 1200 | John Doe | Computer Science | 1 | Chicago | Engineering Academy | Chicago |
| 1298 | Michael Brown | Engineering | 1 | Chicago | Engineering Academy | Chicago |
| 2422 | John Doe | Engineering | 1 | Los Angeles | Engineering Academy | Chicago |
| 2565 | Jane Smith | Business | 1 | Houston | Engineering Academy | Chicago |
| 2923 | Sarah Wilson | History | 1 | Phoenix | Engineering Academy | Chicago |
| 2977 | Michael Brown | Engineering | 1 | Houston | Engineering Academy | Chicago |
| 3073 | John Doe | Biology | 1 | Los Angeles | Engineering Academy | Chicago |
| 3234 | Michael Brown | Computer Science | 1 | New York | Engineering Academy | Chicago |
| 3410 | Michael Brown | Engineering | 1 | Phoenix | Engineering Academy | Chicago |
| 3615 | Jane Smith | Business | 1 | Los Angeles | Engineering Academy | Chicago |
| 3924 | Emily Davis | Business | 1 | Phoenix | Engineering Academy | Chicago |
| 4145 | Emily Davis | History | 1 | Phoenix | Engineering Academy | Chicago |
| 4283 | Jane Smith | History | 1 | Houston | Engineering Academy | Chicago |
| 4436 | Emily Davis | Biology | 1 | New York | Engineering Academy | Chicago |
| 115 | Emily Davis | Computer Science | 2 | Phoenix | Tech University | Houston |
| 140 | Jane Smith | Business | 2 | Phoenix | Tech University | Houston |
| 579 | John Doe | Biology | 2 | Chicago | Tech University | Houston |
| 948 | Emily Davis | Engineering | 2 | Chicago | Tech University | Houston |
| 1085 | Sarah Wilson | History | 2 | Los Angeles | Tech University | Houston |
| 1086 | John Doe | Biology | 2 | Chicago | Tech University | Houston |
| 1136 | Michael Brown | Engineering | 2 | New York | Tech University | Houston |
| 1476 | John Doe | History | 2 | Los Angeles | Tech University | Houston |
| 2524 | Michael Brown | History | 2 | New York | Tech University | Houston |
| 2591 | Michael Brown | Business | 2 | Phoenix | Tech University | Houston |
| 2740 | John Doe | Biology | 2 | Chicago | Tech University | Houston |
| 2781 | Sarah Wilson | History | 2 | New York | Tech University | Houston |
| 3022 | Emily Davis | Business | 2 | Los Angeles | Tech University | Houston |
| 3333 | Jane Smith | Engineering | 2 | Phoenix | Tech University | Houston |
| 3382 | Sarah Wilson | Computer Science | 2 | Chicago | Tech University | Houston |

University and Student Data Viewer

Data Table | Filters | Student Major Distribution | Students by University | Students by Location

Select University ID:

Select Major:

| StudentID | StudentName | Major | UniversityID | Location_x | UniversityName | Location_y |
|-----------|---------------|------------------|--------------|-------------|----------------------|------------|
| 252 | Emily Davis | Engineering | 6 | Chicago | Liberal Arts College | Houston |
| 368 | John Doe | Business | 6 | Los Angeles | Liberal Arts College | Houston |
| 487 | Sarah Wilson | History | 6 | Los Angeles | Liberal Arts College | Houston |
| 1205 | Sarah Wilson | Business | 6 | Phoenix | Liberal Arts College | Houston |
| 1316 | Emily Davis | Biology | 6 | Phoenix | Liberal Arts College | Houston |
| 1520 | Michael Brown | Computer Science | 6 | Houston | Liberal Arts College | Houston |
| 1703 | Michael Brown | Biology | 6 | New York | Liberal Arts College | Houston |
| 1728 | Jane Smith | Engineering | 6 | Houston | Liberal Arts College | Houston |
| 1785 | John Doe | History | 6 | Chicago | Liberal Arts College | Houston |
| 1995 | Jane Smith | Computer Science | 6 | Los Angeles | Liberal Arts College | Houston |
| 2057 | Jane Smith | Business | 6 | Los Angeles | Liberal Arts College | Houston |
| 2084 | Emily Davis | Engineering | 6 | Houston | Liberal Arts College | Houston |
| 2114 | Emily Davis | Business | 6 | Houston | Liberal Arts College | Houston |
| 2249 | Emily Davis | Engineering | 6 | Los Angeles | Liberal Arts College | Houston |
| 2691 | Emily Davis | Computer Science | 6 | Houston | Liberal Arts College | Houston |
| 2701 | John Doe | Business | 6 | Chicago | Liberal Arts College | Houston |
| 2779 | Jane Smith | Business | 6 | Los Angeles | Liberal Arts College | Houston |
| 2839 | Sarah Wilson | Engineering | 6 | Los Angeles | Liberal Arts College | Houston |
| 3249 | Sarah Wilson | Computer Science | 6 | Houston | Liberal Arts College | Houston |
| 3349 | Emily Davis | History | 6 | Houston | Liberal Arts College | Houston |
| 3672 | Emily Davis | Computer Science | 6 | Los Angeles | Liberal Arts College | Houston |
| 3705 | Michael Brown | Engineering | 6 | Chicago | Liberal Arts College | Houston |
| 4002 | Jane Smith | Biology | 6 | New York | Liberal Arts College | Houston |
| 4198 | Emily Davis | Business | 6 | Los Angeles | Liberal Arts College | Houston |

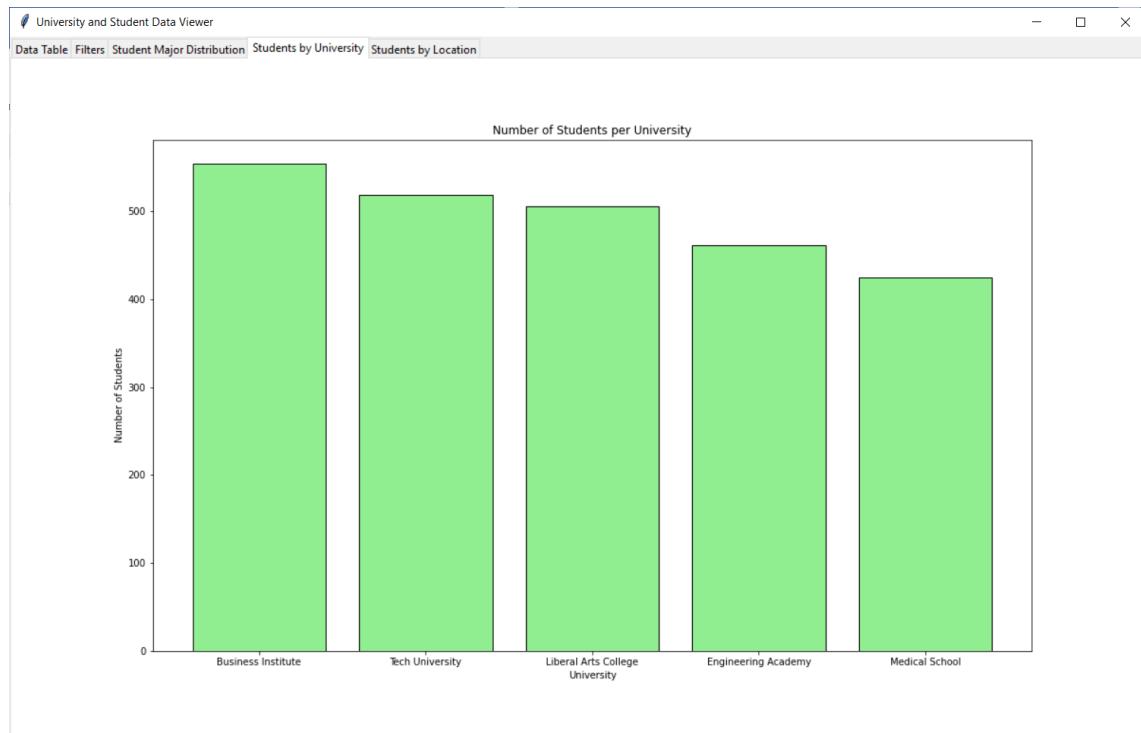
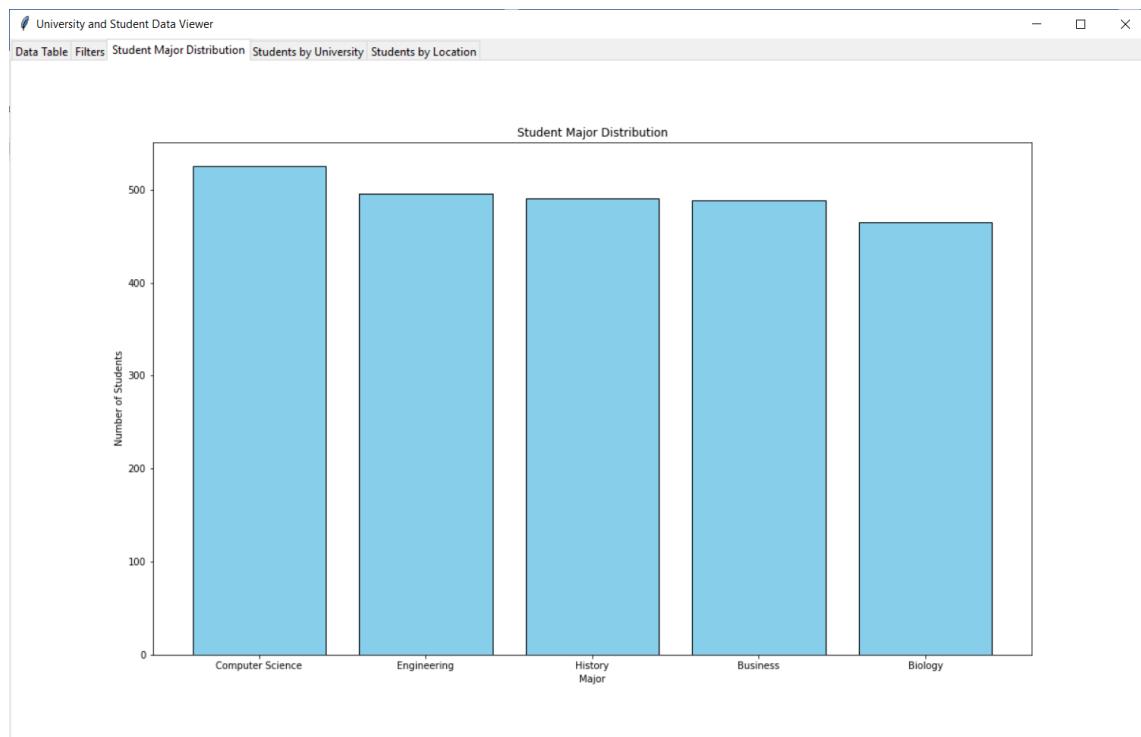
University and Student Data Viewer

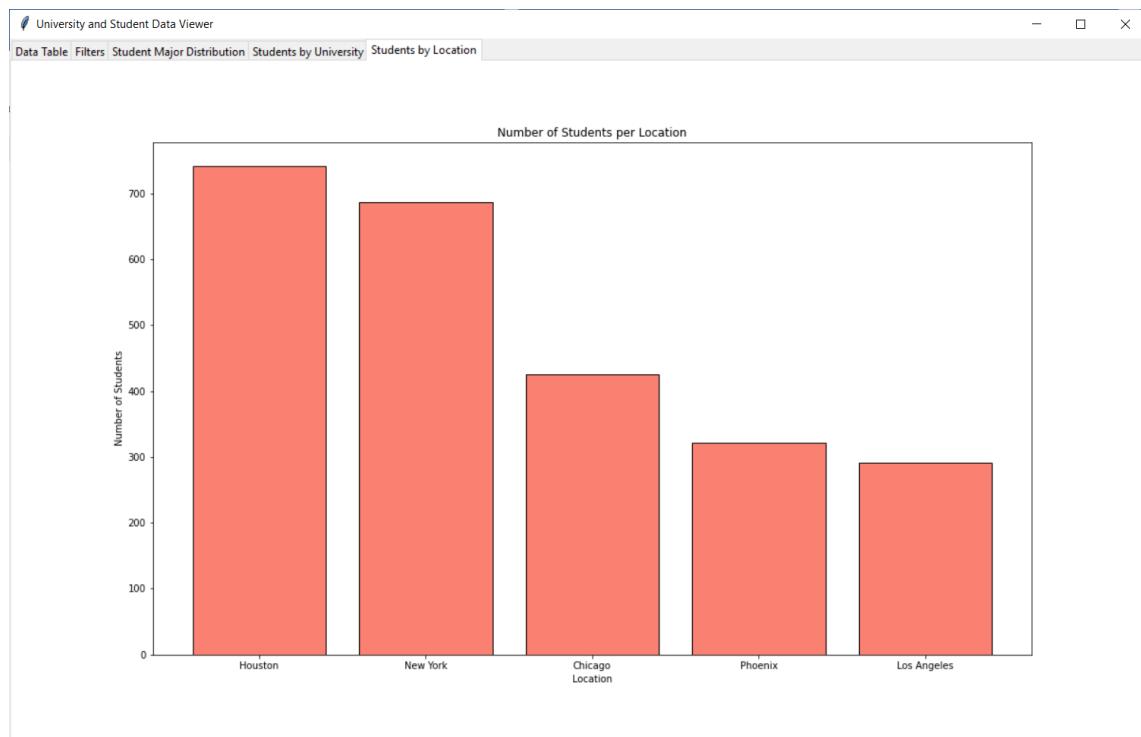
Data Table | Filters | Student Major Distribution | Students by University | Students by Location

Select University ID:

Select Major:

| StudentID | StudentName | Major | UniversityID | Location_x | UniversityName | Location_y |
|-----------|---------------|------------------|--------------|-------------|----------------------|------------|
| 1520 | Michael Brown | Computer Science | 6 | Houston | Liberal Arts College | Houston |
| 1995 | Jane Smith | Computer Science | 6 | Los Angeles | Liberal Arts College | Houston |
| 2691 | Emily Davis | Computer Science | 6 | Houston | Liberal Arts College | Houston |
| 3249 | Sarah Wilson | Computer Science | 6 | Houston | Liberal Arts College | Houston |
| 3672 | Emily Davis | Computer Science | 6 | Los Angeles | Liberal Arts College | Houston |
| 4845 | Sarah Wilson | Computer Science | 6 | Phoenix | Liberal Arts College | Houston |





DATAFRAME GROUPING AND AGGREGATING DATAFRAME GROUPING AND AGGREGATING

INTRODUCTION

Grouping in a DataFrame is a way to aggregate and summarize data based on one or more columns. It allows you to perform operations on subsets of your data that share common values in specified columns. This is often used to compute summary statistics or perform transformations on groups of data.

Here's a basic outline of how grouping works in a DataFrame using pandas in Python:

- Group By Column(s): You specify one or more columns to group by. The DataFrame will then create separate groups for each unique combination of values in those columns.
- Apply Aggregations: Once the data is grouped, you can apply aggregation functions like sum(), mean(), count(), max(), min(), etc., to each group. This provides summarized information for each group.
- Transform or Filter: Beyond aggregation, you can also apply transformations or filters to groups. For example, you might want to normalize data within each group or filter out certain groups based on criteria.

Grouping in a DataFrame involves several detailed steps and concepts. Here's a thorough breakdown:

1. Concept of Grouping

Grouping is a way to aggregate data into subsets based on the values in one or more columns. It helps to perform operations on

these subsets rather than on the entire dataset. This is useful for generating summaries or analyzing patterns within the data.

2. Basic Syntax and Steps

Creating a DataFrame

First, you need a DataFrame. Here's an example DataFrame:

```
import pandas as pd

# Sample DataFrame
data = {
    'Country': ['USA', 'USA', 'Canada', 'Canada',
'Mexico', 'Mexico'],
    'Food': ['Apples', 'Oranges', 'Apples',
'Oranges', 'Apples', 'Oranges'],
    'Production': [100, 150, 200, 250, 300, 350]
}
df = pd.DataFrame(data)
```

Grouping Data

To group data, use the `groupby()` method. This method takes one or more column names as arguments.

```
grouped = df.groupby('Country')
```

`df.groupby('Country')`: This groups the data by the Country column. The result is a `DataFrameGroupBy` object, which is a collection of dataframes, each corresponding to a unique value in the Country column.

3. Operations on Grouped Data

Aggregation

Aggregation involves applying a function to each group to get summarized information. Common aggregation functions include:

- `sum()`: Adds up the values in each group.
- `mean()`: Calculates the average of the values in each group.
- `count()`: Counts the number of entries in each group.
- `max()`: Finds the maximum value in each group.
- `min()`: Finds the minimum value in each group.

Example: To get the total production for each country:

```
total_production = grouped['Production'].sum()
print(total_production)
```

Output:

```
Country
Canada    450
Mexico    650
USA      250
Name: Production, dtype: int64
```

Multiple Aggregations

You can apply multiple aggregation functions using the `agg()` method:

```
aggregated = grouped['Production'].agg(['sum',
'mean', 'max'])
print(aggregated)
```

Output:

| | sum | mean | max |
|---------|-----|------|-----|
| Country | | | |
| Canada | 450 | 225 | 250 |
| Mexico | 650 | 325 | 350 |

| | | | |
|-----|-----|-----|-----|
| USA | 250 | 125 | 150 |
|-----|-----|-----|-----|

Transformations

Transformations allow you to apply functions to each group and return a DataFrame of the same shape as the original. This is useful for operations where the result should be aligned with the original data.

Example: Normalize the production values within each country:

```
normalized =  
grouped['Production'].transform(lambda x: (x -  
x.mean()) / x.std())  
df['Normalized_Production'] = normalized  
print(df)
```

Filtering

Filtering lets you remove groups that don't meet a certain condition. Use the filter() method for this purpose:

Example: Keep only countries where the total production is greater than 400:

```
filtered = grouped.filter(lambda x:  
x['Production'].sum() > 400)  
print(filtered)
```

Output:

| | Country | Food | Production |
|---|---------|---------|------------|
| 2 | Canada | Apples | 200 |
| 3 | Canada | Oranges | 250 |
| 4 | Mexico | Apples | 300 |
| 5 | Mexico | Oranges | 350 |

4. Advanced Grouping

Grouping by Multiple Columns

You can group by multiple columns to create nested groups:

```
grouped_multi = df.groupby(['Country', 'Food'])  
average_production =  
grouped_multi['Production'].mean()  
print(average_production)
```

Output:

```
Country Food  
Canada  Apples    200  
        Oranges   250  
Mexico  Apples    300  
        Oranges   350  
USA     Apples    100  
        Oranges   150  
Name: Production, dtype: int64
```

Grouping with Aggregation and Multiple Functions

You can apply multiple aggregation functions to different columns within the same group:

```
result = df.groupby('Country').agg({  
    'Production': ['sum', 'mean'],  
    'Food': 'count'  
})  
print(result)
```

Output:

| | Production | Food | |
|--|------------|------|-------|
| | sum | mean | count |

| Country | | | |
|---------|-----|-------|---|
| Canada | 450 | 225.0 | 2 |
| Mexico | 650 | 325.0 | 2 |
| USA | 250 | 125.0 | 2 |

5. Understanding GroupBy Object

The DataFrameGroupBy object is not a DataFrame itself but an intermediary object that contains information about the groups. You can apply various functions to this object to get the desired results.

Summary

Grouping is a powerful technique for summarizing and analyzing subsets of data in a DataFrame. By grouping data based on one or more columns, you can perform various operations, such as aggregations, transformations, and filters, to derive insights from your dataset.

EXAMPLE 4.1

Grouping with Synthetic Insurance Data

The purpose of the code is to create a synthetic insurance dataset, simulate the structure of real-world insurance data, and then perform basic data manipulation and storage tasks. The dataset includes various columns such as Policyholder, Age, State, Coverage_Type, and Premium, which represent different aspects of an insurance policy. By using numpy to generate random data and pandas to organize and handle this data, the script provides a framework for testing data processing techniques and analyzing data within a controlled environment.

Once the dataset is created, the script groups the data by State and Coverage_Type to illustrate how to handle and inspect subsets of

the data. This grouping operation is crucial for understanding how data can be segmented based on specific attributes, although the script does not perform aggregation or complex analysis. Finally, the dataset is saved to an Excel file using pandas' `to_excel()` function, facilitating easy sharing and further analysis in spreadsheet software. This comprehensive approach demonstrates the workflow from data generation and manipulation to storage, providing a useful template for similar data processing tasks.

```
import pandas as pd
import numpy as np

# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 1000

# Create larger synthetic insurance dataset
data = {
    'Policyholder': [f'Policyholder_{i}' for i in
range(num_records)],
    'Age': np.random.randint(18, 70,
size=num_records),
    'State': np.random.choice(['New York',
'California', 'Texas', 'Florida', 'Illinois'],
size=num_records),
    'Coverage_Type': np.random.choice(['Basic',
'Standard', 'Premium'], size=num_records),
    'Premium': np.random.uniform(100, 500,
size=num_records).round(2)
}

df = pd.DataFrame(data)
```

```
# Group by State and Coverage_Type
grouped = df.groupby(['State', 'Coverage_Type'])

# Print a sample of the groups (to avoid
# overwhelming output)
print("Sample of grouped data:")
for name, group in grouped:
    print(f"Group Name: {name}")
    print(group.head(1)) # Print only the first
entry of each group
    print()

# Save DataFrame to Excel using to_excel()
output_file =
'large_synthetic_insurance_data.xlsx'
df.to_excel(output_file, sheet_name='Insurance
Data', index=False)

print(f"Data saved to {output_file}")
```

Here's a detailed breakdown of the provided script:

Importing Libraries

```
import pandas as pd
import numpy as np
```

- pandas: A powerful library for data manipulation and analysis in Python. It provides data structures like DataFrames, which are used to store and manipulate tabular data.
- numpy: A library for numerical computing in Python. It is used here to generate random numbers and perform various numerical operations.

Setting a Seed for Reproducibility

```
np.random.seed(0)
```

- np.random.seed(0): Sets the random number generator's seed to 0. This ensures that the random numbers generated are the same each time you run the script, which is useful for reproducibility.

Defining the Number of Records

```
num_records = 1000
```

- num_records: Specifies the number of records (rows) to generate for the synthetic dataset. In this case, it is set to 1000.

Creating the Synthetic Insurance Dataset

```
data = {
    'Policyholder': [f'Policyholder_{i}' for i in
range(num_records)],
    'Age': np.random.randint(18, 70,
size=num_records),
    'State': np.random.choice(['New York',
'California', 'Texas', 'Florida', 'Illinois'],
size=num_records),
    'Coverage_Type': np.random.choice(['Basic',
'Standard', 'Premium'], size=num_records),
    'Premium': np.random.uniform(100, 500,
size=num_records).round(2)
}
```

- 'Policyholder': Creates a list of policyholder names formatted as Policyholder_0, Policyholder_1, ..., up to Policyholder_999. This generates unique names for each record.
- 'Age': Generates random integers between 18 and 70 (inclusive) for each record using np.random.randint(). This simulates the ages of policyholders.
- 'State': Randomly selects a state from the list ['New York', 'California', 'Texas', 'Florida', 'Illinois'] for each record using np.random.choice().
- 'Coverage_Type': Randomly selects a coverage type from ['Basic', 'Standard', 'Premium'] for each record.
- 'Premium': Generates random floating-point numbers between 100 and 500 for each record using np.random.uniform(). The .round(2) method rounds these values to two decimal places.

Creating the DataFrame

```
df = pd.DataFrame(data)
```

- pd.DataFrame(data): Converts the dictionary data into a pandas DataFrame. Each key in the dictionary becomes a column in the DataFrame, and each value (list or array) becomes the data for that column.

Grouping the DataFrame

```
grouped = df.groupby(['State', 'Coverage_Type'])
```

- df.groupby(['State', 'Coverage_Type']): Groups the DataFrame by the State and Coverage_Type columns. The result is a DataFrameGroupBy object that holds the data split into groups based on unique combinations of values in these columns.

Printing a Sample of the Groups

```
print("Sample of grouped data:")
for name, group in grouped:
    print(f"Group Name: {name}")
    print(group.head(1)) # Print only the first
entry of each group
    print()
```

- for name, group in grouped: Iterates over the groups in the DataFrameGroupBy object. name is a tuple representing the grouping keys (e.g., ('New York', 'Basic')), and group is the DataFrame corresponding to that group.
- group.head(1): Prints only the first entry of each group to avoid overwhelming output. This shows a sample record from each group.

Saving the DataFrame to an Excel File

```
output_file = 'large_synthetic_insurance_data.xlsx'
df.to_excel(output_file, sheet_name='Insurance
Data', index=False)
```

- `output_file`: Specifies the name of the Excel file to save the DataFrame to.
- `df.to_excel(output_file, sheet_name='Insurance Data', index=False)`: Saves the DataFrame `df` to an Excel file with the specified filename. The `sheet_name` parameter sets the name of the sheet within the Excel file, and `index=False` prevents saving the DataFrame index as an additional column in the file.

Printing Confirmation

```
print(f"Data saved to {output_file}")
```

- `print(f"Data saved to {output_file}")`: Prints a confirmation message indicating that the data has been successfully saved to the specified Excel file.

This script generates a larger synthetic dataset, groups it, samples a portion of the groups for inspection, and saves the dataset to an Excel file.

EXAMPLE 4.2

GUI Tkinter for Grouping with Synthetic Insurance Data

This Tkinter GUI application is designed for analyzing and visualizing a synthetic insurance dataset. The code generates a dataset with 1,000 records, including policyholder details, age, state, coverage type, and premium amounts. This dataset serves as a mock example for testing and demonstrating the functionality of the GUI. The main window of the application displays this data in a

scrollable table using Tkinter's Treeview widget, which allows users to interactively explore the dataset.

The GUI includes several interactive features to filter and manage the data. Users can apply filters based on state and coverage type through comboboxes. When filters are applied, the data in the Treeview updates to show only the records that match the selected criteria. Additionally, users have the option to save the filtered data to an Excel file, providing a way to export and share the results of their analysis. These features are facilitated by buttons that trigger the corresponding functions for applying filters and saving data.

The application also includes a data visualization component, where users can generate a bar plot of the number of policies by state. This plot is created using Matplotlib and embedded into the Tkinter window. This feature allows users to visually analyze the distribution of policies across different states. Overall, the GUI provides a rich set of tools for data analysis, including viewing, filtering, exporting, and visualizing the insurance data, all within a user-friendly interface.

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Generate the synthetic insurance dataset
np.random.seed(0)
num_records = 1000
data = {
```

```
'Policyholder': [f'Policyholder_{i}' for i in
range(num_records)],
    'Age': np.random.randint(18, 70,
size=num_records),
    'State': np.random.choice(['New York',
'California', 'Texas', 'Florida', 'Illinois'],
size=num_records),
    'Coverage_Type': np.random.choice(['Basic',
'Standard', 'Premium'], size=num_records),
    'Premium': np.random.uniform(100, 500,
size=num_records).round(2)
}
df = pd.DataFrame(data)

# Create the main application window
root = tk.Tk()
root.title("Insurance Data Analysis")
root.geometry("1200x800")

# Create a frame for the table
frame_table = tk.Frame(root)
frame_table.pack(fill=tk.BOTH, expand=True)

# Create a Treeview widget for displaying the
DataFrame
tree = ttk.Treeview(frame_table,
columns=list(df.columns), show='headings')
tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

# Scrollbars for the Treeview
vsb = ttk.Scrollbar(frame_table,
orient="vertical", command=tree.yview)
vsb.pack(side='right', fill='y')
tree.configure(yscrollcommand=vsb.set)
```

```
hsb = ttk.Scrollbar(frame_table,
orient="horizontal", command=tree.xview)
hsb.pack(side='bottom', fill='x')
tree.configure(xscrollcommand=hsb.set)

# Define the columns in the Treeview
for col in df.columns:
    tree.heading(col, text=col)
    tree.column(col, width=100)

# Function to update the Treeview with alternating
# row colors
def update_treeview(dataframe):
    for i in tree.get_children():
        tree.delete(i)

    # Insert rows with alternating colors
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree.insert("", "end", values=list(row),
tags=(tag,))

    # Apply styles for row colors
    tree.tag_configure('even',
background="#f9f9f9")
    tree.tag_configure('odd',
background="#eaeaea")

update_treeview(df)

# Create a frame for filters and buttons
frame_controls = tk.Frame(root)
frame_controls.pack(fill=tk.X)

# Filter options
```

```
tk.Label(frame_controls,
text="State:").pack(side=tk.LEFT, padx=5, pady=5)
state_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['State'].unique()))
state_combobox.set('All')
state_combobox.pack(side=tk.LEFT, padx=5, pady=5)

tk.Label(frame_controls, text="Coverage
Type:").pack(side=tk.LEFT, padx=5, pady=5)
coverage_combobox = ttk.Combobox(frame_controls,
values=['All'] +
list(df['Coverage_Type'].unique()))
coverage_combobox.set('All')
coverage_combobox.pack(side=tk.LEFT, padx=5,
pady=5)

# Button to apply filters
def apply_filters():
    state = state_combobox.get()
    coverage_type = coverage_combobox.get()

    filtered_df = df.copy()
    if state != 'All':
        filtered_df =
filtered_df[filtered_df['State'] == state]
    if coverage_type != 'All':
        filtered_df =
filtered_df[filtered_df['Coverage_Type'] ==
coverage_type]

    update_treeview(filtered_df)

btn_apply_filters = tk.Button(frame_controls,
text="Apply Filters", command=apply_filters)
btn_apply_filters.pack(side=tk.LEFT, padx=5,
pady=5)
```

```
# Button to save the filtered data
def save_to_excel():
    filtered_data = tree.get_children()
    if not filtered_data:
        messagebox.showwarning("Warning", "No data
to save.")
        return
    file_path =
filedialog.asksaveasfilename(defaultextension=".xl
sx", filetypes=[("Excel files", "*.xlsx")])
    if file_path:
        df_filtered =
pd.DataFrame([tree.item(item)['values'] for item
in filtered_data], columns=df.columns)
        df_filtered.to_excel(file_path,
index=False)
        messagebox.showinfo("Info", f"Data saved
to {file_path}")

btn_save = tk.Button(frame_controls, text="Save to
Excel", command=save_to_excel)
btn_save.pack(side=tk.LEFT, padx=5, pady=5)

# Create a frame for data visualization
frame_plot = tk.Frame(root)
frame_plot.pack(fill=tk.BOTH, expand=True)

# Function to create and display a plot
def plot_data():
    plt.figure(figsize=(8, 6))
    df['State'].value_counts().plot(kind='bar')
    plt.title('Number of Policies by State')
    plt.xlabel('State')
    plt.ylabel('Number of Policies')
```

```

    for widget in frame_plot.winfo_children():
        widget.destroy()

    canvas = FigureCanvasTkAgg(plt.gcf(),
master=frame_plot)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,
expand=True)

# Button to generate plot
btn_plot = tk.Button(frame_controls, text="Plot
Data", command=plot_data)
btn_plot.pack(side=tk.LEFT, padx=5, pady=5)

# Run the Tkinter event loop
root.mainloop()

```

Detailed Explanation of the Code

This Tkinter GUI application is designed to manage and visualize a synthetic insurance dataset. Here's a breakdown of each section:

1. Imports and Data Generation:

```

import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from   matplotlib.backends.backend_tkagg      import
FigureCanvasTkAgg

```

- **tkinter:** Provides the GUI components.
- **ttk:** Provides themed widgets.
- **filedialog and messagebox:** For file dialogs and displaying messages.

- pandas: Data manipulation and analysis.
- numpy: Generates synthetic data.
- matplotlib: Creates plots.
- FigureCanvasTkAgg: Integrates matplotlib plots into Tkinter.

2. Dataset Generation:

```
np.random.seed(0)
num_records = 1000
data = {
    'Policyholder': [f'Policyholder_{i}' for i in
range(num_records)],
    'Age': np.random.randint(18, 70,
size=num_records),
    'State': np.random.choice(['New York',
'California', 'Texas', 'Florida', 'Illinois'],
size=num_records),
    'Coverage_Type': np.random.choice(['Basic',
'Standard', 'Premium'], size=num_records),
    'Premium': np.random.uniform(100, 500,
size=num_records).round(2)
}
df = pd.DataFrame(data)
```

- Creates a synthetic dataset with 1,000 records. Each record includes a Policyholder ID, Age, State, Coverage_Type, and Premium.

3. Main Application Window:

```
root = tk.Tk()
root.title("Insurance Data Analysis")
```

```
root.geometry("1200x800")
```

Initializes the main Tkinter window with a title and specified size.

4. Table Frame and Treeview Widget:

```
frame_table = tk.Frame(root)
frame_table.pack(fill=tk.BOTH, expand=True)

tree = ttk.Treeview(frame_table,
columns=list(df.columns), show='headings')
tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
```

Creates a frame to hold the Treeview widget, which displays the DataFrame.

Scrollbars:

```
vsb = ttk.Scrollbar(frame_table,
orient="vertical", command=tree.yview)
vsb.pack(side='right', fill='y')
tree.configure(yscrollcommand=vsb.set)

hsb = ttk.Scrollbar(frame_table,
orient="horizontal", command=tree.xview)
hsb.pack(side='bottom', fill='x')
tree.configure(xscrollcommand=hsb.set)
```

Adds vertical and horizontal scrollbars to navigate through large datasets.

Treeview Columns:

```
for col in df.columns:
    tree.heading(col, text=col)
```

```
tree.column(col, width=100)
```

Sets up column headings and widths based on DataFrame columns.

4. Updating the Treeview with Alternating Row Colors:

```
def update_treeview(dataframe):
    for i in tree.get_children():
        tree.delete(i)

    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree.insert("", "end", values=list(row),
tags=(tag,))

        tree.tag_configure('even',
background='#f9f9f9')
        tree.tag_configure('odd',
background='#eaeaea')

update_treeview(df)
```

update_treeview(dataframe): Updates the Treeview widget with alternating row colors (#f9f9f9 for even rows and #eaeaea for odd rows).

5. Filters and Buttons:

```
frame_controls = tk.Frame(root)
frame_controls.pack(fill=tk.X)

tk.Label(frame_controls,
text="State:").pack(side=tk.LEFT, padx=5, pady=5)
state_combobox      =      ttk.Combobox(frame_controls,
values=['All'] + list(df['State'].unique()))
```

```

state_combobox.set('All')
state_combobox.pack(side=tk.LEFT, padx=5, pady=5)

tk.Label(frame_controls, text="Coverage Type:").pack(side=tk.LEFT, padx=5, pady=5)
coverage_combobox = ttk.Combobox(frame_controls, values=['All'] +
list(df['Coverage_Type'].unique()))
coverage_combobox.set('All')
coverage_combobox.pack(side=tk.LEFT, padx=5, pady=5)

```

Adds comboboxes for filtering by State and Coverage_Type.

Filter Function:

```

def apply_filters():
    state = state_combobox.get()
    coverage_type = coverage_combobox.get()

    filtered_df = df.copy()
    if state != 'All':
        filtered_df = filtered_df[filtered_df['State'] == state]
    if coverage_type != 'All':
        filtered_df = filtered_df[filtered_df['Coverage_Type'] ==
coverage_type]

    update_treeview(filtered_df)

btn_apply_filters = tk.Button(frame_controls, text="Apply Filters", command=apply_filters)
btn_apply_filters.pack(side=tk.LEFT, padx=5, pady=5)

```

`apply_filters()`: Filters the DataFrame based on selected criteria and updates the Treeview.

Save Data Function:

```
def save_to_excel():
    filtered_data = tree.get_children()
    if not filtered_data:
        messagebox.showwarning("Warning", "No data
to save.")
        return
    file_path =
filedialog.asksaveasfilename(defaultextension=".xl
sx", filetypes=[("Excel files", "*.xlsx")])
    if file_path:
        df_filtered =
pd.DataFrame([tree.item(item)['values'] for item
in filtered_data], columns=df.columns)
        df_filtered.to_excel(file_path,
index=False)
        messagebox.showinfo("Info", f"Data saved
to {file_path}")

btn_save = tk.Button(frame_controls, text="Save to
Excel", command=save_to_excel)
btn_save.pack(side=tk.LEFT, padx=5, pady=5)
```

`save_to_excel()`: Saves the currently displayed data to an Excel file. Displays a file dialog for selecting the save location.

6. Data Visualization:

```
frame_plot = tk.Frame(root)
frame_plot.pack(fill=tk.BOTH, expand=True)
```

```

def plot_data():
    plt.figure(figsize=(8, 6))
    df['State'].value_counts().plot(kind='bar')
    plt.title('Number of Policies by State')
    plt.xlabel('State')
    plt.ylabel('Number of Policies')

    for widget in frame_plot.winfo_children():
        widget.destroy()

    canvas = FigureCanvasTkAgg(plt.gcf(),
master=frame_plot)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,
expand=True)

btn_plot = tk.Button(frame_controls, text="Plot
Data", command=plot_data)
btn_plot.pack(side=tk.LEFT, padx=5, pady=5)

```

`plot_data()`: Creates a bar plot showing the number of policies by state and embeds it in the Tkinter window using `FigureCanvasTkAgg`.

7. Run the Tkinter Event Loop:

```
root.mainloop()
```

Starts the Tkinter event loop, displaying the GUI and handling user interactions.

Summary

This script creates a comprehensive Tkinter GUI that allows users to view, filter, and analyze a synthetic insurance dataset. The

features include a scrollable table with alternating row colors, filtering options, saving functionality, and data visualization with embedded plots.

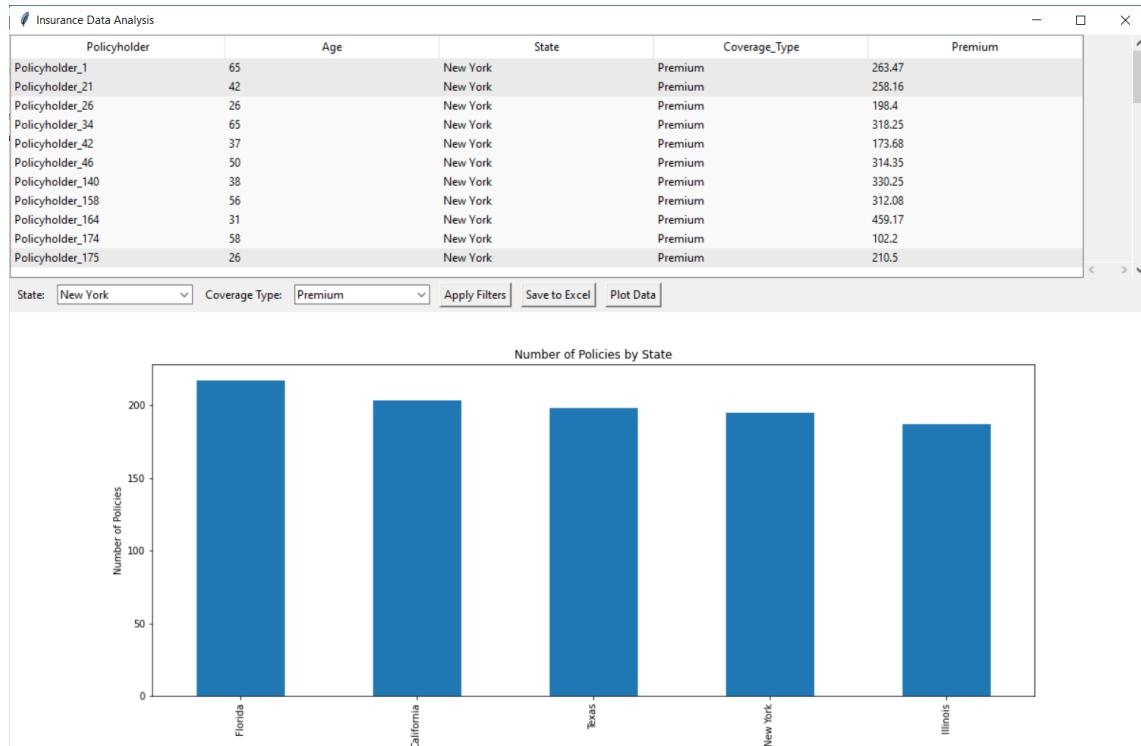
| Insurance Data Analysis | | | | | |
|-------------------------|-----|----------------|---------------|---------------|---------------|
| Policyholder | Age | State | Coverage_Type | Premium | |
| Policyholder_0 | 62 | New York | Standard | 385.35 | |
| Policyholder_1 | 65 | New York | Premium | 263.47 | |
| Policyholder_2 | 18 | Florida | Basic | 307.37 | |
| Policyholder_3 | 21 | Illinois | Standard | 366.07 | |
| Policyholder_4 | 21 | Illinois | Premium | 165.92 | |
| Policyholder_5 | 57 | Texas | Standard | 110.88 | |
| Policyholder_6 | 27 | Texas | Basic | 227.0 | |
| Policyholder_7 | 37 | Florida | Premium | 338.23 | |
| Policyholder_8 | 39 | Florida | Standard | 294.64 | |
| Policyholder_9 | 68 | Illinois | Standard | 377.02 | |
| Policyholder_10 | 54 | Texas | Basic | 427.88 | |
| Policyholder_11 | 41 | California | Basic | 295.38 | |
| Policyholder_12 | 24 | New York | Basic | 153.71 | |
| Policyholder_13 | 42 | Illinois | Premium | 440.25 | |
| Policyholder_14 | 42 | California | Premium | 330.0 | |
| Policyholder_15 | 30 | Florida | Standard | 395.97 | |
| Policyholder_16 | 19 | New York | Standard | 381.87 | |
| Policyholder_17 | 56 | Florida | Premium | 487.28 | |
| Policyholder_18 | 57 | Texas | Standard | 218.12 | |
| Policyholder_19 | 41 | California | Premium | 382.12 | |
| Policyholder_20 | 64 | New York | Standard | 246.27 | |
| Policyholder_21 | 42 | New York | Premium | 258.16 | |
| State: | All | Coverage Type: | All | Apply Filters | Save to Excel |
| | | | | Plot Data | |

Insurance Data Analysis

| Policyholder | Age | State | Coverage_Type | Premium |
|------------------|-----|----------|---------------|---------|
| Policyholder_0 | 62 | New York | Standard | 385.35 |
| Policyholder_1 | 65 | New York | Premium | 263.47 |
| Policyholder_12 | 24 | New York | Basic | 153.71 |
| Policyholder_16 | 19 | New York | Standard | 381.87 |
| Policyholder_20 | 64 | New York | Standard | 246.27 |
| Policyholder_21 | 42 | New York | Premium | 258.16 |
| Policyholder_25 | 31 | New York | Basic | 217.03 |
| Policyholder_26 | 26 | New York | Premium | 198.4 |
| Policyholder_30 | 34 | New York | Basic | 433.67 |
| Policyholder_33 | 33 | New York | Standard | 344.2 |
| Policyholder_34 | 65 | New York | Premium | 318.25 |
| Policyholder_41 | 47 | New York | Standard | 117.54 |
| Policyholder_42 | 37 | New York | Premium | 173.68 |
| Policyholder_46 | 50 | New York | Premium | 314.35 |
| Policyholder_53 | 53 | New York | Standard | 111.71 |
| Policyholder_60 | 54 | New York | Basic | 246.4 |
| Policyholder_83 | 30 | New York | Standard | 271.28 |
| Policyholder_89 | 22 | New York | Standard | 199.31 |
| Policyholder_91 | 21 | New York | Standard | 227.36 |
| Policyholder_107 | 53 | New York | Basic | 389.45 |
| Policyholder_110 | 23 | New York | Basic | 443.81 |
| Policyholder_122 | 31 | New York | Standard | 334.02 |

Insurance Data Analysis

| Policyholder | Age | State | Coverage_Type | Premium |
|------------------|-----|----------|---------------|---------|
| Policyholder_1 | 65 | New York | Premium | 263.47 |
| Policyholder_21 | 42 | New York | Premium | 258.16 |
| Policyholder_26 | 26 | New York | Premium | 198.4 |
| Policyholder_34 | 65 | New York | Premium | 318.25 |
| Policyholder_42 | 37 | New York | Premium | 173.68 |
| Policyholder_46 | 50 | New York | Premium | 314.35 |
| Policyholder_140 | 38 | New York | Premium | 330.25 |
| Policyholder_158 | 56 | New York | Premium | 312.08 |
| Policyholder_164 | 31 | New York | Premium | 459.17 |
| Policyholder_174 | 58 | New York | Premium | 102.2 |
| Policyholder_175 | 26 | New York | Premium | 210.5 |
| Policyholder_178 | 26 | New York | Premium | 486.8 |
| Policyholder_216 | 29 | New York | Premium | 355.23 |
| Policyholder_244 | 46 | New York | Premium | 136.39 |
| Policyholder_251 | 41 | New York | Premium | 388.86 |
| Policyholder_261 | 59 | New York | Premium | 300.21 |
| Policyholder_268 | 21 | New York | Premium | 139.3 |
| Policyholder_273 | 63 | New York | Premium | 350.17 |
| Policyholder_285 | 54 | New York | Premium | 450.67 |
| Policyholder_289 | 23 | New York | Premium | 279.94 |
| Policyholder_293 | 35 | New York | Premium | 209.34 |
| Policyholder_327 | 35 | New York | Premium | 175.36 |



EXAMPLE 4.3

Grouping Aggregating with Synthetic Sales Data

This project is designed to create, analyze, and aggregate a large synthetic sales dataset for business intelligence purposes. By generating a dataset with 10,000 records that includes information on salespersons, regions, products, sales amounts, and transaction timestamps, the project simulates a comprehensive sales environment. The increased number of products and the hourly frequency of transactions enhance the realism and granularity of the data, providing a more detailed view of sales operations. The dataset includes multiple products, regions, and salespersons to reflect a diverse sales landscape.

The core objective of the project is to aggregate the sales data to analyze performance metrics. By grouping the data by region, product, and salesperson, and then calculating the total sales amount and the number of transactions for each group, the project generates valuable insights into sales patterns and performance.

This aggregated data is saved to an Excel file, making it accessible for further analysis and reporting. The output helps businesses understand sales trends, evaluate the performance of salespersons, and make data-driven decisions to optimize sales strategies.

```
import pandas as pd
import numpy as np

# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000 # Larger dataset

# Define a larger set of products
num_products = 50 # Increase the number of products
products = [f'Product_{i}' for i in range(num_products)]

# Create synthetic sales dataset
data = {
    'Salesperson': [f'Salesperson_{i % 50}' for i in range(num_records)], # Increased range
    'Region': np.random.choice(['North', 'South', 'East', 'West'], size=num_records),
    'Product': np.random.choice(products, size=num_records), # Use the larger set of products
    'Sales_Amount': np.random.uniform(50, 500, size=num_records).round(2),
    'Date': pd.date_range(start='2024-01-01', periods=num_records, freq='H') # Using hourly frequency
}
```

```
df = pd.DataFrame(data)

# Group by Region, Product, and Salesperson, then
# aggregate sales amount and count of sales
grouped_df = df.groupby(['Region', 'Product',
'Salesperson']).agg({
    'Sales_Amount': 'sum',
    'Salesperson': 'count' # Count the number of
sales (transactions) per salesperson
}).rename(columns={'Sales_Amount': 'Total_Sales',
'Salesperson': 'Number_of_Sales'}).reset_index()

# Save the aggregated DataFrame to an Excel file
output_file =
'sales_per_salesperson_larger_products.xlsx'
grouped_df.to_excel(output_file, sheet_name='Sales
Per Salesperson', index=False)

print(f"Data saved to {output_file}")
```

Here's a detailed explanation of each part of the code:

1. Imports and Seed Initialization

```
import pandas as pd
import numpy as np
```

- import pandas as pd: Imports the pandas library, which provides data structures and data analysis tools for Python, including DataFrames.
- import numpy as np: Imports the numpy library, which supports numerical operations and provides functions for generating random numbers.

```
np.random.seed(0)
```

- np.random.seed(0): Sets the random seed to 0 to ensure reproducibility of the random numbers generated by numpy. This means that every time you run the code, the random numbers will be the same, making your results consistent.

2. Define Dataset Parameters

```
num_records = 10000 # Larger dataset
```

- num_records = 10000: Specifies the number of records (rows) in the dataset. This creates a larger dataset with 10,000 entries.

```
num_products = 50 # Increase the number of products
products = [f'Product_{i}' for i in range(num_products)]
```

- num_products = 50: Sets the number of unique products to 50, expanding the variety of products in the dataset.
- products = [f'Product_{i}' for i in range(num_products)]: Creates a list of product names from Product_0 to Product_49 using a list comprehension.

3. Generate Synthetic Sales Dataset

```
data = {
    'Salesperson': [f'Salesperson_{i % 50}' for i
in range(num_records)], # Increased range
```

```

        'Region': np.random.choice(['North', 'South',
'East', 'West'], size=num_records),
        'Product': np.random.choice(products,
size=num_records),      # Use the larger set of
products
        'Sales_Amount': np.random.uniform(50, 500,
size=num_records).round(2),
        'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H')      # Using hourly
frequency
}

```

- 'Salesperson': [f'Salesperson_{i % 50}' for i in range(num_records)]: Assigns each record a salesperson from Salesperson_0 to Salesperson_49, repeating as needed to cover all records. This simulates having multiple salespersons, but with repetition to match the total record count.
- 'Region': np.random.choice(['North', 'South', 'East', 'West'], size=num_records): Randomly assigns one of four regions to each record.
- 'Product': np.random.choice(products, size=num_records): Randomly assigns one of the 50 products to each record.
- 'Sales_Amount': np.random.uniform(50, 500, size=num_records).round(2): Generates random sales amounts between \$50 and \$500, rounding to two decimal places.
- 'Date': pd.date_range(start='2024-01-01', periods=num_records, freq='H'): Creates a date range starting from January 1, 2024, with hourly frequency, providing timestamps for each record.

```
df = pd.DataFrame(data)
```

- df = pd.DataFrame(data): Converts the data dictionary into a pandas DataFrame, which is a tabular data structure.

4. Group and Aggregate Data

```
grouped_df = df.groupby(['Region', 'Product',  
'Salesperson']).agg({  
    'Sales_Amount': 'sum',  
    'Salesperson': 'count' # Count the number of  
    sales (transactions) per salesperson  
}).rename(columns={'Sales_Amount': 'Total_Sales',  
    'Salesperson': 'Number_of_Sales'}).reset_index()
```

- df.groupby(['Region', 'Product', 'Salesperson']): Groups the DataFrame by Region, Product, and Salesperson.
- .agg({'Sales_Amount': 'sum', 'Salesperson': 'count'}): Aggregates the grouped data by:
 - 'Sales_Amount': 'sum': Summing the Sales_Amount for each group.
 - 'Salesperson': 'count': Counting the number of transactions (sales) for each salesperson in each group.
- .rename(columns={'Sales_Amount': 'Total_Sales', 'Salesperson': 'Number_of_Sales'}): Renames columns for clarity:
 - 'Sales_Amount' to 'Total_Sales'
 - 'Salesperson' to 'Number_of_Sales'

- `.reset_index()`: Resets the index of the resulting DataFrame, converting the grouped columns into regular columns.

5. Save to Excel

```
output_file =  
'sales_per_salesperson_larger_products.xlsx'  
grouped_df.to_excel(output_file, sheet_name='Sales  
Per Salesperson', index=False)
```

- `output_file` =
`'sales_per_salesperson_larger_products.xlsx'`: Specifies the file name for the Excel output.
- `grouped_df.to_excel(output_file, sheet_name='Sales Per Salesperson', index=False)`: Saves the aggregated DataFrame to an Excel file with the specified sheet name Sales Per Salesperson, without including DataFrame indices.

```
print(f"Data saved to {output_file}")
```

- `print(f"Data saved to {output_file}")`: Prints a message to confirm that the data has been saved successfully.

This code provides a comprehensive dataset with total sales and transaction counts per salesperson, region, and product, and saves the result to an Excel file for further analysis.

EXAMPLE 4.4

GUI Tkinter for Grouping Aggregating with Synthetic Sales Data

The purpose of this project is to develop a comprehensive graphical user interface (GUI) for analyzing and visualizing synthetic sales data. By generating a large dataset with various attributes, such as salesperson, region, product, sales amount, and date, the project aims to provide a robust platform for users to interact with and gain insights from this data. The GUI facilitates the exploration of raw and aggregated data, allowing users to filter and sort the information dynamically, which enhances their ability to perform detailed analyses and make data-driven decisions.

The application features two main views: a raw data table and an aggregated data table. The raw data table displays the complete dataset with the option to scroll vertically and horizontally, while the aggregated data table summarizes the data by grouping it according to region, product, and salesperson. This dual-view approach enables users to compare raw and summarized information easily, providing a clearer understanding of the data's structure and trends. Additionally, the GUI includes functionalities for applying filters based on region, product, and salesperson, which helps users narrow down the data to focus on specific aspects of interest.

Beyond data presentation, the project integrates visualization capabilities to enhance the analytical experience. It includes a button to generate plots that graphically represent the data, such as the number of sales by region. This feature allows users to visually interpret trends and patterns, making complex data more accessible and understandable. Furthermore, the GUI supports exporting filtered data to Excel, providing users with a convenient way to save and share their findings. Overall, the project aims to offer a powerful and user-friendly tool for data analysis, visualization, and reporting.

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000 # Larger dataset

# Define a larger set of products
num_products = 50 # Increase the number of
products
products = [f'Product_{i}' for i in
range(num_products)]

# Create synthetic sales dataset
data = {
    'Salesperson': [f'Salesperson_{i % 50}' for i
in range(num_records)], # Increased range
    'Region': np.random.choice(['North', 'South',
'East', 'West'], size=num_records),
    'Product': np.random.choice(products,
size=num_records), # Use the larger set of
products
    'Sales_Amount': np.random.uniform(50, 500,
size=num_records).round(2),
    'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H') # Using hourly
frequency
}
```

```
df = pd.DataFrame(data)

# Group by Region, Product, and Salesperson, then
# aggregate sales amount and count of sales
grouped_df = df.groupby(['Region', 'Product',
'Salesperson']).agg({
    'Sales_Amount': 'sum',
    'Salesperson': 'count' # Count the number of
sales (transactions) per salesperson
}).rename(columns={'Sales_Amount': 'Total_Sales',
'Salesperson': 'Number_of_Sales'}).reset_index()

# Create the main application window
root = tk.Tk()
root.title("Sales Data Analysis")
root.geometry("1200x800")

# Create a frame for the raw data table
frame_raw_data = tk.Frame(root)
frame_raw_data.pack(fill=tk.BOTH, expand=True)

# Create a Treeview widget for displaying the raw
# DataFrame
tree_raw = ttk.Treeview(frame_raw_data,
columns=list(df.columns), show='headings')
tree_raw.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Scrollbars for the raw data Treeview
vsb_raw = ttk.Scrollbar(frame_raw_data,
orient="vertical", command=tree_raw.yview)
vsb_raw.pack(side='right', fill='y')
tree_raw.configure(yscrollcommand=vsb_raw.set)
```

```
hsb_raw = ttk.Scrollbar(frame_raw_data,
orient="horizontal", command=tree_raw.xview)
hsb_raw.pack(side='bottom', fill='x')
tree_raw.configure(xscrollcommand=hsb_raw.set)

# Define the columns in the raw data Treeview
for col in df.columns:
    tree_raw.heading(col, text=col)
    tree_raw.column(col, width=100)

# Function to update the raw data Treeview with
# alternating row colors
def update_raw_treeview(dataframe):
    for i in tree_raw.get_children():
        tree_raw.delete(i)

    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree_raw.insert("", "end",
values=list(row), tags=(tag,))

    # Apply styles for row colors
    tree_raw.tag_configure('even',
background='#f5f5f5') # Light grey
    tree_raw.tag_configure('odd',
background='#ffffff') # White

update_raw_treeview(df)

# Create a frame for the aggregated data table
frame_grouped_data = tk.Frame(root)
frame_grouped_data.pack(fill=tk.BOTH, expand=True)

# Create a Treeview widget for displaying the
# grouped DataFrame
```

```
tree_grouped = ttk.Treeview(frame_grouped_data,
columns=list(grouped_df.columns), show='headings')
tree_grouped.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Scrollbars for the grouped data Treeview
vsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="vertical", command=tree_grouped.yview)
vsb_grouped.pack(side='right', fill='y')
tree_grouped.configure(yscrollcommand=vsb_grouped.
set)

hsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="horizontal", command=tree_grouped.xview)
hsb_grouped.pack(side='bottom', fill='x')
tree_grouped.configure(xscrollcommand=hsb_grouped.
set)

# Define the columns in the grouped data Treeview
for col in grouped_df.columns:
    tree_grouped.heading(col, text=col)
    tree_grouped.column(col, width=120)

# Function to update the grouped data Treeview
# with alternating row colors
def update_grouped_treeview(dataframe):
    for i in tree_grouped.get_children():
        tree_grouped.delete(i)

    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree_grouped.insert("", "end",
values=list(row), tags=(tag,))

    # Apply styles for row colors
```

```
    tree_grouped.tag_configure('even',
background='#f5f5f5') # Light grey
    tree_grouped.tag_configure('odd',
background='#ffffff') # White

update_grouped_treeview(grouped_df)

# Create a frame for filters and buttons
frame_controls = tk.Frame(root)
frame_controls.pack(fill=tk.X)

# Filter options
tk.Label(frame_controls,
text="Region:").pack(side=tk.LEFT, padx=5, pady=5)
region_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Region'].unique()))
region_combobox.set('All')
region_combobox.pack(side=tk.LEFT, padx=5, pady=5)

tk.Label(frame_controls,
text="Product:").pack(side=tk.LEFT, padx=5,
pady=5)
product_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Product'].unique()))
product_combobox.set('All')
product_combobox.pack(side=tk.LEFT, padx=5,
pady=5)

tk.Label(frame_controls,
text="Salesperson:").pack(side=tk.LEFT, padx=5,
pady=5)
salesperson_combobox =
ttk.Combobox(frame_controls, values=['All'] +
list(df['Salesperson'].unique()))
salesperson_combobox.set('All')
```

```
salesperson_combobox.pack(side=tk.LEFT, padx=5,
pady=5)

# Button to apply filters
def apply_filters():
    region = region_combobox.get()
    product = product_combobox.get()
    salesperson = salesperson_combobox.get()

    filtered_df = df.copy()
    if region != 'All':
        filtered_df =
filtered_df[filtered_df['Region'] == region]
    if product != 'All':
        filtered_df =
filtered_df[filtered_df['Product'] == product]
    if salesperson != 'All':
        filtered_df =
filtered_df[filtered_df['Salesperson'] ==
salesperson]

    update_raw_treeview(filtered_df)

    # Group by Region, Product, and Salesperson,
    # then aggregate sales amount and count of sales
    grouped_filtered_df =
filtered_df.groupby(['Region', 'Product',
'Salesperson']).agg({
    'Sales_Amount': 'sum',
    'Salesperson': 'count'
}).rename(columns={'Sales_Amount':
'Total_Sales', 'Salesperson':
'Number_of_Sales'}).reset_index()

    update_grouped_treeview(grouped_filtered_df)
```

```
btn_apply_filters = tk.Button(frame_controls,
text="Apply Filters", command=apply_filters)
btn_apply_filters.pack(side=tk.LEFT, padx=5,
pady=5)

# Button to save the filtered data
def save_to_excel():
    filtered_data = tree_raw.get_children()
    if not filtered_data:
        messagebox.showwarning("Warning", "No data
to save.")
        return
    file_path =
filedialog.asksaveasfilename(defaultextension=".xl
sx", filetypes=[("Excel files", "*.xlsx")])
    if file_path:
        df_filtered =
pd.DataFrame([tree_raw.item(item)['values'] for
item in filtered_data], columns=df.columns)
        df_filtered.to_excel(file_path,
index=False)
        messagebox.showinfo("Info", f"Data saved
to {file_path}")

btn_save = tk.Button(frame_controls, text="Save to
Excel", command=save_to_excel)
btn_save.pack(side=tk.LEFT, padx=5, pady=5)

# Create a frame for data visualization
frame_plot = tk.Frame(root)
frame_plot.pack(fill=tk.BOTH, expand=True)

# Function to create and display a plot
def plot_data():
    plt.figure(figsize=(10, 6))
    df['Region'].value_counts().plot(kind='bar')
```

```

plt.title('Number of Sales by Region')
plt.xlabel('Region')
plt.ylabel('Number of Sales')

for widget in frame_plot.winfo_children():
    widget.destroy()

canvas = FigureCanvasTkAgg(plt.gcf(),
master=frame_plot)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,
expand=True)

# Button to generate plot
btn_plot = tk.Button(frame_controls, text="Plot
Data", command=plot_data)
btn_plot.pack(side=tk.LEFT, padx=5, pady=5)

# Run the Tkinter event loop
root.mainloop()

```

This Tkinter GUI application is designed for interactive exploration and analysis of synthetic sales data. It provides features for viewing raw and aggregated data, filtering the data, saving the results, and visualizing the data with plots. Here is a detailed explanation of the code:

Imports

```

import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

```

```
from matplotlib.backends.backend_tkagg import  
FigureCanvasTkAgg
```

- tkinter: Standard Python library for creating graphical user interfaces (GUIs).
- ttk: Provides access to Tk themed widgets, which offer a more modern look.
- filedialog: Module for file dialogs, allowing users to open or save files.
- messagebox: Module for displaying message boxes.
- pandas: Data manipulation and analysis library.
- numpy: Library for numerical operations, including generating random data.
- matplotlib.pyplot: Plotting library used for creating visualizations.
- FigureCanvasTkAgg: Matplotlib's canvas for embedding plots in Tkinter applications.

Dataset Creation

```
np.random.seed(0)  
num_records = 10000  
num_products = 50  
products = [f'Product_{i}' for i in range(num_products)]  
  
data = {  
    'Salesperson': [f'Salesperson_{i % 50}' for i in range(num_records)],  
    'Region': np.random.choice(['North', 'South', 'East', 'West'], size=num_records),  
    'Product': np.random.choice(products, size=num_records),
```

```

        'Sales_Amount': np.random.uniform(50, 500,
size=num_records).round(2),
        'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H')
}

df = pd.DataFrame(data)

```

- np.random.seed(0): Sets the seed for random number generation to ensure reproducibility.
- num_records: Specifies the number of rows in the dataset.
- num_products: Defines the number of unique products.
- products: Generates a list of product names.
- data: Creates a dictionary with synthetic data, including salespersons, regions, products, sales amounts, and timestamps.
- df: Converts the dictionary into a Pandas DataFrame.

Data Aggregation

```

grouped_df = df.groupby(['Region', 'Product',
'Salesperson']).agg({
    'Sales_Amount': 'sum',
    'Salesperson': 'count'
}).rename(columns={'Sales_Amount': 'Total_Sales',
'Salesperson': 'Number_of_Sales'}).reset_index()

```

- groupby: Groups the data by Region, Product, and Salesperson.
- agg: Aggregates data to calculate the total sales amount and count of transactions per salesperson.
- rename: Renames columns for clarity.

- `reset_index`: Resets the index to convert grouped data into a standard DataFrame.

Tkinter GUI Setup

```
root = tk.Tk()
root.title("Sales Data Analysis")
root.geometry("1200x800")
```

- `root`: Creates the main application window.
- `title`: Sets the window title.
- `geometry`: Specifies the window size.

Raw Data Table

```
frame_raw_data = tk.Frame(root)
frame_raw_data.pack(fill=tk.BOTH, expand=True)

tree_raw      =      ttk.Treeview(frame_raw_data,
columns=list(df.columns), show='headings')
tree_raw.pack(side=tk.LEFT,           fill=tk.BOTH,
expand=True)

vsb_raw       =      ttk.Scrollbar(frame_raw_data,
orient="vertical", command=tree_raw.yview)
vsb_raw.pack(side='right', fill='y')
tree_raw.configure(yscrollcommand=vsb_raw.set)

hsb_raw       =      ttk.Scrollbar(frame_raw_data,
orient="horizontal", command=tree_raw.xview)
hsb_raw.pack(side='bottom', fill='x')
tree_raw.configure(xscrollcommand=hsb_raw.set)

for col in df.columns:
```

```
tree_raw.heading(col, text=col)
tree_raw.column(col, width=100)
```

- frame_raw_data: A frame to hold the Treeview widget for displaying raw data.
- tree_raw: A Treeview widget that displays data with columns defined by the DataFrame.
- vsb_raw and hsb_raw: Vertical and horizontal scrollbars for navigating large datasets.
- for col in df.columns: Sets column headings and widths.

Function to Update Raw Data Treeview

```
def update_raw_treeview(dataframe):
    for i in tree_raw.get_children():
        tree_raw.delete(i)

    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
                    tree_raw.insert("", "end",
values=list(row), tags=(tag,))

                    tree_raw.tag_configure('even',
background='#f5f5f5')
                    tree_raw.tag_configure('odd',
background='#ffffff')
```

- update_raw_treeview: Updates the Treeview with alternating row colors.
- tags: Assigns tags to rows for alternating colors.
- tag_configure: Configures the styles for the tags.

Aggregated Data Table

```

frame_grouped_data = tk.Frame(root)
frame_grouped_data.pack(fill=tk.BOTH, expand=True)

tree_grouped = ttk.Treeview(frame_grouped_data,
columns=list(grouped_df.columns), show='headings')
tree_grouped.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

vsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="vertical", command=tree_grouped.yview)
vsb_grouped.pack(side='right', fill='y')
tree_grouped.configure(yscrollcommand=vsb_grouped.
set)

hsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="horizontal", command=tree_grouped.xview)
hsb_grouped.pack(side='bottom', fill='x')
tree_grouped.configure(xscrollcommand=hsb_grouped.
set)

for col in grouped_df.columns:
    tree_grouped.heading(col, text=col)
    tree_grouped.column(col, width=120)

```

- frame_grouped_data: A frame to hold the Treeview widget for displaying aggregated data.
- tree_grouped: A Treeview widget similar to tree_raw but for aggregated data.

Function to Update Grouped Data Treeview

```

def update_grouped_treeview(dataframe):
    for i in tree_grouped.get_children():
        tree_grouped.delete(i)

```

```

        for index, row in dataframe.iterrows():
            tag = 'even' if index % 2 == 0 else 'odd'
            tree_grouped.insert("", "end",
values=list(row), tags=(tag,))

            tree_grouped.tag_configure('even',
background="#f5f5f5")
            tree_grouped.tag_configure('odd',
background="#ffffff")

```

- `update_grouped_treeview`: Updates the Treeview for grouped data with alternating row colors.

Filter Controls

```

frame_controls = tk.Frame(root)
frame_controls.pack(fill=tk.X)

tk.Label(frame_controls,
text="Region:").pack(side=tk.LEFT, padx=5, pady=5)
region_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Region'].unique()))
region_combobox.set('All')
region_combobox.pack(side=tk.LEFT, padx=5, pady=5)

tk.Label(frame_controls,
text="Product:").pack(side=tk.LEFT,           padx=5,
pady=5)
product_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Product'].unique()))
product_combobox.set('All')
product_combobox.pack(side=tk.LEFT,           padx=5,
pady=5)

```

```
tk.Label(frame_controls,  
text="Salesperson:").pack(side=tk.LEFT,      padx=5,  
pady=5)  
salesperson_combobox =  
ttk.Combobox(frame_controls,    values=['All'] +  
list(df['Salesperson'].unique()))  
salesperson_combobox.set('All')  
salesperson_combobox.pack(side=tk.LEFT,      padx=5,  
pady=5)
```

- frame_controls: Creates a frame to contain filter controls such as labels and comboboxes.
- tk.Label(frame_controls, text="Region:"): Adds a label for the region filter.
- region_combobox: A combobox (dropdown menu) to select a region filter, initialized with all unique region values plus an "All" option.
- product_combobox: A combobox to select a product filter, initialized with all unique product values plus an "All" option.
- salesperson_combobox: A combobox to select a salesperson filter, initialized with all unique salesperson names plus an "All" option.

Sales Data Analysis

| Salesperson | Region | Product | Sales_Amount | Date |
|----------------|-----------|----------------|-------------------|---------------------|
| Salesperson_0 | North | Product_43 | 78.54 | 2024-01-01 00:00:00 |
| Salesperson_1 | West | Product_9 | 370.98 | 2024-01-01 01:00:00 |
| Salesperson_2 | South | Product_35 | 290.44 | 2024-01-01 02:00:00 |
| Salesperson_3 | North | Product_14 | 488.11 | 2024-01-01 03:00:00 |
| Salesperson_4 | West | Product_32 | 442.01 | 2024-01-01 04:00:00 |
| Salesperson_5 | West | Product_26 | 205.45 | 2024-01-01 05:00:00 |
| Salesperson_6 | West | Product_12 | 217.61 | 2024-01-01 06:00:00 |
| Salesperson_7 | West | Product_45 | 347.13 | 2024-01-01 07:00:00 |
| Salesperson_8 | South | Product_14 | 256.3 | 2024-01-01 08:00:00 |
| Salesperson_9 | West | Product_19 | 299.58 | 2024-01-01 09:00:00 |
| Salesperson_10 | South | Product_29 | 486.93 | 2024-01-01 10:00:00 |
| Salesperson_11 | East | Product_17 | 90.8 | 2024-01-01 11:00:00 |
| Salesperson_12 | North | Product_6 | 317.65 | 2024-01-01 12:00:00 |
| Salesperson_13 | West | Product_47 | 354.92 | 2024-01-01 13:00:00 |
| Region | Product | Salesperson | Total_Sales | Number_of_Sales |
| East | Product_0 | Salesperson_10 | 766.86 | 2 |
| East | Product_0 | Salesperson_12 | 246.38 | 1 |
| East | Product_0 | Salesperson_17 | 470.27 | 1 |
| East | Product_0 | Salesperson_18 | 1147.929999999998 | 3 |
| East | Product_0 | Salesperson_19 | 848.050000000001 | 3 |
| East | Product_0 | Salesperson_2 | 286.15 | 1 |
| East | Product_0 | Salesperson_20 | 481.71 | 1 |
| East | Product_0 | Salesperson_21 | 732.819999999999 | 2 |
| East | Product_0 | Salesperson_24 | 491.94 | 1 |
| East | Product_0 | Salesperson_26 | 485.599999999997 | 2 |
| East | Product_0 | Salesperson_27 | 339.39 | 1 |
| East | Product_0 | Salesperson_32 | 72.94 | 1 |
| East | Product_0 | Salesperson_36 | 594.47 | 2 |
| East | Product_0 | Salesperson_37 | 636.71 | 2 |

Region: All Product: All Salesperson: All Apply Filters Save to Excel Plot Data

Sales Data Analysis

| Salesperson | Region | Product | Sales_Amount | Date |
|----------------|-----------|----------------|------------------|---------------------|
| Salesperson_2 | South | Product_35 | 290.44 | 2024-01-01 02:00:00 |
| Salesperson_8 | South | Product_14 | 256.3 | 2024-01-01 08:00:00 |
| Salesperson_10 | South | Product_29 | 486.93 | 2024-01-01 10:00:00 |
| Salesperson_19 | South | Product_22 | 108.62 | 2024-01-01 19:00:00 |
| Salesperson_25 | South | Product_3 | 74.13 | 2024-01-02 01:00:00 |
| Salesperson_26 | South | Product_4 | 194.67 | 2024-01-02 02:00:00 |
| Salesperson_27 | South | Product_30 | 87.31 | 2024-01-02 03:00:00 |
| Salesperson_28 | South | Product_5 | 392.14 | 2024-01-02 04:00:00 |
| Salesperson_30 | South | Product_13 | 233.22 | 2024-01-02 06:00:00 |
| Salesperson_35 | South | Product_30 | 267.17 | 2024-01-02 11:00:00 |
| Salesperson_43 | South | Product_46 | 85.79 | 2024-01-02 19:00:00 |
| Salesperson_45 | South | Product_7 | 487.14 | 2024-01-02 21:00:00 |
| Salesperson_1 | South | Product_31 | 253.73 | 2024-01-03 03:00:00 |
| Salesperson_2 | South | Product_3 | 125.24 | 2024-01-03 04:00:00 |
| Region | Product | Salesperson | Total_Sales | Number_of_Sales |
| South | Product_0 | Salesperson_0 | 374.43 | 1 |
| South | Product_0 | Salesperson_1 | 685.65 | 3 |
| South | Product_0 | Salesperson_10 | 468.49 | 1 |
| South | Product_0 | Salesperson_11 | 390.67 | 2 |
| South | Product_0 | Salesperson_14 | 400.8 | 1 |
| South | Product_0 | Salesperson_17 | 423.12 | 1 |
| South | Product_0 | Salesperson_2 | 489.47 | 2 |
| South | Product_0 | Salesperson_20 | 290.08 | 1 |
| South | Product_0 | Salesperson_25 | 586.880000000001 | 2 |
| South | Product_0 | Salesperson_26 | 391.679999999995 | 2 |
| South | Product_0 | Salesperson_27 | 237.35 | 1 |
| South | Product_0 | Salesperson_28 | 402.45 | 1 |
| South | Product_0 | Salesperson_29 | 90.37 | 1 |
| South | Product_0 | Salesperson_3 | 353.14 | 1 |

Region: South Product: All Salesperson: All Apply Filters Save to Excel Plot Data

Sales Data Analysis

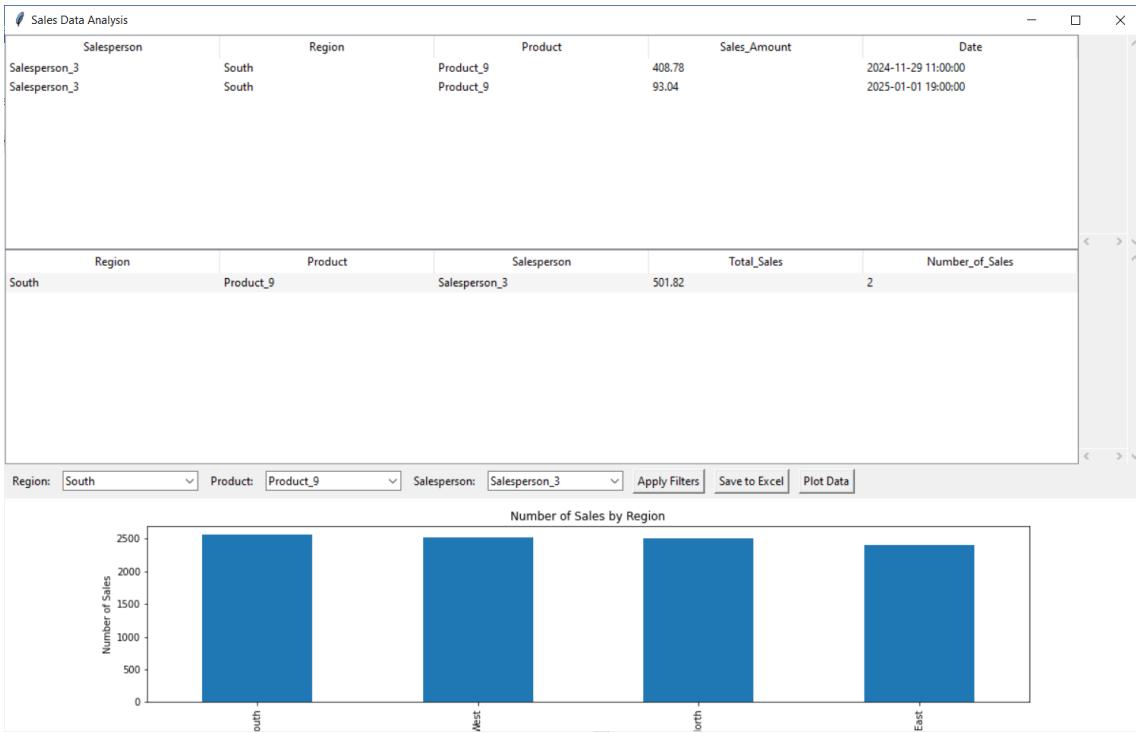
| Salesperson | Region | Product | Sales_Amount | Date |
|----------------|-----------|----------------|------------------|---------------------|
| Salesperson_21 | South | Product_9 | 196.25 | 2024-01-03 23:00:00 |
| Salesperson_19 | South | Product_9 | 496.49 | 2024-01-10 03:00:00 |
| Salesperson_26 | South | Product_9 | 249.96 | 2024-01-16 16:00:00 |
| Salesperson_34 | South | Product_9 | 294.13 | 2024-01-17 00:00:00 |
| Salesperson_44 | South | Product_9 | 410.33 | 2024-01-21 14:00:00 |
| Salesperson_5 | South | Product_9 | 90.96 | 2024-01-22 01:00:00 |
| Salesperson_41 | South | Product_9 | 97.35 | 2024-01-23 13:00:00 |
| Salesperson_23 | South | Product_9 | 408.58 | 2024-01-31 03:00:00 |
| Salesperson_48 | South | Product_9 | 299.74 | 2024-02-11 14:00:00 |
| Salesperson_49 | South | Product_9 | 132.2 | 2024-02-19 23:00:00 |
| Salesperson_8 | South | Product_9 | 421.44 | 2024-03-14 06:00:00 |
| Salesperson_24 | South | Product_9 | 91.29 | 2024-03-23 06:00:00 |
| Salesperson_45 | South | Product_9 | 393.92 | 2024-03-24 03:00:00 |
| Salesperson_28 | South | Product_9 | 326.3 | 2024-03-25 12:00:00 |
| Region | Product | Salesperson | Total_Sales | Number_of_Sales |
| South | Product_9 | Salesperson_0 | 50.46 | 1 |
| South | Product_9 | Salesperson_10 | 722.94 | 2 |
| South | Product_9 | Salesperson_12 | 333.77 | 1 |
| South | Product_9 | Salesperson_15 | 140.27 | 1 |
| South | Product_9 | Salesperson_16 | 169.11 | 1 |
| South | Product_9 | Salesperson_18 | 121.84 | 1 |
| South | Product_9 | Salesperson_19 | 995.410000000001 | 2 |
| South | Product_9 | Salesperson_21 | 545.04 | 2 |
| South | Product_9 | Salesperson_22 | 224.96 | 1 |
| South | Product_9 | Salesperson_23 | 572.69 | 2 |
| South | Product_9 | Salesperson_24 | 304.42 | 2 |
| South | Product_9 | Salesperson_26 | 249.96 | 1 |
| South | Product_9 | Salesperson_27 | 51.03 | 1 |
| South | Product_9 | Salesperson_28 | 715.34 | 2 |

Region: South | Product: Product_9 | Salesperson: All | Apply Filters | Save to Excel | Plot Data

Sales Data Analysis

| Salesperson | Region | Product | Sales_Amount | Date |
|---------------|-----------|---------------|--------------|---------------------|
| Salesperson_3 | South | Product_9 | 408.78 | 2024-11-29 11:00:00 |
| Salesperson_3 | South | Product_9 | 93.04 | 2025-01-01 19:00:00 |
| Region | Product | Salesperson | Total_Sales | Number_of_Sales |
| South | Product_9 | Salesperson_3 | 501.82 | 2 |

Region: South | Product: Product_9 | Salesperson: Salesperson_3 | Apply Filters | Save to Excel | Plot Data



EXAMPLE 4.5

Grouping and Multiple Aggregations with Synthetic Transportation Data

The purpose of the code is to create a comprehensive synthetic dataset for analyzing transportation activities and summarize the data through various aggregations. By generating a large dataset with information on vehicles, routes, distances traveled, and durations, the code simulates a realistic transportation scenario. This allows for the examination of patterns and performance metrics in transportation operations, such as the total and average distances covered and durations spent per vehicle and route.

The code also performs data aggregation to provide meaningful summaries, which can be used for reporting or decision-making. It groups the dataset by vehicle and route, then calculates the total and average distance traveled and duration for each group. This

aggregated data is then saved to an Excel file, making it accessible for further analysis or presentation. The overall goal is to facilitate the analysis of transportation data, offering insights into vehicle usage and route efficiency.

```
import pandas as pd
import numpy as np

# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000 # Larger dataset for more
comprehensive analysis

# Define a larger set of routes and vehicles
num_routes = 50
num_vehicles = 100
routes = [f'Route_{i}' for i in range(num_routes)]
vehicles = [f'Vehicle_{i}' for i in
range(num_vehicles)]

# Create synthetic transportation dataset
data = {
    'Vehicle': np.random.choice(vehicles,
size=num_records), # Randomly assign vehicles
    'Route': np.random.choice(routes,
size=num_records), # Randomly assign routes
    'Distance_Traveled': np.random.uniform(5, 500,
size=num_records).round(2), # Random distance
values
    'Duration_Hours': np.random.uniform(0.5, 10,
size=num_records).round(2), # Random duration
values}
```

```

        'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H') # Hourly frequency
}

df = pd.DataFrame(data)

# Group by Vehicle and Route, then aggregate
distance and duration
grouped_df = df.groupby(['Vehicle',
'Route']).agg({
    'Distance_Traveled': ['sum', 'mean'], # Total
and average distance traveled
    'Duration_Hours': ['sum', 'mean'] # Total and
average duration
}).reset_index()

# Flatten the MultiIndex columns
grouped_df.columns = ['_'.join(col).strip() for
col in grouped_df.columns.values]
grouped_df.rename(columns={'Vehicle_': 'Vehicle',
'Route_': 'Route'}, inplace=True)

# Save the aggregated DataFrame to an Excel file
output_file = 'transportation_data_summary.xlsx'
grouped_df.to_excel(output_file,
sheet_name='Transportation Summary', index=False)

print(f"Data saved to {output_file}")

```

Here's a detailed explanation of each part of the code:

1. Importing Libraries

```

import pandas as pd
import numpy as np

```

- import pandas as pd: Imports the pandas library, which is used for data manipulation and analysis. It provides data structures like DataFrames for handling structured data.
- import numpy as np: Imports the numpy library, which is used for numerical operations. It provides functions for generating random numbers and other mathematical operations.

2. Setting Random Seed

```
np.random.seed(0)
```

- np.random.seed(0): Sets the seed for the random number generator in numpy. This ensures that the random numbers generated are the same each time the code is run, allowing for reproducibility of results.

3. Defining Number of Records and Entities

```
num_records = 10000 # Larger dataset for more
comprehensive analysis
num_routes = 50
num_vehicles = 100
```

- num_records = 10000: Specifies the number of records to generate in the synthetic dataset. A larger dataset allows for more comprehensive analysis.
- num_routes = 50: Defines the number of different routes.
- num_vehicles = 100: Defines the number of different vehicles.

4. Creating Lists for Routes and Vehicles

```
routes = [f'Route_{i}' for i in range(num_routes)]
vehicles = [f'Vehicle_{i}' for i in range(num_vehicles)]
```

- routes: Creates a list of route names using a list comprehension. Each route is named Route_0, Route_1, up to Route_49.
- vehicles: Creates a list of vehicle names using a list comprehension. Each vehicle is named Vehicle_0, Vehicle_1, up to Vehicle_99.

5. Generating Synthetic Data

```
data = {
    'Vehicle': np.random.choice(vehicles,
size=num_records), # Randomly assign vehicles
    'Route': np.random.choice(routes,
size=num_records), # Randomly assign routes
    'Distance_Traveled': np.random.uniform(5, 500,
size=num_records).round(2), # Random distance values
    'Duration_Hours': np.random.uniform(0.5, 10,
size=num_records).round(2), # Random duration values
    'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H') # Hourly frequency
}
```

- Vehicle: Assigns a random vehicle to each record from the vehicles list.

- Route: Assigns a random route to each record from the routes list.
- Distance_Traveled: Generates random distance values between 5 and 500 units. The values are rounded to 2 decimal places.
- Duration_Hours: Generates random duration values between 0.5 and 10 hours, rounded to 2 decimal places.
- Date: Creates a date range starting from January 1, 2024, with hourly frequency for the number of records specified.

6. Creating DataFrame

```
df = pd.DataFrame(data)
```

- df: Creates a DataFrame from the synthetic data dictionary. This DataFrame contains all the generated records with columns for Vehicle, Route, Distance_Traveled, Duration_Hours, and Date.

7. Grouping and Aggregating Data

```
grouped_df = df.groupby(['Vehicle', 'Route']).agg({
    'Distance_Traveled': ['sum', 'mean'], # Total and average distance traveled
    'Duration_Hours': ['sum', 'mean'] # Total and average duration
}).reset_index()
```

- df.groupby(['Vehicle', 'Route']): Groups the DataFrame by Vehicle and Route, so that each unique combination of

these columns is treated as a group.

- `.agg({...})`: Aggregates data within each group. For `Distance_Traveled`, it calculates the total (sum) and average (mean). Similarly, for `Duration_Hours`, it calculates the total (sum) and average (mean).
- `.reset_index()`: Resets the index of the DataFrame, converting the grouped indices into columns.

8. Flattening MultiIndex Columns

```
grouped_df.columns = ['_'.join(col).strip() for col in grouped_df.columns.values]
grouped_df.rename(columns={'Vehicle_': 'Vehicle',
'Route_': 'Route'}, inplace=True)
```

- `grouped_df.columns`: Flattens the MultiIndex columns created by the aggregation. The `agg` function results in a MultiIndex (e.g., `Distance_Traveled_sum`, `Distance_Traveled_mean`). This line concatenates the levels of the MultiIndex into single column names (e.g., `Distance_Traveled_sum`).
- `.rename(...)`: Renames columns to remove trailing underscores added during the flattening process, ensuring that the column names are clean and readable.

9. Saving Data to Excel

```
output_file = 'transportation_data_summary.xlsx'
grouped_df.to_excel(output_file,
sheet_name='Transportation Summary', index=False)
```

- `output_file`: Specifies the name of the output Excel file.

- `grouped_df.to_excel(...)`: Saves the aggregated DataFrame to an Excel file named `transportation_data_summary.xlsx`. The `sheet_name` parameter names the sheet within the Excel file as Transportation Summary. The `index=False` parameter ensures that the DataFrame's index is not saved to the Excel file.

10. Print Confirmation

```
print(f"Data saved to {output_file}")
```

- `print(...)`: Prints a confirmation message indicating that the data has been successfully saved to the specified file.

This code effectively generates a comprehensive transportation dataset, aggregates key metrics, and saves the summarized data to an Excel file for further analysis.

EXAMPLE 4.6

GUI Tkinter for Grouping and Multiple Aggregations with Synthetic Transportation Data

The purpose of this project is to create an interactive graphical user interface (GUI) for analyzing and visualizing synthetic transportation data using Python's Tkinter and Matplotlib libraries. The GUI is designed to offer users a comprehensive tool for inspecting and understanding large datasets related to transportation, such as vehicle routes, distances traveled, and duration of trips. By leveraging Tkinter's capabilities, the project aims to provide a user-friendly interface that displays both raw and

aggregated data in tabular form, allowing users to filter and sort the data based on various criteria.

Additionally, the project focuses on incorporating data visualization features to enhance the analysis process. Through the integration of Matplotlib, the application can generate and display a variety of plots and charts directly within the Tkinter window. These visualizations include histograms, scatter plots, and bar charts, which help users to quickly grasp patterns and trends in the data. By providing these visual insights, the project aims to make data analysis more intuitive and accessible, facilitating a deeper understanding of transportation metrics.

Overall, this project seeks to bridge the gap between raw data and actionable insights by combining data manipulation and visualization in a single, interactive application. The inclusion of filtering options, dynamic updates to the displayed data, and integrated plotting capabilities ensures that users can perform comprehensive data analysis efficiently. The goal is to create a versatile tool that supports decision-making processes and enhances the ability to extract valuable information from large and complex datasets.

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Set a seed for reproducibility
np.random.seed(0)
```

```
# Define the number of records
num_records = 10000
num_routes = 50
num_vehicles = 100
routes = [f'Route_{i}' for i in range(num_routes)]
vehicles = [f'Vehicle_{i}' for i in
range(num_vehicles)]

# Create synthetic transportation dataset
data = {
    'Vehicle': np.random.choice(vehicles,
size=num_records),
    'Route': np.random.choice(routes,
size=num_records),
    'Distance_Traveled': np.random.uniform(5, 500,
size=num_records).round(2),
    'Duration_Hours': np.random.uniform(0.5, 10,
size=num_records).round(2),
    'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H')
}
df = pd.DataFrame(data)

# Group by Vehicle and Route, then aggregate
# distance and duration
grouped_df = df.groupby(['Vehicle',
'Route']).agg({
    'Distance_Traveled': ['sum', 'mean'],
    'Duration_Hours': ['sum', 'mean']
}).reset_index()
grouped_df.columns = ['_'.join(col).strip() for
col in grouped_df.columns.values]
grouped_df.rename(columns={'Vehicle_': 'Vehicle',
'Route_': 'Route'}, inplace=True)
```

```
# Create the main application window
root = tk.Tk()
root.title("Transportation Data Analysis")
root.geometry("1200x800")

# Create a frame for raw data table
frame_raw_data = tk.Frame(root)
frame_raw_data.pack(fill=tk.BOTH, expand=True)

# Create a Treeview widget for displaying the raw
# DataFrame
tree_raw = ttk.Treeview(frame_raw_data,
columns=list(df.columns), show='headings')
tree_raw.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Scrollbars for the raw data Treeview
vsb_raw = ttk.Scrollbar(frame_raw_data,
orient="vertical", command=tree_raw.yview)
vsb_raw.pack(side='right', fill='y')
tree_raw.configure(yscrollcommand=vsb_raw.set)

hsb_raw = ttk.Scrollbar(frame_raw_data,
orient="horizontal", command=tree_raw.xview)
hsb_raw.pack(side='bottom', fill='x')
tree_raw.configure(xscrollcommand=hsb_raw.set)

# Define the columns in the raw data Treeview
for col in df.columns:
    tree_raw.heading(col, text=col)
    tree_raw.column(col, width=120)

# Function to update the raw data Treeview with
# alternating row colors
def update_raw_treeview(dataframe):
    for i in tree_raw.get_children():
```

```
        tree_raw.delete(i)
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree_raw.insert("", "end",
values=list(row), tags=(tag,))
        tree_raw.tag_configure('even',
background='#f5f5f5') # Light grey
        tree_raw.tag_configure('odd',
background='#ffffff') # White

update_raw_treeview(df)

# Create a frame for aggregated data table
frame_grouped_data = tk.Frame(root)
frame_grouped_data.pack(fill=tk.BOTH, expand=True)

# Create a Treeview widget for displaying the
grouped DataFrame
tree_grouped = ttk.Treeview(frame_grouped_data,
columns=list(grouped_df.columns), show='headings')
tree_grouped.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Scrollbars for the grouped data Treeview
vsb_grouped = tk.Scrollbar(frame_grouped_data,
orient="vertical", command=tree_grouped.yview)
vsb_grouped.pack(side='right', fill='y')
tree_grouped.configure(yscrollcommand=vsb_grouped.
set)

hsb_grouped = tk.Scrollbar(frame_grouped_data,
orient="horizontal", command=tree_grouped.xview)
hsb_grouped.pack(side='bottom', fill='x')
tree_grouped.configure(xscrollcommand=hsb_grouped.
set)
```

```

# Define the columns in the grouped data Treeview
for col in grouped_df.columns:
    tree_grouped.heading(col, text=col)
    tree_grouped.column(col, width=120)

# Function to update the grouped data Treeview
# with alternating row colors
def update_grouped_treeview(dataframe):
    for i in tree_grouped.get_children():
        tree_grouped.delete(i)
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree_grouped.insert("", "end",
values=list(row), tags=(tag,))
        tree_grouped.tag_configure('even',
background='#f5f5f5') # Light grey
        tree_grouped.tag_configure('odd',
background='#ffffff') # White

update_grouped_treeview(grouped_df)

# Create a frame for filters and buttons
frame_controls = tk.Frame(root)
frame_controls.pack(fill=tk.X)

# Filter options
tk.Label(frame_controls,
text="Vehicle:").pack(side=tk.LEFT, padx=5,
pady=5)
vehicle_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Vehicle'].unique()))
vehicle_combobox.set('All')
vehicle_combobox.pack(side=tk.LEFT, padx=5,
pady=5)

```

```
tk.Label(frame_controls,
text="Route:").pack(side=tk.LEFT, padx=5, pady=5)
route_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Route'].unique()))
route_combobox.set('All')
route_combobox.pack(side=tk.LEFT, padx=5, pady=5)

tk.Label(frame_controls, text="Start
Date:").pack(side=tk.LEFT, padx=5, pady=5)
start_date_entry = tk.Entry(frame_controls)
start_date_entry.pack(side=tk.LEFT, padx=5,
pady=5)

tk.Label(frame_controls, text="End
Date:").pack(side=tk.LEFT, padx=5, pady=5)
end_date_entry = tk.Entry(frame_controls)
end_date_entry.pack(side=tk.LEFT, padx=5, pady=5)

# Function to apply filters and update the data
views
def apply_filters():
    vehicle = vehicle_combobox.get()
    route = route_combobox.get()
    start_date = start_date_entry.get()
    end_date = end_date_entry.get()

    filtered_df = df.copy()
    if vehicle != 'All':
        filtered_df =
filtered_df[filtered_df['Vehicle'] == vehicle]
    if route != 'All':
        filtered_df =
filtered_df[filtered_df['Route'] == route]
    if start_date:
        filtered_df =
filtered_df[filtered_df['Date'] >= start_date]
```

```

    if end_date:
        filtered_df =
filtered_df[filtered_df['Date'] <= end_date]

    update_raw_treeview(filtered_df)

    # Group by Vehicle and Route, then aggregate
    distance and duration
    grouped_filtered_df =
filtered_df.groupby(['Vehicle', 'Route']).agg({
        'Distance_Traveled': ['sum', 'mean'],
        'Duration_Hours': ['sum', 'mean']
    }).reset_index()
    grouped_filtered_df.columns =
['_'.join(col).strip() for col in
grouped_filtered_df.columns.values]
    grouped_filtered_df.rename(columns=
{'Vehicle_': 'Vehicle', 'Route_': 'Route'},
inplace=True)

    update_grouped_treeview(grouped_filtered_df)

btn_apply_filters = tk.Button(frame_controls,
text="Apply Filters", command=apply_filters)
btn_apply_filters.pack(side=tk.LEFT, padx=5,
pady=5)

# Function to save filtered data to Excel
def save_to_excel():
    filtered_data = tree_raw.get_children()
    if not filtered_data:
        messagebox.showwarning("Warning", "No data
to save.")
    return
    file_path =
filedialog.asksaveasfilename(defaultextension=".xl"

```

```
sx", filetypes=[("Excel files", "*.xlsx")])
    if file_path:
        df_filtered =
pd.DataFrame([tree_raw.item(item)['values'] for
item in filtered_data], columns=df.columns)
        df_filtered.to_excel(file_path,
index=False)
        messagebox.showinfo("Info", f"Data saved
to {file_path}")

btn_save = tk.Button(frame_controls, text="Save to
Excel", command=save_to_excel)
btn_save.pack(side=tk.LEFT, padx=5, pady=5)

# Create a frame for data visualization
frame_plot = tk.Frame(root)
frame_plot.pack(fill=tk.BOTH, expand=True)

# Function to create and display a plot
def plot_data():
    plt.figure(figsize=(10, 6))
    df['Route'].value_counts().plot(kind='bar')
    plt.title('Number of Trips by Route')
    plt.xlabel('Route')
    plt.ylabel('Number of Trips')

    for widget in frame_plot.winfo_children():
        widget.destroy()

    canvas = FigureCanvasTkAgg(plt.gcf(),
master=frame_plot)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,
expand=True)
```

```
btn_plot = tk.Button(frame_controls, text="Plot Data", command=plot_data)
btn_plot.pack(side=tk.LEFT, padx=5, pady=5)

# Start the Tkinter event loop
root.mainloop()
```

Here's a detailed explanation of each part of the Tkinter GUI code:

1. Import Libraries

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from    matplotlib.backends.backend_tkagg    import
FigureCanvasTkAgg
```

- **tkinter:** Provides the core tools to create the graphical user interface (GUI). It's used for creating windows, dialogs, buttons, and other GUI elements.
- **ttk:** A module within tkinter that offers enhanced versions of widgets, like Treeview for tables and Combobox for dropdowns.
- **filedialog:** A submodule in tkinter for opening and saving files through file dialogs.
- **messagebox:** Provides standard pop-up dialogs for showing messages, warnings, and errors to the user.
- **pandas:** A data manipulation library that allows easy handling and analysis of data in tabular form.
- **numpy:** A library for numerical operations, including random number generation and array handling.

- matplotlib: A plotting library for creating static, animated, and interactive visualizations.
- FigureCanvasTkAgg: A backend for matplotlib that integrates with Tkinter to display plots within a Tkinter window.

2. Generate Synthetic Data

```
# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000
num_routes = 50
num_vehicles = 100
routes = [f'Route_{i}' for i in range(num_routes)]
vehicles = [f'Vehicle_{i}' for i in range(num_vehicles)]

# Create synthetic transportation dataset
data = {
    'Vehicle': np.random.choice(vehicles,
size=num_records),
    'Route': np.random.choice(routes,
size=num_records),
    'Distance_Traveled': np.random.uniform(5, 500,
size=num_records).round(2),
    'Duration_Hours': np.random.uniform(0.5, 10,
size=num_records).round(2),
    'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H')
}
df = pd.DataFrame(data)
```

- np.random.seed(0): Initializes the random number generator with a fixed seed value, ensuring that every time the script is run, the random numbers (and therefore the dataset) are the same. This is useful for reproducibility in experiments.
- num_records: Specifies the total number of rows in the synthetic dataset.
- num_routes, num_vehicles: Defines the number of different routes and vehicles to be included in the dataset.
- routes: A list of route identifiers (Route_0, Route_1, ..., Route_49), generated using a list comprehension.
- vehicles: A list of vehicle identifiers (Vehicle_0, Vehicle_1, ..., Vehicle_99), also generated using a list comprehension.
- np.random.choice: Randomly selects elements from the routes and vehicles lists to populate the 'Vehicle' and 'Route' columns.
- np.random.uniform: Generates random floating-point numbers within a specified range for 'Distance_Traveled' and 'Duration_Hours'. Values are rounded to two decimal places.
- pd.date_range: Creates a time series from January 1, 2024, with an hourly frequency for num_records periods.
- df: A DataFrame created from the synthetic data dictionary, with columns for 'Vehicle', 'Route', 'Distance_Traveled', 'Duration_Hours', and 'Date'.

3. Group and Aggregate Data

```
# Group by Vehicle and Route, then aggregate
distance and duration
```

```

grouped_df = df.groupby(['Vehicle',
'Route']).agg({
    'Distance_Traveled': ['sum', 'mean'],
    'Duration_Hours': ['sum', 'mean']
}).reset_index()
grouped_df.columns = ['_'.join(col).strip() for
col in grouped_df.columns.values]
grouped_df.rename(columns={'Vehicle_': 'Vehicle',
'Route_': 'Route'}, inplace=True)

```

- `groupby(['Vehicle', 'Route'])`: Groups the data by the 'Vehicle' and 'Route' columns, so that each unique combination of vehicle and route will be aggregated separately.
- `agg()`: Aggregates the grouped data. It computes:
 - 'Distance_Traveled': ['sum', 'mean']: The total ('sum') and average ('mean') distance traveled.
 - 'Duration_Hours': ['sum', 'mean']: The total ('sum') and average ('mean') duration of trips.
- `reset_index()`: Resets the index of the resulting DataFrame, turning the multi-level index (created by `groupby`) into regular columns.
- `grouped_df.columns`: Flattens the multi-level column names resulting from the aggregation into a single level, combining levels with an underscore.
- `rename()`: Renames the columns to remove the extra prefix added during aggregation, resulting in simpler column names like 'Vehicle', 'Route', 'Distance_Traveled_sum', etc.

4. Create Main Application Window

```
root = tk.Tk()
root.title("Transportation Data Analysis")
root.geometry("1200x800")
```

- `tk.Tk()`: Initializes the main window of the Tkinter application.
- `title("Transportation Data Analysis")`: Sets the title of the window, which appears in the title bar.
- `geometry("1200x800")`: Sets the initial size of the window to 1200 pixels wide and 800 pixels high.

5. Create Frame for Raw Data

```
frame_raw_data = tk.Frame(root)
frame_raw_data.pack(fill=tk.BOTH, expand=True)
```

- `tk.Frame(root)`: Creates a container widget within the main window to hold other widgets.
- `pack(fill=tk.BOTH, expand=True)`: Configures the frame to expand and fill both the vertical and horizontal space available within its parent.

6. Create Treeview for Raw Data

```
tree_raw      =      ttk.Treeview(frame_raw_data,
columns=list(df.columns), show='headings')
tree_raw.pack(side=tk.LEFT,           fill=tk.BOTH,
expand=True)
```

- `ttk.Treeview`: Creates a widget for displaying tabular data in a tree-like structure. Here, it's used to show the raw data.

- columns=list(df.columns): Sets the column headers based on the DataFrame's columns.
- show='headings': Ensures that only column headings are displayed, without row numbers.

7. Add Scrollbars for Raw Data

```
vsb_raw      =      ttk.Scrollbar(frame_raw_data,
orient="vertical", command=tree_raw.yview)
vsb_raw.pack(side='right', fill='y')
tree_raw.configure(yscrollcommand=vsb_raw.set)

hsb_raw      =      ttk.Scrollbar(frame_raw_data,
orient="horizontal", command=tree_raw.xview)
hsb_raw.pack(side='bottom', fill='x')
tree_raw.configure(xscrollcommand=hsb_raw.set)
```

- `ttk.Scrollbar`: Adds vertical and horizontal scrollbars to the frame for navigating through large datasets.
- `orient`: Specifies the orientation of the scrollbar ('vertical' or 'horizontal').
- `command`: Binds the scrollbar's movement to the Treeview's scrolling.
- `pack(side='right', fill='y')` and `pack(side='bottom', fill='x')`: Positions the scrollbars on the right and bottom of the frame, respectively.
- `configure(yscrollcommand=vsb_raw.set)` and `configure(xscrollcommand=hsb_raw.set)`: Links the Treeview's scrollbars to the scrollbars.

8. Define Columns for Raw Data Treeview

```
for col in df.columns:  
    tree_raw.heading(col, text=col)  
    tree_raw.column(col, width=120)
```

- heading(col, text=col): Sets the text for the column headings in the Treeview.
- column(col, width=120): Sets the width of each column to 120 pixels.

9. Update Raw Data Treeview with Alternating Row Colors

```
def update_raw_treeview(dataframe):  
    for i in tree_raw.get_children():  
        tree_raw.delete(i)  
    for index, row in dataframe.iterrows():  
        tag = 'even' if index % 2 == 0 else 'odd'  
        tree_raw.insert("", "end",  
values=list(row), tags=(tag,))  
        tree_raw.tag_configure('even',  
background='#f5f5f5')  
        tree_raw.tag_configure('odd',  
background='#ffffff')
```

- update_raw_treeview: Function to update the Treeview with new data and apply alternating row colors.
- get_children(): Retrieves all current rows in the Treeview.
- delete(i): Removes existing rows.
- insert("", "end", values=list(row), tags=(tag,)): Adds new rows with alternating row colors based on the index.
- tag_configure: Sets the background color for rows tagged as 'even' and 'odd' to improve readability.

10. Create Frame for Aggregated Data

```
frame_grouped_data = tk.Frame(root)
frame_grouped_data.pack(fill=tk.BOTH, expand=True)
```

- frame_grouped_data: A new frame for holding the Treeview that will display the aggregated data.

11. Create Treeview for Aggregated Data

```
tree_grouped = ttk.Treeview(frame_grouped_data,
columns=list(grouped_df.columns), show='headings')
tree_grouped.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)
```

- tree_grouped: Similar to tree_raw, but for displaying aggregated data.

12. Add Scrollbars for Aggregated Data

```
vsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="vertical", command=tree_grouped.yview)
vsb_grouped.pack(side='right', fill='y')
tree_grouped.configure(yscrollcommand=vsb_grouped.set)
```

```
hsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="horizontal", command=tree_grouped.xview)
hsb_grouped.pack(side='bottom', fill='x')
tree_grouped.configure(xscrollcommand=hsb_grouped.set)
```

- vsb_grouped and hsb_grouped: Vertical and horizontal scrollbars for navigating the aggregated data Treeview.

13. Define Columns for Aggregated Data Treeview

```
for col in grouped_df.columns:  
    tree_grouped.heading(col, text=col)  
    tree_grouped.column(col, width=120)
```

- heading and column: Configure column headers and widths for the aggregated data Treeview.

14. Update Aggregated Data Treeview with Alternating Row Colors

```
def update_grouped_treeview(dataframe):  
    for i in tree_grouped.get_children():  
        tree_grouped.delete(i)  
    for index, row in dataframe.iterrows():  
        tag = 'even' if index % 2 == 0 else 'odd'  
        tree_grouped.insert("", "end",  
values=list(row), tags=(tag,))  
        tree_grouped.tag_configure('even',  
background='#f5f5f5')  
        tree_grouped.tag_configure('odd',  
background='#ffffff')
```

- update_grouped_treeview: Updates the aggregated data Treeview with new data and applies alternating row colors.

15. Create Frame for Filters and Buttons

```
frame_controls = tk.Frame(root)  
frame_controls.pack(fill=tk.X)
```

- frame_controls: A frame for holding filter controls and buttons.

16. Add Filter Controls

```
tk.Label(frame_controls,
text="Vehicle:").pack(side=tk.LEFT,           padx=5,
pady=5)
vehicle_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Vehicle'].unique()))
vehicle_combobox.set('All')
vehicle_combobox.pack(side=tk.LEFT,           padx=5,
pady=5)

tk.Label(frame_controls,
text="Route:").pack(side=tk.LEFT, padx=5, pady=5)
route_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Route'].unique()))
route_combobox.set('All')
route_combobox.pack(side=tk.LEFT, padx=5, pady=5)
```

- tk.Label: Creates labels for the filter controls.
- ttk.Combobox: Creates dropdown menus for selecting filters. It includes options for filtering by 'Vehicle' and 'Route'.
- values=['All'] + list(df['Vehicle'].unique()): Sets the options in the combobox to include 'All' and the unique values from the 'Vehicle' column.
- set('All'): Sets the default value of the combobox to 'All'.
- pack(side=tk.LEFT, padx=5, pady=5): Arranges the comboboxes and labels in the frame, with padding around each widget.

17. Add Filtering Functionality

```
def filter_data():
    vehicle_filter = vehicle_combobox.get()
    route_filter = route_combobox.get()

    filtered_df = df.copy()

    if vehicle_filter != 'All':
        filtered_df = filtered_df[filtered_df['Vehicle'] == vehicle_filter]
    if route_filter != 'All':
        filtered_df = filtered_df[filtered_df['Route'] == route_filter]

    update_raw_treeview(filtered_df)
```

- filter_data: A function that filters the raw data based on the selected values in the comboboxes.
- vehicle_filter, route_filter: Gets the current selection from the comboboxes.
- filtered_df: A copy of the original DataFrame, which is filtered based on the selected vehicle and route.
- update_raw_treeview(filtered_df): Updates the Treeview with the filtered data.

18. Add Plotting Functionality

```
def plot_data():
    fig, axs = plt.subplots(2, 2, figsize=(10, 10))
    fig.tight_layout(pad=4.0)

    # Example plots
```

```

        axs[0, 0].hist(df['Distance_Traveled'],
bins=30, color='blue', edgecolor='black')
        axs[0, 0].set_title('Distance Traveled Distribution')

        axs[0, 1].hist(df['Duration_Hours'], bins=30,
color='green', edgecolor='black')
        axs[0, 1].set_title('Duration Hours Distribution')

        axs[1, 0].scatter(df['Distance_Traveled'],
df['Duration_Hours'])
        axs[1, 0].set_title('Distance vs Duration')

        axs[1, 1].bar(grouped_df['Vehicle'],
grouped_df['Distance_Traveled_sum'])
        axs[1, 1].set_title('Total Distance by Vehicle')

for ax in axs.flat:
    ax.label_outer()

canvas = FigureCanvasTkAgg(fig, master=root)
canvas.draw()
    canvas.get_tk_widget().pack(side=tk.RIGHT,
fill=tk.BOTH, expand=True)

```

- plot_data: A function that creates and displays a set of plots within the Tkinter application.
- fig, axs: Creates a figure with a 2x2 grid of subplots.
- hist: Plots histograms of 'Distance_Traveled' and 'Duration_Hours'.
- scatter: Creates a scatter plot of 'Distance_Traveled' vs. 'Duration_Hours'.

- bar: Creates a bar chart showing total distance traveled by each vehicle.
- FigureCanvasTkAgg: Integrates the Matplotlib figure with Tkinter, allowing the figure to be displayed in the Tkinter window.

This detailed breakdown covers each segment of the Tkinter GUI, explaining the purpose and functionality of the various components and code blocks.

Transportation Data Analysis

| Vehicle | Route | Distance_Traveled | Duration_Hours | Date |
|------------|----------|-------------------|----------------|---------------------|
| Vehicle_44 | Route_39 | 191.96 | 8.61 | 2024-01-01 00:00:00 |
| Vehicle_47 | Route_15 | 21.68 | 4.74 | 2024-01-01 01:00:00 |
| Vehicle_64 | Route_34 | 233.31 | 2.34 | 2024-01-01 02:00:00 |
| Vehicle_67 | Route_17 | 236.56 | 4.01 | 2024-01-01 03:00:00 |
| Vehicle_67 | Route_10 | 75.13 | 1.39 | 2024-01-01 04:00:00 |
| Vehicle_9 | Route_6 | 337.75 | 8.84 | 2024-01-01 05:00:00 |
| Vehicle_83 | Route_39 | 201.53 | 3.39 | 2024-01-01 06:00:00 |
| Vehicle_21 | Route_41 | 404.76 | 4.86 | 2024-01-01 07:00:00 |
| Vehicle_36 | Route_10 | 431.72 | 9.73 | 2024-01-01 08:00:00 |
| Vehicle_87 | Route_15 | 149.35 | 4.62 | 2024-01-01 09:00:00 |
| Vehicle_70 | Route_15 | 107.63 | 6.12 | 2024-01-01 10:00:00 |
| Vehicle_88 | Route_45 | 374.26 | 6.41 | 2024-01-01 11:00:00 |
| Vehicle_88 | Route_19 | 464.67 | 2.5 | 2024-01-01 12:00:00 |
| Vehicle_12 | Route_42 | 201.42 | 8.88 | 2024-01-01 13:00:00 |

| Vehicle | Route | Distance_Traveled_sum | Distance_Traveled_mean | Duration_Hours_sum | Duration_Hours_mean |
|-----------|----------|-----------------------|------------------------|--------------------|---------------------|
| Vehicle_0 | Route_1 | 415.81 | 415.81 | 5.21 | 5.21 |
| Vehicle_0 | Route_10 | 220.95 | 220.95 | 6.53 | 6.53 |
| Vehicle_0 | Route_11 | 274.99 | 91.66333333333334 | 17.48 | 5.8266666666666667 |
| Vehicle_0 | Route_12 | 475.89 | 158.63 | 20.75 | 6.9166666666666667 |
| Vehicle_0 | Route_13 | 248.93 | 248.93 | 6.28 | 6.28 |
| Vehicle_0 | Route_14 | 289.08 | 289.08 | 3.88 | 3.88 |
| Vehicle_0 | Route_15 | 478.2100000000004 | 159.40333333333334 | 13.53 | 4.51 |
| Vehicle_0 | Route_16 | 217.69 | 217.69 | 3.5 | 3.5 |
| Vehicle_0 | Route_17 | 939.09 | 234.7725 | 18.17 | 4.5425 |
| Vehicle_0 | Route_18 | 851.680000000001 | 283.8933333333334 | 15.66 | 5.22 |
| Vehicle_0 | Route_19 | 1429.32 | 357.33 | 17.56 | 4.39 |
| Vehicle_0 | Route_2 | 636.77 | 318.385 | 4.229999999999995 | 2.114999999999998 |
| Vehicle_0 | Route_20 | 268.2 | 134.1 | 15.25 | 7.625 |
| Vehicle_0 | Route_21 | 304.3 | 152.15 | 12.25 | 6.125 |

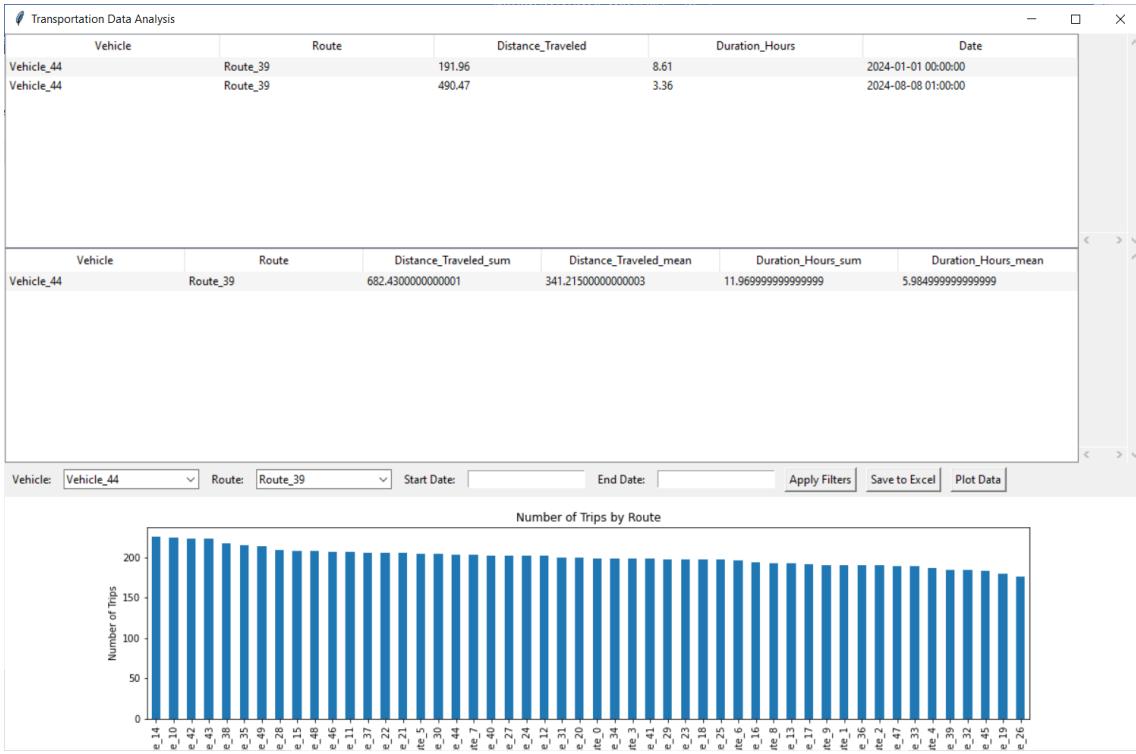
Vehicle: Route: Start Date: End Date: Apply Filters

Transportation Data Analysis

| Vehicle | Route | Distance_Traveled | Duration_Hours | Date |
|------------|------------|-----------------------|------------------------|---|
| Vehicle_44 | Route_39 | 191.96 | 8.61 | 2024-01-01 00:00:00 |
| Vehicle_44 | Route_17 | 289.77 | 5.3 | 2024-01-10 00:00:00 |
| Vehicle_44 | Route_7 | 392.15 | 1.43 | 2024-01-12 05:00:00 |
| Vehicle_44 | Route_35 | 76.17 | 5.24 | 2024-01-13 17:00:00 |
| Vehicle_44 | Route_32 | 144.0 | 4.78 | 2024-01-18 11:00:00 |
| Vehicle_44 | Route_5 | 359.25 | 8.03 | 2024-01-24 08:00:00 |
| Vehicle_44 | Route_19 | 171.6 | 5.16 | 2024-02-03 00:00:00 |
| Vehicle_44 | Route_14 | 496.81 | 9.26 | 2024-02-17 13:00:00 |
| Vehicle_44 | Route_10 | 72.25 | 7.07 | 2024-02-19 15:00:00 |
| Vehicle_44 | Route_23 | 244.7 | 3.23 | 2024-02-19 17:00:00 |
| Vehicle_44 | Route_3 | 73.81 | 9.22 | 2024-02-21 04:00:00 |
| Vehicle_44 | Route_41 | 266.65 | 2.35 | 2024-02-25 12:00:00 |
| Vehicle_44 | Route_37 | 86.17 | 5.99 | 2024-02-28 19:00:00 |
| Vehicle_44 | Route_22 | 458.98 | 0.81 | 2024-03-01 08:00:00 |
| Vehicle | Route | Distance_Traveled_sum | Distance_Traveled_mean | Duration_Hours_sum |
| Vehicle_44 | Route_0 | 409.7 | 204.85 | 15.170000000000002 |
| Vehicle_44 | Route_10 | 72.25 | 72.25 | 7.07 |
| Vehicle_44 | Route_11 | 428.12 | 428.12 | 4.41 |
| Vehicle_44 | Route_12 | 664.54 | 332.27 | 10.469999999999999 |
| Vehicle_44 | Route_13 | 315.04 | 315.04 | 2.66 |
| Vehicle_44 | Route_14 | 1491.87 | 298.3739999999997 | 23.54 |
| Vehicle_44 | Route_15 | 626.8 | 313.4 | 17.1 |
| Vehicle_44 | Route_16 | 727.1600000000001 | 363.5800000000004 | 6.0 |
| Vehicle_44 | Route_17 | 1291.8899999999999 | 322.9724999999997 | 19.35 |
| Vehicle_44 | Route_18 | 270.57 | 270.57 | 9.42 |
| Vehicle_44 | Route_19 | 963.1500000000001 | 321.05 | 18.15 |
| Vehicle_44 | Route_2 | 165.06 | 82.53 | 11.67 |
| Vehicle_44 | Route_20 | 528.5999999999999 | 176.1999999999996 | 14.02 |
| Vehicle_44 | Route_22 | 690.54 | 345.27 | 10.76 |
| Vehicle: | Vehicle_44 | Route: | All | Start Date: End Date: Apply Filters Save to Excel Plot Data |

Transportation Data Analysis

| Vehicle | Route | Distance_Traveled | Duration_Hours | Date |
|------------|------------|-----------------------|------------------------|---|
| Vehicle_44 | Route_39 | 191.96 | 8.61 | 2024-01-01 00:00:00 |
| Vehicle_44 | Route_39 | 490.47 | 3.36 | 2024-08-08 01:00:00 |
| Vehicle | Route | Distance_Traveled_sum | Distance_Traveled_mean | Duration_Hours_sum |
| Vehicle_44 | Route_39 | 682.4300000000001 | 341.2150000000003 | 11.969999999999999 |
| Vehicle: | Vehicle_44 | Route: | Route_39 | Start Date: End Date: Apply Filters Save to Excel Plot Data |



EXAMPLE 4.7

Advanced Grouping Multiple Columns and Multiple Aggregations with Synthetic Marketing Data

The project creates and processes a synthetic marketing dataset to analyze marketing performance across various campaigns and channels. It begins by generating a large dataset with 10,000 records, where each record contains information on Campaign, Channel, Sales, Clicks, and Date. The Campaign and Channel columns represent different marketing initiatives and distribution methods, while Sales and Clicks quantify their performance. The Date column adds a time dimension to the dataset. This data is then organized into a DataFrame using pandas.

The core of the code involves grouping the dataset by Campaign and Channel, followed by performing multiple aggregations on the Sales and Clicks columns. These aggregations include computing

the total (sum), average (mean), maximum (max), and minimum (min) values, providing a comprehensive view of each campaign's and channel's performance. The results are then saved to an Excel file, making the summarized data available for reporting or further analysis. This approach helps in understanding marketing effectiveness and making data-driven decisions based on aggregated performance metrics.

```
import pandas as pd
import numpy as np

# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000

# Define a larger set of campaigns and channels
num_campaigns = 50
num_channels = 20
campaigns = [f'Campaign_{i}' for i in
range(num_campaigns)]
channels = [f'Channel_{i}' for i in
range(num_channels)]

# Create synthetic marketing dataset
data = {
    'Campaign': np.random.choice(campaigns,
size=num_records),
    'Channel': np.random.choice(channels,
size=num_records),
    'Sales': np.random.uniform(100, 5000,
size=num_records).round(2),
    'Clicks': np.random.randint(10, 500,
size=num_records),
```

```

        'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H')
}
df = pd.DataFrame(data)

# Group by Campaign and Channel, then aggregate
Sales and Clicks
grouped_df = df.groupby(['Campaign',
'Channel']).agg({
    'Sales': ['sum', 'mean', 'max', 'min'],
    'Clicks': ['sum', 'mean', 'max', 'min']
}).reset_index()

# Flatten the MultiIndex columns
grouped_df.columns = ['_'.join(col).strip() for
col in grouped_df.columns.values]
grouped_df.rename(columns={'Campaign_':
'Campaign', 'Channel_': 'Channel'}, inplace=True)

# Save the aggregated DataFrame to an Excel file
output_file = 'marketing_data_summary.xlsx'
grouped_df.to_excel(output_file,
sheet_name='Marketing Summary', index=False)

print(f"Data saved to {output_file}")

```

Here is a detailed explanation of each part of the code:

1. Import Libraries

```

import pandas as pd
import numpy as np

```

- **pandas (pd):** A powerful data manipulation library in Python. It provides data structures like DataFrames to

- handle and analyze data efficiently.
- numpy (np): A library for numerical operations. It is commonly used for generating random numbers and performing mathematical operations.

2. Set a Seed for Reproducibility

```
np.random.seed(0)
```

- np.random.seed(0): This function sets the seed for NumPy's random number generator. By setting the seed to a fixed value (0), the code ensures that the random numbers generated are reproducible. This means that every time the code is run, the same random numbers will be produced, which is useful for debugging and consistent results.

3. Define the Number of Records

```
num_records = 10000
```

- num_records: Specifies the number of records to generate for the synthetic dataset. Here, 10,000 records are created, which is a large enough dataset to allow for meaningful analysis.

4. Define a Larger Set of Campaigns and Channels

```
num_campaigns = 50
num_channels = 20
campaigns = [f'Campaign_{i}' for i in range(num_campaigns)]
```

```
channels = [f'Channel_{i}' for i in range(num_channels)]
```

- num_campaigns and num_channels: Define the number of unique campaigns and channels in the dataset.
- campaigns and channels: Generate lists of campaign names and channel names using list comprehensions. For example, campaigns will contain names like Campaign_0, Campaign_1, ..., Campaign_49.

5. Create Synthetic Marketing Dataset

```
data = {
    'Campaign': np.random.choice(campaigns,
size=num_records),
    'Channel': np.random.choice(channels,
size=num_records),
    'Sales': np.random.uniform(100, 5000,
size=num_records).round(2),
    'Clicks': np.random.randint(10, 500,
size=num_records),
    'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H')
}
df = pd.DataFrame(data)
```

- Campaign: Randomly selects campaign names from the campaigns list for each record.
- Channel: Randomly selects channel names from the channels list for each record.
- Sales: Generates random sales values between 100 and 5000, rounded to two decimal places.

- Clicks: Generates random integer values between 10 and 500 for the number of clicks.
- Date: Creates a date range starting from January 1, 2024, with hourly frequency for the specified number of records.
- df: Converts the data dictionary into a pandas DataFrame.

6. Group by Campaign and Channel, then Aggregate Sales and Clicks

```
grouped_df = df.groupby(['Campaign', 'Channel']).agg({
    'Sales': ['sum', 'mean', 'max', 'min'],
    'Clicks': ['sum', 'mean', 'max', 'min']
}).reset_index()
```

- groupby(['Campaign', 'Channel']): Groups the DataFrame by the Campaign and Channel columns.
- agg({ 'Sales': ['sum', 'mean', 'max', 'min'], 'Clicks': ['sum', 'mean', 'max', 'min'] }): Aggregates the data for each group by calculating the sum, mean, maximum, and minimum values for both Sales and Clicks.
- reset_index(): Resets the index of the DataFrame, making Campaign and Channel regular columns instead of hierarchical index levels.

7. Flatten the MultiIndex Columns

```
grouped_df.columns = ['_'.join(col).strip() for
col in grouped_df.columns.values]
grouped_df.rename(columns={'Campaign_':
'Campaign', 'Channel_': 'Channel'}, inplace=True)
```

- grouped_df.columns: Flattens the MultiIndex columns resulting from the aggregation by joining the tuple values with underscores.
- rename(columns={'Campaign_': 'Campaign', 'Channel_': 'Channel'}, inplace=True): Renames the columns to remove any trailing underscores and to have a more readable format.

8. Save the Aggregated DataFrame to an Excel File

```
output_file = 'marketing_data_summary.xlsx'  
grouped_df.to_excel(output_file,  
sheet_name='Marketing Summary', index=False)
```

- output_file: Specifies the name of the Excel file where the aggregated data will be saved.
- to_excel(output_file, sheet_name='Marketing Summary', index=False): Saves the grouped_df DataFrame to an Excel file with the specified sheet name (Marketing Summary) and without including the DataFrame index in the output file.

9. Print Confirmation Message

```
print(f"Data saved to {output_file}")
```

- print(f"Data saved to {output_file}"): Outputs a confirmation message indicating that the data has been successfully saved to the specified Excel file.

EXAMPLE 4.8

GUI Tkinter for Advanced Grouping Multiple Columns and Multiple Aggregations with Synthetic Marketing Data

The purpose of this project is to create a comprehensive Tkinter-based graphical user interface (GUI) for analyzing and visualizing synthetic marketing data. The application generates a large dataset that mimics real-world marketing metrics, such as sales and clicks for various campaigns and channels. By implementing a Tkinter GUI, users can interact with this data in a structured and visually appealing manner, which includes displaying raw and aggregated data in table format, applying various filters, and visualizing data through interactive plots.

The GUI includes several key features to facilitate data analysis. It presents raw marketing data and aggregated summaries in Treeview widgets, allowing users to view and compare information easily. Filters enable users to customize the displayed data based on campaign, channel, and date range criteria. This dynamic filtering capability helps users drill down into specific subsets of data and analyze trends or anomalies based on their preferences. Additionally, the GUI provides functionality to save the filtered results to an Excel file, which is useful for reporting and further analysis outside of the application.

Data visualization is another crucial aspect of the project. The GUI includes a plotting feature that generates bar charts representing the number of records per campaign. By utilizing color maps for distinct bar colors, the application enhances visual clarity and makes it easier to identify trends and patterns. This integration of data visualization helps users quickly interpret large datasets and gain insights into marketing performance, making the GUI a

valuable tool for both exploratory analysis and decision-making. Overall, the project aims to provide a user-friendly, interactive platform for marketing data exploration and reporting.

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.cm as cm

# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000

# Define a larger set of campaigns and channels
num_campaigns = 50
num_channels = 20
campaigns = [f'Campaign_{i}' for i in
range(num_campaigns)]
channels = [f'Channel_{i}' for i in
range(num_channels)]

# Create synthetic marketing dataset
data = {
    'Campaign': np.random.choice(campaigns,
size=num_records),
    'Channel': np.random.choice(channels,
size=num_records),
    'Sales': np.random.uniform(100, 5000,
size=num_records).round(2),
```

```
'Clicks': np.random.randint(10, 500,  
size=num_records),  
    'Date': pd.date_range(start='2024-01-01',  
periods=num_records, freq='H')  
}  
df = pd.DataFrame(data)  
  
# Group by Campaign and Channel, then aggregate  
Sales and Clicks  
grouped_df = df.groupby(['Campaign',  
'Channel']).agg({  
    'Sales': ['sum', 'mean', 'max', 'min'],  
    'Clicks': ['sum', 'mean', 'max', 'min']  
}).reset_index()  
grouped_df.columns = ['_'.join(col).strip() for  
col in grouped_df.columns.values]  
grouped_df.rename(columns={'Campaign_':  
'Campaign', 'Channel_': 'Channel'}, inplace=True)  
  
# Save the aggregated DataFrame to an Excel file  
output_file = 'marketing_data_summary.xlsx'  
grouped_df.to_excel(output_file,  
sheet_name='Marketing Summary', index=False)  
  
print(f"Data saved to {output_file}")  
  
# Create the main application window  
root = tk.Tk()  
root.title("Marketing Data Analysis")  
root.geometry("1200x800")  
  
# Create a frame for raw data table  
frame_raw_data = tk.Frame(root)  
frame_raw_data.pack(fill=tk.BOTH, expand=True)
```

```
# Create a Treeview widget for displaying the raw
DataFrame
tree_raw = ttk.Treeview(frame_raw_data,
columns=list(df.columns), show='headings' )
```

```
tree_raw.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Scrollbars for the raw data Treeview
vsb_raw = ttk.Scrollbar(frame_raw_data,
orient="vertical", command=tree_raw.yview)
vsb_raw.pack(side='right', fill='y')
tree_raw.configure(yscrollcommand=vsb_raw.set)

hsb_raw = ttk.Scrollbar(frame_raw_data,
orient="horizontal", command=tree_raw.xview)
hsb_raw.pack(side='bottom', fill='x')
tree_raw.configure(xscrollcommand=hsb_raw.set)

# Define the columns in the raw data Treeview
for col in df.columns:
    tree_raw.heading(col, text=col)
    tree_raw.column(col, width=120)

# Function to update the raw data Treeview with
alternating row colors
def update_raw_treeview(dataframe):
    for i in tree_raw.get_children():
        tree_raw.delete(i)
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree_raw.insert("", "end",
values=list(row), tags=(tag,))
        tree_raw.tag_configure('even',
background="#f5f5f5") # Light grey
        tree_raw.tag_configure('odd',
background="#ffffff") # White

update_raw_treeview(df)

# Create a frame for aggregated data table
```

```
frame_grouped_data = tk.Frame(root)
frame_grouped_data.pack(fill=tk.BOTH, expand=True)

# Create a Treeview widget for displaying the
grouped DataFrame
tree_grouped = ttk.Treeview(frame_grouped_data,
columns=list(grouped_df.columns), show='headings')
tree_grouped.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Scrollbars for the grouped data Treeview
vsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="vertical", command=tree_grouped.yview)
vsb_grouped.pack(side='right', fill='y')
tree_grouped.configure(yscrollcommand=vsb_grouped.
set)

hsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="horizontal", command=tree_grouped.xview)
hsb_grouped.pack(side='bottom', fill='x')
tree_grouped.configure(xscrollcommand=hsb_grouped.
set)

# Define the columns in the grouped data Treeview
for col in grouped_df.columns:
    tree_grouped.heading(col, text=col)
    tree_grouped.column(col, width=120)

# Function to update the grouped data Treeview
# with alternating row colors
def update_grouped_treeview(dataframe):
    for i in tree_grouped.get_children():
        tree_grouped.delete(i)
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
```

```
        tree_grouped.insert("", "end",
values=list(row), tags=(tag,))
    tree_grouped.tag_configure('even',
background='#f5f5f5') # Light grey
    tree_grouped.tag_configure('odd',
background='#ffffff') # White

update_grouped_treeview(grouped_df)

# Create a frame for filters and buttons
frame_controls = tk.Frame(root)
frame_controls.pack(fill=tk.X)

# Filter options
tk.Label(frame_controls,
text="Campaign:").pack(side=tk.LEFT, padx=5,
pady=5)
campaign_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Campaign'].unique()))
campaign_combobox.set('All')
campaign_combobox.pack(side=tk.LEFT, padx=5,
pady=5)

tk.Label(frame_controls,
text="Channel:").pack(side=tk.LEFT, padx=5,
pady=5)
channel_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Channel'].unique()))
channel_combobox.set('All')
channel_combobox.pack(side=tk.LEFT, padx=5,
pady=5)

tk.Label(frame_controls, text="Start
Date:").pack(side=tk.LEFT, padx=5, pady=5)
start_date_entry = tk.Entry(frame_controls)
```

```
start_date_entry.pack(side=tk.LEFT, padx=5,
pady=5)

tk.Label(frame_controls, text="End
Date:").pack(side=tk.LEFT, padx=5, pady=5)
end_date_entry = tk.Entry(frame_controls)
end_date_entry.pack(side=tk.LEFT, padx=5, pady=5)

# Function to apply filters and update the data
views
def apply_filters():
    campaign = campaign_combobox.get()
    channel = channel_combobox.get()
    start_date = start_date_entry.get()
    end_date = end_date_entry.get()

    filtered_df = df.copy()
    if campaign != 'All':
        filtered_df =
filtered_df[filtered_df['Campaign'] == campaign]
    if channel != 'All':
        filtered_df =
filtered_df[filtered_df['Channel'] == channel]
    if start_date:
        filtered_df =
filtered_df[filtered_df['Date'] >= start_date]
    if end_date:
        filtered_df =
filtered_df[filtered_df['Date'] <= end_date]

    update_raw_treeview(filtered_df)

    # Group by Campaign and Channel, then
aggregate Sales and Clicks
    grouped_filtered_df =
filtered_df.groupby(['Campaign', 'Channel']).agg({
```

```

        'Sales': ['sum', 'mean', 'max', 'min'],
        'Clicks': ['sum', 'mean', 'max', 'min']
    }).reset_index()
    grouped_filtered_df.columns =
['_'.join(col).strip() for col in
grouped_filtered_df.columns.values]
    grouped_filtered_df.rename(columns=
{'Campaign_': 'Campaign', 'Channel_': 'Channel'}, inplace=True)

    update_grouped_treeview(grouped_filtered_df)

btn_apply_filters = tk.Button(frame_controls,
text="Apply Filters", command=apply_filters)
btn_apply_filters.pack(side=tk.LEFT, padx=5,
pady=5)

# Function to save filtered data to Excel
def save_to_excel():
    filtered_data = tree_raw.get_children()
    if not filtered_data:
        messagebox.showwarning("Warning", "No data
to save.")
        return
    file_path =
filedialog.asksaveasfilename(defaultextension=".xl
sx", filetypes=[("Excel files", "*.xlsx")])
    if file_path:
        df_filtered =
pd.DataFrame([tree_raw.item(item)['values'] for
item in filtered_data], columns=df.columns)
        df_filtered.to_excel(file_path,
index=False)
        messagebox.showinfo("Info", f"Data saved
to {file_path}")

```

```
btn_save = tk.Button(frame_controls, text="Save to Excel", command=save_to_excel)
btn_save.pack(side=tk.LEFT, padx=5, pady=5)

# Create a frame for data visualization
frame_plot = tk.Frame(root, bg='#f8f9f9') # Light grey background
frame_plot.pack(fill=tk.BOTH, expand=True)

# Function to plot data
def plot_data():
    # Clear the previous plot
    for widget in frame_plot.winfo_children():
        widget.destroy()

    campaign_counts =
df['Campaign'].value_counts()

    fig, ax = plt.subplots(figsize=(10, 7))

    # Generate a color for each bar
    num_colors = len(campaign_counts)
    colors = cm.viridis(np.linspace(0, 1,
num_colors)) # Using a color map

    bars = ax.bar(campaign_counts.index,
campaign_counts.values, color=colors)

    ax.set_title('Number of Records per Campaign')
    ax.set_xlabel('Campaign')
    ax.set_ylabel('Number of Records')
    ax.tick_params(axis='x', rotation=90)

    canvas = FigureCanvasTkAgg(fig,
master=frame_plot)
    canvas.draw()
```

```
    canvas.get_tk_widget().pack(fill=tk.BOTH,  
expand=True)  
  
btn_plot = tk.Button(frame_controls, text="Plot  
Data", command=plot_data)  
btn_plot.pack(side=tk.LEFT, padx=5, pady=5)  
  
# Start the Tkinter event loop  
root.mainloop()
```

Here's a detailed explanation of each part of the code:

Import Statements

```
import tkinter as tk  
from tkinter import ttk, filedialog, messagebox  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.backends.backend_tkagg import  
FigureCanvasTkAgg  
import matplotlib.cm as cm
```

- `import tkinter as tk`: Imports the `tkinter` module, which is used to create GUI applications in Python. The alias `tk` simplifies usage throughout the code.
- `from tkinter import ttk, filedialog, messagebox`: Imports specific components from `tkinter`:
 - `ttk`: Provides access to themed widgets which have a more modern appearance than standard `tkinter` widgets.
 - `filedialog`: Allows users to open or save files through a graphical dialog.

- messagebox: Used to display pop-up messages to the user, such as warnings or informational messages.
- import pandas as pd: Imports the pandas library, which is essential for data manipulation and analysis.
- import numpy as np: Imports the numpy library, which is used for numerical operations and generating random data.
- import matplotlib.pyplot as plt: Imports pyplot from matplotlib, a library used for creating static, animated, and interactive visualizations.
- from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg: Allows embedding Matplotlib plots into Tkinter GUIs by creating a canvas that integrates Matplotlib figures with Tkinter.
- import matplotlib.cm as cm: Provides color maps (colormaps) that can be used to generate color gradients for visualizations.

Data Generation

```
# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000

# Define a larger set of campaigns and channels
num_campaigns = 50
num_channels = 20
campaigns = [f'Campaign_{i}' for i in range(num_campaigns)]
channels = [f'Channel_{i}' for i in range(num_channels)]
```

```

# Create synthetic marketing dataset
data = {
    'Campaign': np.random.choice(campaigns,
size=num_records),
    'Channel': np.random.choice(channels,
size=num_records),
    'Sales': np.random.uniform(100, 5000,
size=num_records).round(2),
    'Clicks': np.random.randint(10, 500,
size=num_records),
    'Date': pd.date_range(start='2024-01-01',
periods=num_records, freq='H')
}
df = pd.DataFrame(data)

```

- `np.random.seed(0)`: Sets the seed for the random number generator to ensure that the random data generated is the same every time the script is run. This helps in reproducibility.
- `num_records`: Specifies the number of records to generate for the dataset.
- `num_campaigns` and `num_channels`: Define the number of different campaigns and channels to simulate.
- `campaigns` and `channels`: Generate lists of campaign and channel names using list comprehensions.
- `data`: Creates a dictionary with column names as keys and randomly generated data as values:
 - `np.random.choice(campaigns, size=num_records)`: Randomly selects campaign names from the campaigns list.
 - `np.random.choice(channels, size=num_records)`: Randomly selects channel names from the channels

list.

- `np.random.uniform(100, 5000, size=num_records).round(2)`: Generates random sales values uniformly distributed between 100 and 5000 and rounds them to two decimal places.
- `np.random.randint(10, 500, size=num_records)`: Generates random integers for clicks, ranging from 10 to 500.
- `pd.date_range(start='2024-01-01', periods=num_records, freq='H')`: Creates a date range starting from January 1, 2024, with hourly frequency, for the number of records specified.
- `df = pd.DataFrame(data)`: Converts the dictionary into a Pandas DataFrame for easier data manipulation and analysis.

Data Aggregation and Export

```
# Group by Campaign and Channel, then aggregate Sales and Clicks
grouped_df = df.groupby(['Campaign', 'Channel']).agg({
    'Sales': ['sum', 'mean', 'max', 'min'],
    'Clicks': ['sum', 'mean', 'max', 'min']
}).reset_index()
grouped_df.columns = ['_'.join(col).strip() for col in grouped_df.columns.values]
grouped_df.rename(columns={'Campaign_': 'Campaign', 'Channel_': 'Channel'}, inplace=True)

# Save the aggregated DataFrame to an Excel file
output_file = 'marketing_data_summary.xlsx'
grouped_df.to_excel(output_file,
sheet_name='Marketing Summary', index=False)
```

```
print(f"Data saved to {output_file}")
```

- df.groupby(['Campaign', 'Channel']): Groups the DataFrame by the Campaign and Channel columns.
- agg(): Aggregates the grouped data using several statistical functions:
 - 'sum': Calculates the total sum for Sales and Clicks.
 - 'mean': Calculates the average value for Sales and Clicks.
 - 'max': Finds the maximum value for Sales and Clicks.
 - 'min': Finds the minimum value for Sales and Clicks.
- reset_index(): Resets the index of the resulting DataFrame so that Campaign and Channel become regular columns again.
- grouped_df.columns: Flattens the hierarchical column index (MultiIndex) created by agg().
- rename(columns={'Campaign_': 'Campaign', 'Channel_': 'Channel'}): Renames columns to remove extraneous underscores added during aggregation.
- to_excel(): Saves the DataFrame to an Excel file with the name marketing_data_summary.xlsx. The sheet is named "Marketing Summary", and the index is not included in the saved file.

GUI Setup

```
# Create the main application window
root = tk.Tk()
root.title("Marketing Data Analysis")
root.geometry("1200x800")
```

- `root = tk.Tk()`: Initializes the main Tkinter window.
- `root.title("Marketing Data Analysis")`: Sets the title of the main window.
- `root.geometry("1200x800")`: Sets the size of the main window to 1200x800 pixels.

Raw Data Table

```
# Create a frame for raw data table
frame_raw_data = tk.Frame(root)
frame_raw_data.pack(fill=tk.BOTH, expand=True)

# Create a Treeview widget for displaying the raw DataFrame
tree_raw      =      ttk.Treeview(frame_raw_data,
columns=list(df.columns), show='headings')
tree_raw.pack(side=tk.LEFT,           fill=tk.BOTH,
expand=True)

# Scrollbars for the raw data Treeview
vsb_raw       =      ttk.Scrollbar(frame_raw_data,
orient="vertical", command=tree_raw.yview)
vsb_raw.pack(side='right', fill='y')
tree_raw.configure(yscrollcommand=vsb_raw.set)

hsb_raw       =      ttk.Scrollbar(frame_raw_data,
orient="horizontal", command=tree_raw.xview)
hsb_raw.pack(side='bottom', fill='x')
tree_raw.configure(xscrollcommand=hsb_raw.set)

# Define the columns in the raw data Treeview
for col in df.columns:
    tree_raw.heading(col, text=col)
    tree_raw.column(col, width=120)
```

- frame_raw_data: A container for the raw data table.
- tree_raw: A Treeview widget to display the raw DataFrame. columns specifies the columns to display, and show='headings' means only the column headers are visible.
- vsb_raw and hsb_raw: Vertical and horizontal scrollbars to navigate through the data in the Treeview.
- tree_raw.configure(): Configures the Treeview to use the scrollbars.
- tree_raw.heading() and tree_raw.column(): Set column headings and widths.

Function to Update Raw Data Table

```
def update_raw_treeview(dataframe):
    for i in tree_raw.get_children():
        tree_raw.delete(i)
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
                    tree_raw.insert("", "end",
values=list(row), tags=(tag,))
                    tree_raw.tag_configure('even',
background='#f5f5f5') # Light grey
                    tree_raw.tag_configure('odd',
background='#ffffff') # White

update_raw_treeview(df)
```

- update_raw_treeview(dataframe): Updates the Treeview with the data from the DataFrame:
 - tree_raw.delete(i): Deletes existing rows in the Treeview.

- `tree_raw.insert("", "end", values=list(row), tags=(tag,))`: Inserts rows into the Treeview, alternating row colors using tags.
- `tag_configure()`: Sets background colors for alternating rows.

Aggregated Data Table

```
# Create a frame for aggregated data table
frame_grouped_data = tk.Frame(root)
frame_grouped_data.pack(fill=tk.BOTH, expand=True)

# Create a Treeview widget for displaying the grouped DataFrame
tree_grouped = ttk.Treeview(frame_grouped_data,
columns=list(grouped_df.columns), show='headings')
tree_grouped.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

# Scrollbars for the grouped data Treeview
vsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="vertical", command=tree_grouped.yview)
vsb_grouped.pack(side='right', fill='y')
tree_grouped.configure(yscrollcommand=vsb_grouped.set)

hsb_grouped = ttk.Scrollbar(frame_grouped_data,
orient="horizontal", command=tree_grouped.xview)
hsb_grouped.pack(side='bottom', fill='x')
tree_grouped.configure(xscrollcommand=hsb_grouped.set)

# Define the columns in the grouped data Treeview
for col in grouped_df.columns:
```

```
tree_grouped.heading(col, text=col)
tree_grouped.column(col, width=120)
```

- frame_grouped_data: Container for the aggregated data table.
- tree_grouped: A Treeview widget to display the aggregated DataFrame. Similar setup to the raw data table but for the aggregated data.
- vsb_grouped and hsb_grouped: Vertical and horizontal scrollbars for navigating the aggregated data.
- tree_grouped.configure(): Configures the Treeview to use the scrollbars.
- tree_grouped.heading() and tree_grouped.column(): Set column headings and widths for the aggregated data Treeview.

Filters and Buttons

```
# Create a frame for filters and buttons
frame_controls = tk.Frame(root)
frame_controls.pack(fill=tk.X)

# Filter options
tk.Label(frame_controls,
text="Campaign:").pack(side=tk.LEFT,           padx=5,
pady=5)
campaign_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Campaign'].unique()))
campaign_combobox.set('All')
campaign_combobox.pack(side=tk.LEFT,           padx=5,
pady=5)
```

```

tk.Label(frame_controls,
text="Channel:").pack(side=tk.LEFT,           padx=5,
pady=5)
channel_combobox = ttk.Combobox(frame_controls,
values=['All'] + list(df['Channel'].unique()))
channel_combobox.set('All')
channel_combobox.pack(side=tk.LEFT,           padx=5,
pady=5)

tk.Label(frame_controls,                      text="Start
Date:").pack(side=tk.LEFT, padx=5, pady=5)
start_date_entry = tk.Entry(frame_controls)
start_date_entry.pack(side=tk.LEFT,           padx=5,
pady=5)

tk.Label(frame_controls,                      text="End
Date:").pack(side=tk.LEFT, padx=5, pady=5)
end_date_entry = tk.Entry(frame_controls)
end_date_entry.pack(side=tk.LEFT, padx=5, pady=5)

```

- frame_controls: A container for the filter options and buttons, packed horizontally (fill=tk.X) to align with the top of the window.
- tk.Label: Creates labels for each filter option.
 - text="Campaign": Label text for the campaign filter.
 - pack(side=tk.LEFT, padx=5, pady=5): Places the label to the left and adds padding.
- campaign_combobox and channel_combobox: ComboBoxes for selecting campaign and channel filters.
 - values=['All'] + list(df['Campaign'].unique()): Sets the list of values in the ComboBox to include all unique campaign/channel names plus an "All" option.
 - set('All'): Sets the default value to "All".

- `start_date_entry` and `end_date_entry`: Entry widgets for entering start and end dates for filtering.

Functions for Filtering Data

```
# Function to apply filters and update the data
views
def apply_filters():
    campaign = campaign_combobox.get()
    channel = channel_combobox.get()
    start_date = start_date_entry.get()
    end_date = end_date_entry.get()

    filtered_df = df.copy()
    if campaign != 'All':
        filtered_df = filtered_df[filtered_df['Campaign'] == campaign]
    if channel != 'All':
        filtered_df = filtered_df[filtered_df['Channel'] == channel]
    if start_date:
        filtered_df = filtered_df[filtered_df['Date'] >= start_date]
    if end_date:
        filtered_df = filtered_df[filtered_df['Date'] <= end_date]

    update_raw_treeview(filtered_df)

    # Group by Campaign and Channel, then
    # aggregate Sales and Clicks
    grouped_filtered_df = filtered_df.groupby(['Campaign', 'Channel']).agg({
        'Sales': ['sum', 'mean', 'max', 'min'],
        'Clicks': ['sum', 'mean', 'max', 'min']
    }).reset_index()
```

```

        grouped_filtered_df.columns      =
['_'.join(col).strip()           for          col       in
grouped_filtered_df.columns.values]
        grouped_filtered_df.rename(columns=
{'Campaign_': 'Campaign', 'Channel_': 'Channel'}, inplace=True)

    update_grouped_treeview(grouped_filtered_df)

btn_apply_filters = tk.Button(frame_controls,
text="Apply Filters", command=apply_filters)
btn_apply_filters.pack(side=tk.LEFT,         padx=5,
pady=5)

```

- `apply_filters()`: Function that applies the filters selected by the user and updates the data views:
 - `campaign = campaign_combobox.get()`: Gets the selected campaign filter.
 - `channel = channel_combobox.get()`: Gets the selected channel filter.
 - `start_date = start_date_entry.get()`: Gets the start date.
 - `end_date = end_date_entry.get()`: Gets the end date.
 - `filtered_df`: A copy of the original DataFrame, which is then filtered based on the selected criteria.
 - `filtered_df[filtered_df['Campaign'] == campaign]`: Filters data by campaign if it's not 'All'.
 - `filtered_df[filtered_df['Channel'] == channel]`: Filters data by channel if it's not 'All'.
 - `filtered_df[filtered_df['Date'] >= start_date]`: Filters data by start date if specified.
 - `filtered_df[filtered_df['Date'] <= end_date]`: Filters data by end date if specified.

- `update_raw_treeview(filtered_df)`: Updates the raw data Treeview with the filtered data.
- `grouped_filtered_df`: Groups and aggregates the filtered data.
 - `groupby(['Campaign', 'Channel'])`: Groups by campaign and channel.
 - `agg()`: Aggregates sales and clicks with sum, mean, max, and min.
 - `reset_index()`: Resets the index for the aggregated DataFrame.
- `update_grouped_treeview(grouped_filtered_df)`: Updates the grouped data Treeview with the aggregated data.
- `btn_apply_filters`: A button to apply the filters. The command parameter binds the button to the `apply_filters` function.

Function to Save Data to Excel

```
# Function to save filtered data to Excel
def save_to_excel():
    filtered_data = tree_raw.get_children()
    if not filtered_data:
        messagebox.showwarning("Warning", "No data
to save.")
        return
    file_path =
        filedialog.asksaveasfilename(defaultextension=".xl
sx", filetypes=[("Excel files", "*.xlsx")])
    if file_path:
        df_filtered =
            pd.DataFrame([tree_raw.item(item)['values']
              for item in filtered_data], columns=df.columns)
```

```

        df_filtered.to_excel(file_path,
index=False)
    messagebox.showinfo("Info", f"Data saved
to {file_path}")

btn_save = tk.Button(frame_controls, text="Save to
Excel", command=save_to_excel)
btn_save.pack(side=tk.LEFT, padx=5, pady=5)

```

- `save_to_excel()`: Function to save the filtered data to an Excel file:
 - `filtered_data = tree_raw.get_children()`: Retrieves the data from the raw data Treeview.
 - `if not filtered_data`: Checks if there is any data to save and shows a warning if not.
 - `filedialog.asksaveasfilename()`: Opens a file dialog to select the save location and file name.
 - `df_filtered = pd.DataFrame([tree_raw.item(item)
['values'] for item in filtered_data], columns=df.columns)`: Converts the Treeview data into a DataFrame.
 - `df_filtered.to_excel(file_path, index=False)`: Saves the DataFrame to an Excel file at the specified path.
 - `messagebox.showinfo("Info", f"Data saved to
{file_path}")`: Shows an informational message after saving the file.
- `btn_save`: A button to trigger the `save_to_excel` function.

Data Visualization

```

# Create a frame for data visualization
frame_plot = tk.Frame(root, bg='#f8f9f9') # Light
grey background

```

```
frame_plot.pack(fill=tk.BOTH, expand=True)

# Function to plot data
def plot_data():
    # Clear the previous plot
    for widget in frame_plot.winfo_children():
        widget.destroy()

    campaign_counts = df['Campaign'].value_counts()

    fig, ax = plt.subplots(figsize=(10, 7))

    # Generate a color for each bar
    num_colors = len(campaign_counts)
    colors = cm.viridis(np.linspace(0, 1, num_colors)) # Using a color map

    bars = ax.bar(campaign_counts.index,
                  campaign_counts.values, color=colors)

    ax.set_title('Number of Records per Campaign')
    ax.set_xlabel('Campaign')
    ax.set_ylabel('Number of Records')
    ax.tick_params(axis='x', rotation=90)

    canvas = FigureCanvasTkAgg(fig,
                               master=frame_plot)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,
                               expand=True)

btn_plot = tk.Button(frame_controls, text="Plot Data",
                     command=plot_data)
btn_plot.pack(side=tk.LEFT, padx=5, pady=5)
```

- frame_plot: Container for the plot with a light grey background.
- plot_data(): Function to create and display a bar plot of campaign counts:
 - for widget in frame_plot.winfo_children(): Clears any existing widgets in the plot frame.
 - campaign_counts = df['Campaign'].value_counts(): Calculates the number of records for each campaign.
 - fig, ax = plt.subplots(figsize=(10, 7)): Creates a Matplotlib figure and axis with specified size.
 - colors = cm.viridis(np.linspace(0, 1, num_colors)): Generates a color map for the bars.
 - bars = ax.bar(campaign_counts.index, campaign_counts.values, color=colors): Creates a bar chart with colors.
 - ax.set_title(): Sets the plot title.
 - ax.set_xlabel(): Sets the x-axis label.
 - ax.set_ylabel(): Sets the y-axis label.
 - ax.tick_params(axis='x', rotation=90): Rotates x-axis labels for better readability.
 - FigureCanvasTkAgg(fig, master=frame_plot): Embeds the Matplotlib figure in the Tkinter frame.
 - canvas.draw(): Renders the figure.
 - canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True): Packs the canvas widget in the frame.
- btn_plot: A button to trigger the plot_data function.

Main Loop

```
# Start the Tkinter event loop
root.mainloop()
```

`root.mainloop()`: Starts the Tkinter event loop, which keeps the application running and responsive to user interactions.

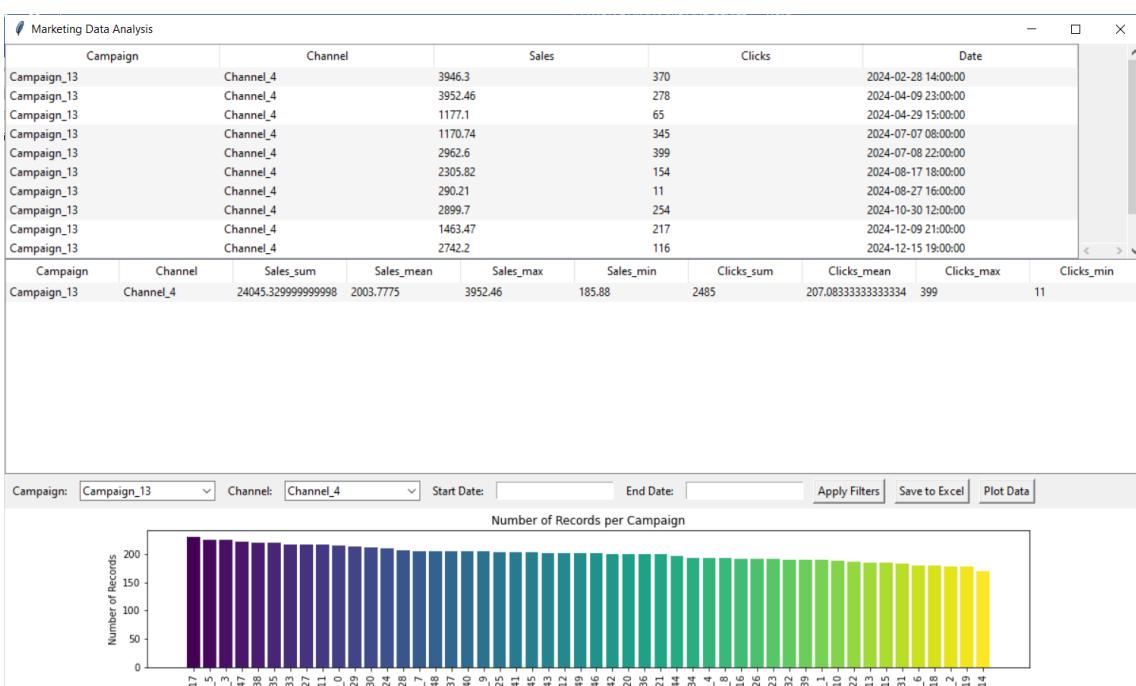
| Marketing Data Analysis | | | | | | | | | |
|-------------------------|------------|-----------|--------------------|---------------------|-------------|------------|--------------------|---------------|------------|
| Campaign | | Channel | | Sales | | Clicks | | Date | |
| Campaign_44 | Channel_9 | 1201.26 | 452 | 2024-01-01 00:00:00 | | | | | |
| Campaign_47 | Channel_4 | 542.42 | 69 | 2024-01-01 01:00:00 | | | | | |
| Campaign_0 | Channel_7 | 4102.74 | 426 | 2024-01-01 02:00:00 | | | | | |
| Campaign_3 | Channel_7 | 2619.01 | 327 | 2024-01-01 03:00:00 | | | | | |
| Campaign_3 | Channel_7 | 1393.35 | 471 | 2024-01-01 04:00:00 | | | | | |
| Campaign_39 | Channel_0 | 1064.04 | 144 | 2024-01-01 05:00:00 | | | | | |
| Campaign_9 | Channel_14 | 363.69 | 426 | 2024-01-01 06:00:00 | | | | | |
| Campaign_19 | Channel_14 | 1650.48 | 204 | 2024-01-01 07:00:00 | | | | | |
| Campaign_21 | Channel_3 | 1352.4 | 483 | 2024-01-01 08:00:00 | | | | | |
| Campaign_36 | Channel_12 | 917.74 | 432 | 2024-01-01 09:00:00 | | | | | |
| Campaign_23 | Channel_18 | 2276.6 | 142 | 2024-01-01 10:00:00 | | | | | |
| Campaign_6 | Channel_10 | 2906.89 | 136 | 2024-01-01 11:00:00 | | | | | |
| Campaign_24 | Channel_4 | 386.14 | 224 | 2024-01-01 12:00:00 | | | | | |
| Campaign | | Sales_sum | Sales_mean | Sales_max | Sales_min | Clicks_sum | Clicks_mean | Clicks_max | Clicks_min |
| Campaign_0 | Channel_0 | 13302.5 | 3325.625 | 4560.37 | 2167.47 | 1095 | 273.75 | 482 | 134 |
| Campaign_0 | Channel_1 | 9876.58 | 2469.145 | 3966.59 | 635.88 | 664 | 166.0 | 372 | 10 |
| Campaign_0 | Channel_10 | 36435.61 | 2602.5435714285713 | 4961.06 | 129.82 | 3142 | 224.42857142857142 | 443 | 44 |
| Campaign_0 | Channel_11 | 45687.91 | 2687.524117647059 | 4955.2 | 232.33 | 4039 | 237.58823529411765 | 433 | 74 |
| Campaign_0 | Channel_12 | 26657.57 | 2961.95222222222 | 4932.23 | 550.17 | 2251 | 250.11111111111111 | 472 | 12 |
| Campaign_0 | Channel_13 | 19367.49 | 3227.915000000004 | 4682.71 | 448.61 | 2037 | 339.5 | 468 | 176 |
| Campaign_0 | Channel_14 | 45678.48 | 2854.905 | 4495.67 | 126.71 | 4397 | 274.8125 | 487 | 47 |
| Campaign_0 | Channel_15 | 13745.73 | 1963.6757142857143 | 3050.83 | 681.65 | 1640 | 234.28571428571428 | 492 | 35 |
| Campaign_0 | Channel_16 | 21805.04 | 1982.27636363636 | 4865.9 | 163.28 | 3315 | 301.3636363636364 | 441 | 83 |
| Campaign_0 | Channel_17 | 16223.09 | 1622.309 | 4920.29 | 312.33 | 2820 | 282.0 | 496 | 13 |
| Campaign_0 | Channel_18 | 36396.78 | 3639.678 | 4858.71 | 416.03 | 3239 | 323.9 | 490 | 84 |
| Campaign_0 | Channel_19 | 18410.98 | 2630.14 | 3876.2 | 1050.28 | 1687 | 241.0 | 448 | 10 |
| Campaign_0 | Channel_2 | 27942.5 | 2794.25 | 4986.56 | 847.19 | 2893 | 289.3 | 498 | 106 |
| Campaign: | | All | Channel: | All | Start Date: | End Date: | Apply Filters | Save to Excel | Plot Data |

| Marketing Data Analysis | | | | | | | | | |
|-------------------------|------------|-------------------|--------------------|---------------------|-------------|------------|--------------------|---------------|------------|
| Campaign | | Channel | | Sales | | Clicks | | Date | |
| Campaign_13 | Channel_10 | 956.37 | 496 | 2024-01-02 00:00:00 | | | | | |
| Campaign_13 | Channel_14 | 1347.36 | 435 | 2024-01-05 00:00:00 | | | | | |
| Campaign_13 | Channel_0 | 4490.66 | 104 | 2024-01-05 20:00:00 | | | | | |
| Campaign_13 | Channel_15 | 1731.85 | 202 | 2024-01-07 10:00:00 | | | | | |
| Campaign_13 | Channel_18 | 4701.83 | 57 | 2024-01-07 18:00:00 | | | | | |
| Campaign_13 | Channel_12 | 136.26 | 343 | 2024-01-08 12:00:00 | | | | | |
| Campaign_13 | Channel_11 | 3639.33 | 303 | 2024-01-09 06:00:00 | | | | | |
| Campaign_13 | Channel_14 | 1706.88 | 353 | 2024-01-15 17:00:00 | | | | | |
| Campaign_13 | Channel_8 | 1886.05 | 268 | 2024-01-16 13:00:00 | | | | | |
| Campaign_13 | Channel_7 | 1304.87 | 52 | 2024-01-18 02:00:00 | | | | | |
| Campaign_13 | Channel_17 | 3573.47 | 469 | 2024-01-18 07:00:00 | | | | | |
| Campaign_13 | Channel_11 | 2234.76 | 257 | 2024-01-24 19:00:00 | | | | | |
| Campaign_13 | Channel_11 | 2272.08 | 486 | 2024-01-25 17:00:00 | | | | | |
| Campaign | | Sales_sum | Sales_mean | Sales_max | Sales_min | Clicks_sum | Clicks_mean | Clicks_max | Clicks_min |
| Campaign_13 | Channel_0 | 25340.66 | 2815.628888888889 | 4490.66 | 1470.34 | 2430 | 270.0 | 498 | 51 |
| Campaign_13 | Channel_1 | 28376.13 | 2026.8664285714287 | 4091.36 | 202.81 | 2683 | 191.64285714285714 | 486 | 11 |
| Campaign_13 | Channel_10 | 12443.03 | 1777.5757142857144 | 4602.83 | 331.06 | 1653 | 236.14285714285714 | 496 | 34 |
| Campaign_13 | Channel_11 | 16658.15 | 2379.7357142857145 | 3639.33 | 853.27 | 1836 | 262.2857142857143 | 486 | 37 |
| Campaign_13 | Channel_12 | 29383.13 | 2671.193636363636 | 4938.49 | 136.26 | 3996 | 363.2727272727275 | 493 | 125 |
| Campaign_13 | Channel_13 | 14378.8 | 2054.1142857142854 | 3612.68 | 785.99 | 1855 | 265.0 | 492 | 54 |
| Campaign_13 | Channel_14 | 32761.32 | 3640.1466666666665 | 4985.12 | 1347.36 | 2551 | 283.44444444444446 | 486 | 22 |
| Campaign_13 | Channel_15 | 17898.77 | 2237.34625 | 4472.74 | 226.01 | 1689 | 211.125 | 466 | 63 |
| Campaign_13 | Channel_16 | 35994.56 | 3599.455999999999 | 4819.45 | 1071.71 | 3245 | 324.5 | 475 | 176 |
| Campaign_13 | Channel_17 | 20076.82999999999 | 2230.758888888886 | 4117.11 | 262.24 | 2738 | 304.2222222222223 | 469 | 62 |
| Campaign_13 | Channel_18 | 33099.14 | 2546.087692307692 | 4711.77 | 146.3 | 3882 | 298.61538461538464 | 495 | 57 |
| Campaign_13 | Channel_19 | 25542.64 | 3192.83 | 4519.86 | 2137.96 | 2558 | 319.75 | 462 | 39 |
| Campaign_13 | Channel_2 | 23708.27 | 2963.53375 | 4548.08 | 1262.64 | 1495 | 186.875 | 472 | 29 |
| Campaign: | | Campaign_13 | Channel: | All | Start Date: | End Date: | Apply Filters | Save to Excel | Plot Data |

Marketing Data Analysis

| Campaign | Channel | Sales | Clicks | Date |
|-------------|-----------|---------|--------|---------------------|
| Campaign_13 | Channel_4 | 3946.3 | 370 | 2024-02-28 14:00:00 |
| Campaign_13 | Channel_4 | 3952.46 | 278 | 2024-04-09 23:00:00 |
| Campaign_13 | Channel_4 | 1177.1 | 65 | 2024-04-29 15:00:00 |
| Campaign_13 | Channel_4 | 1170.74 | 345 | 2024-07-07 08:00:00 |
| Campaign_13 | Channel_4 | 2962.6 | 399 | 2024-07-08 22:00:00 |
| Campaign_13 | Channel_4 | 2305.82 | 154 | 2024-08-17 18:00:00 |
| Campaign_13 | Channel_4 | 290.21 | 11 | 2024-08-27 16:00:00 |
| Campaign_13 | Channel_4 | 2899.7 | 254 | 2024-10-30 12:00:00 |
| Campaign_13 | Channel_4 | 1463.47 | 217 | 2024-12-09 21:00:00 |
| Campaign_13 | Channel_4 | 2742.2 | 116 | 2024-12-15 19:00:00 |
| Campaign_13 | Channel_4 | 185.88 | 27 | 2025-01-05 14:00:00 |
| Campaign_13 | Channel_4 | 948.85 | 249 | 2025-01-10 23:00:00 |

| Campaign | Channel | Sales_sum | Sales_mean | Sales_max | Sales_min | Clicks_sum | Clicks_mean | Clicks_max | Clicks_min |
|-------------|-----------|--------------------|------------|-----------|-----------|------------|--------------------|------------|------------|
| Campaign_13 | Channel_4 | 24045.329999999998 | 2003.7775 | 3952.46 | 185.88 | 2485 | 207.08333333333334 | 399 | 11 |



EXAMPLE 4.9

Advanced Grouping Multiple Columns and Multiple Aggregations with Synthetic Weather Data

This code creates a synthetic weather dataset for analysis and demonstration purposes. It generates a large dataset containing records of weather conditions across various cities, including variables such as temperature, humidity, and weather conditions over time. By using random sampling and predefined ranges, the code ensures that the dataset is diverse and representative of various weather scenarios. The dataset includes 10,000 records, each with a timestamp, city name, temperature, humidity, and weather condition, creating a comprehensive view of weather patterns.

The code further processes the dataset by grouping it based on city and weather condition, then aggregating key metrics like average, maximum, and minimum values for temperature and humidity. This aggregation helps summarize the data, making it easier to analyze and interpret. Finally, the aggregated results are saved to an Excel file, providing a structured format for further analysis or reporting. This approach allows users to efficiently explore and visualize weather data trends and conditions across different cities.

```
import pandas as pd
import numpy as np

# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000

# Define a set of cities and weather conditions
cities = [f'City_{i}' for i in range(50)]
```

```
weather_conditions = ['Sunny', 'Cloudy', 'Rainy',
'Snowy', 'Windy']

# Create synthetic weather dataset
data = {
    'City': np.random.choice(cities,
size=num_records),
    'Date': pd.date_range(start='2023-01-01',
periods=num_records, freq='H'),
    'Temperature': np.random.uniform(-10, 35,
size=num_records).round(1),
    'Humidity': np.random.uniform(20, 100,
size=num_records).round(1),
    'Condition':
np.random.choice(weather_conditions,
size=num_records)
}
df = pd.DataFrame(data)

# Group by City and Weather Condition, then
aggregate Temperature and Humidity
grouped_df = df.groupby(['City',
'Condition']).agg({
    'Temperature': ['mean', 'max', 'min'],
    'Humidity': ['mean', 'max', 'min']
}).reset_index()

# Flatten the MultiIndex columns
grouped_df.columns = ['_'.join(col).strip() for
col in grouped_df.columns.values]
grouped_df.rename(columns={'City_': 'City',
'Condition_': 'Condition'}, inplace=True)

# Save the aggregated DataFrame to an Excel file
output_file = 'weather_data_summary.xlsx'
```

```
grouped_df.to_excel(output_file,  
sheet_name='Weather Summary', index=False)  
  
print(f"Data saved to {output_file}")
```

Here's a detailed explanation of each part of the code that creates a synthetic weather dataset, performs advanced grouping and aggregation, and saves the results to an Excel file:

Import Libraries:

```
import pandas as pd  
import numpy as np
```

- pandas is used for data manipulation and analysis. It provides data structures like DataFrames for handling tabular data.
- numpy is used for numerical operations, particularly for generating random data and performing mathematical operations.

Set Seed for Reproducibility:

```
np.random.seed(0)
```

- Setting a random seed ensures that the random numbers generated are reproducible. This is useful for debugging and comparing results across different runs.

Define the Number of Records:

```
num_records = 10000
```

- num_records specifies the number of records (rows) to be generated in the synthetic dataset. Here, we generate 10,000 records.

Define a Set of Cities and Weather Conditions:

```
cities = [f'City_{i}' for i in range(50)]
weather_conditions = ['Sunny', 'Cloudy', 'Rainy',
'Snowy', 'Windy']
```

- cities is a list of city names formatted as 'City_0', 'City_1', ..., 'City_49'. This list contains 50 city names.
- weather_conditions is a list of possible weather conditions that can appear in the dataset.

Create Synthetic Weather Dataset:

```
data = {
    'City': np.random.choice(cities,
size=num_records),
    'Date': pd.date_range(start='2023-01-01',
periods=num_records, freq='H'),
    'Temperature': np.random.uniform(-10, 35,
size=num_records).round(1),
    'Humidity': np.random.uniform(20, 100,
size=num_records).round(1),
    'Condition': np.random.choice(weather_conditions,
size=num_records)
}
df = pd.DataFrame(data)
```

- data is a dictionary where each key corresponds to a column in the DataFrame:

- 'City': Randomly selects city names from the cities list.
- 'Date': Creates a date range starting from January 1, 2023, with hourly frequency, resulting in 10,000 timestamps.
- 'Temperature': Generates random temperatures between -10 and 35 degrees Celsius, rounded to one decimal place.
- 'Humidity': Generates random humidity values between 20% and 100%, rounded to one decimal place.
- 'Condition': Randomly selects weather conditions from the weather_conditions list.
- df is a DataFrame created from the data dictionary.

Group by City and Weather Condition, Then Aggregate:

```
grouped_df = df.groupby(['City',  
'Condition']).agg({  
    'Temperature': ['mean', 'max', 'min'],  
    'Humidity': ['mean', 'max', 'min']  
}).reset_index()
```

- The DataFrame df is grouped by City and Condition.
- Aggregation functions are applied to Temperature and Humidity:
 - mean: Average value.
 - max: Maximum value.
 - min: Minimum value.
- reset_index() is used to convert the grouped indices (City and Condition) back into columns.

Flatten the MultiIndex Columns:

```
grouped_df.columns = ['_'.join(col).strip() for col in grouped_df.columns.values]
grouped_df.rename(columns={'City_': 'City',
                           'Condition_': 'Condition'}, inplace=True)
```

- After aggregation, columns have a MultiIndex (e.g., Temperature_mean, Temperature_max).
- grouped_df.columns flattens this MultiIndex into single-level column names using '_'.join(col).strip().
- rename() is used to correct column names, ensuring that City_ and Condition_ are renamed to City and Condition.

Save the Aggregated DataFrame to an Excel File:

```
output_file = 'weather_data_summary.xlsx'
grouped_df.to_excel(output_file,
                     sheet_name='Weather Summary', index=False)
```

- output_file specifies the name of the Excel file to save the data.
- to_excel() method saves the grouped_df DataFrame to an Excel file with the sheet name 'Weather Summary'.
- index=False ensures that the DataFrame index is not included in the Excel file.

Print Confirmation:

```
print(f"Data saved to {output_file}")
```

- Prints a confirmation message indicating that the data has been saved successfully.

This detailed breakdown outlines each step of the process for generating, manipulating, and saving synthetic weather data.

EXAMPLE 4.10

GUI Tkinter Advanced Grouping Multiple Columns and Multiple Aggregations with Synthetic Weather Data

The purpose of this project is to create a comprehensive graphical user interface (GUI) application using Tkinter that allows users to interact with a synthetic weather dataset. This dataset, generated with a high volume of records, simulates weather conditions across various cities and dates. The application is designed to provide users with several functionalities: displaying raw and aggregated data, allowing data input, and visualizing data through plots. The GUI aims to offer a user-friendly experience for exploring and analyzing weather data.

In detail, the application consists of multiple tabs, each serving a specific function. The "Raw Data" tab displays the original dataset in a scrollable table, making it easy to view and interact with the full data. The "Aggregated Data" tab provides a summary view of the dataset, showing aggregated statistics such as the mean, max, and min values of temperature and humidity for each city and weather condition. This tab helps users quickly grasp key metrics and trends in the data.

The "Input Data" tab enables users to manually add new weather records to the dataset. This feature includes fields for entering city, date, temperature, humidity, and weather condition, and updates

both the raw and aggregated data views upon submission. Finally, the "Plot Data" tab offers a visual representation of the data, with a bar chart illustrating the number of records per city. This graphical representation, enhanced with a color map, provides an intuitive way to analyze the distribution of data across cities. Overall, the project aims to provide a rich and interactive tool for managing and visualizing weather data, making it accessible and useful for various analytical purposes.

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.cm as cm

# Set a seed for reproducibility
np.random.seed(0)

# Define the number of records
num_records = 10000

# Define a set of cities and weather conditions
cities = [f'City_{i}' for i in range(50)]
weather_conditions = ['Sunny', 'Cloudy', 'Rainy',
'Snowy', 'Windy']

# Create synthetic weather dataset
data = {
    'City': np.random.choice(cities,
size=num_records),
    'Date': pd.date_range(start='2023-01-01',
periods=num_records, freq='H'),
```

```
'Temperature': np.random.uniform(-10, 35,
size=num_records).round(1),
'Humidity': np.random.uniform(20, 100,
size=num_records).round(1),
'Condition':
np.random.choice(weather_conditions,
size=num_records)
}
df = pd.DataFrame(data)

# Group by City and Weather Condition, then
# aggregate Temperature and Humidity
grouped_df = df.groupby(['City',
'Condition']).agg({
    'Temperature': ['mean', 'max', 'min'],
    'Humidity': ['mean', 'max', 'min']
}).reset_index()
grouped_df.columns = ['_'.join(col).strip() for
col in grouped_df.columns.values]
grouped_df.rename(columns={'City_': 'City',
'Condition_': 'Condition'}, inplace=True)

# Create the main application window
root = tk.Tk()
root.title("Weather Data Analysis")
root.geometry("1200x800")

# Create a notebook for tabs
notebook = ttk.Notebook(root)
notebook.pack(fill=tk.BOTH, expand=True)

# Tab 1: Raw Data
tab_raw = ttk.Frame(notebook)
notebook.add(tab_raw, text="Raw Data")
```

```
# Create a Treeview widget for displaying the raw DataFrame
tree_raw = ttk.Treeview(tab_raw,
columns=list(df.columns), show='headings')
tree_raw.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Scrollbars for the raw data Treeview
vsb_raw = ttk.Scrollbar(tab_raw,
orient="vertical", command=tree_raw.yview)
vsb_raw.pack(side='right', fill='y')
tree_raw.configure(yscrollcommand=vsb_raw.set)

hsb_raw = ttk.Scrollbar(tab_raw,
orient="horizontal", command=tree_raw.xview)
hsb_raw.pack(side='bottom', fill='x')
tree_raw.configure(xscrollcommand=hsb_raw.set)

# Define the columns in the raw data Treeview
for col in df.columns:
    tree_raw.heading(col, text=col)
    tree_raw.column(col, width=120)

# Function to update the raw data Treeview with alternating row colors
def update_raw_treeview(dataframe):
    for i in tree_raw.get_children():
        tree_raw.delete(i)
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree_raw.insert("", "end",
values=list(row), tags=(tag,))
        tree_raw.tag_configure('even',
background="#f5f5f5") # Light grey
        tree_raw.tag_configure('odd',
background="#ffffff") # White
```

```
update_raw_treeview(df)

# Tab 2: Aggregated Data
tab_grouped = ttk.Frame(notebook)
notebook.add(tab_grouped, text="Aggregated Data")

# Create a Treeview widget for displaying the
grouped DataFrame
tree_grouped = ttk.Treeview(tab_grouped,
columns=list(grouped_df.columns), show='headings')
tree_grouped.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

# Scrollbars for the grouped data Treeview
vsb_grouped = ttk.Scrollbar(tab_grouped,
orient="vertical", command=tree_grouped.yview)
vsb_grouped.pack(side='right', fill='y')
tree_grouped.configure(yscrollcommand=vsb_grouped.
set)

hsb_grouped = ttk.Scrollbar(tab_grouped,
orient="horizontal", command=tree_grouped.xview)
hsb_grouped.pack(side='bottom', fill='x')
tree_grouped.configure(xscrollcommand=hsb_grouped.
set)

# Define the columns in the grouped data Treeview
for col in grouped_df.columns:
    tree_grouped.heading(col, text=col)
    tree_grouped.column(col, width=120)

# Function to update the grouped data Treeview
# with alternating row colors
def update_grouped_treeview(dataframe):
    for i in tree_grouped.get_children():
```

```
        tree_grouped.delete(i)
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
        tree_grouped.insert("", "end",
values=list(row), tags=(tag,))
        tree_grouped.tag_configure('even',
background='#f5f5f5') # Light grey
        tree_grouped.tag_configure('odd',
background='#ffffff') # White

update_grouped_treeview(grouped_df)

# Tab 3: Input Data
tab_input = ttk.Frame(notebook)
notebook.add(tab_input, text="Input Data")

# Create widgets for data input
tk.Label(tab_input, text="City:").grid(row=0,
column=0, padx=5, pady=5, sticky=tk.E)
city_entry = tk.Entry(tab_input)
city_entry.grid(row=0, column=1, padx=5, pady=5)

tk.Label(tab_input, text="Date (YYYY-MM-DD
HH:MM:SS):").grid(row=1, column=0, padx=5, pady=5,
sticky=tk.E)
date_entry = tk.Entry(tab_input)
date_entry.grid(row=1, column=1, padx=5, pady=5)

tk.Label(tab_input,
text="Temperature:").grid(row=2, column=0, padx=5,
pady=5, sticky=tk.E)
temperature_entry = tk.Entry(tab_input)
temperature_entry.grid(row=2, column=1, padx=5,
pady=5)
```

```
tk.Label(tab_input, text="Humidity:").grid(row=3, column=0, padx=5, pady=5, sticky=tk.E)
humidity_entry = tk.Entry(tab_input)
humidity_entry.grid(row=3, column=1, padx=5, pady=5)

tk.Label(tab_input, text="Condition:").grid(row=4, column=0, padx=5, pady=5, sticky=tk.E)
condition_combobox = ttk.Combobox(tab_input, values=['Sunny', 'Cloudy', 'Rainy', 'Snowy', 'Windy'])
condition_combobox.set('Sunny')
condition_combobox.grid(row=4, column=1, padx=5, pady=5)

def add_data():
    try:
        city = city_entry.get()
        date = pd.to_datetime(date_entry.get())
        temperature =
float(temperature_entry.get())
        humidity = float(humidity_entry.get())
        condition = condition_combobox.get()

        # Append new data to the DataFrame
        global df
        new_data = pd.DataFrame({
            'City': [city],
            'Date': [date],
            'Temperature': [temperature],
            'Humidity': [humidity],
            'Condition': [condition]
        })
        df = pd.concat([df, new_data],
ignore_index=True)
```

```

# Update Treeviews
update_raw_treeview(df)

# Update grouped data
grouped_df = df.groupby(['City',
'Condition']).agg({
    'Temperature': ['mean', 'max', 'min'],
    'Humidity': ['mean', 'max', 'min']
}).reset_index()
grouped_df.columns =
['_'.join(col).strip() for col in
grouped_df.columns.values]
grouped_df.rename(columns={'City_':
'City', 'Condition_': 'Condition'}, inplace=True)
update_grouped_treeview(grouped_df)

messagebox.showinfo("Success", "Data added
successfully.")
except Exception as e:
    messagebox.showerror("Error", f"Failed to
add data: {e}")

btn_add_data = tk.Button(tab_input, text="Add
Data", command=add_data)
btn_add_data.grid(row=5, column=1, padx=5, pady=5,
sticky=tk.E)

# Tab 4: Plot Data
tab_plot = ttk.Frame(notebook)
notebook.add(tab_plot, text="Plot Data")

# Create a frame for data visualization
frame_plot = tk.Frame(tab_plot, bg="#f8f9f9") #
Light grey background
frame_plot.pack(fill=tk.BOTH, expand=True)

```

```
def plot_data():
    # Clear the previous plot
    for widget in frame_plot.winfo_children():
        widget.destroy()

    city_counts = df['City'].value_counts()

    fig, ax = plt.subplots(figsize=(10, 7))

    # Generate a color for each bar
    num_colors = len(city_counts)
    colors = cm.viridis(np.linspace(0, 1,
num_colors)) # Using a color map

    bars = ax.bar(city_counts.index,
city_counts.values, color=colors)

    ax.set_title('Number of Records per City')
    ax.set_xlabel('City')
    ax.set_ylabel('Number of Records')
    ax.tick_params(axis='x', rotation=90)

    canvas = FigureCanvasTkAgg(fig,
master=frame_plot)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,
expand=True)

btn_plot_data = tk.Button(tab_plot, text="Plot
Data", command=plot_data)
btn_plot_data.pack(side=tk.LEFT, padx=5, pady=5)

# Start the Tkinter event loop
root.mainloop()
```

Here's a detailed explanation of each part of the provided Tkinter GUI application code:

Imports and Setup

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from    matplotlib.backends.backend_tkagg    import
FigureCanvasTkAgg
import matplotlib.cm as cm
```

- **tkinter** and **ttk**: Modules from the Tkinter library for creating the graphical user interface (GUI). **tkinter** is used for the basic GUI components, while **ttk** provides access to more advanced widgets.
- **filedialog** and **messagebox**: **filedialog** allows file selection dialogs, and **messagebox** is used to show informational or error messages.
- **pandas**: Library for data manipulation and analysis, used here to handle the synthetic weather dataset.
- **numpy**: Library for numerical operations, used to generate random data.
- **matplotlib.pyplot**: Used for plotting data and generating visualizations.
- **FigureCanvasTkAgg**: Integrates Matplotlib plots with Tkinter widgets.
- **matplotlib.cm**: Provides colormaps for enhancing plot visuals.

Data Generation

```
np.random.seed(0)

num_records = 10000
cities = [f'City_{i}' for i in range(50)]
weather_conditions = ['Sunny', 'Cloudy', 'Rainy',
'Snowy', 'Windy']

data = {
    'City': np.random.choice(cities,
size=num_records),
    'Date': pd.date_range(start='2023-01-01',
periods=num_records, freq='H'),
    'Temperature': np.random.uniform(-10, 35,
size=num_records).round(1),
    'Humidity': np.random.uniform(20, 100,
size=num_records).round(1),
    'Condition': np.random.choice(weather_conditions,
size=num_records)
}
df = pd.DataFrame(data)
```

- `np.random.seed(0)`: Sets the random seed for reproducibility, ensuring that the same random numbers are generated each time the code is run.
- `num_records`: Defines the number of records in the dataset.
- `cities`: Generates a list of city names using a list comprehension.
- `weather_conditions`: Defines possible weather conditions.
- `data`: Creates a dictionary where each key corresponds to a column in the DataFrame. Data is generated using

random choices and distributions.

- df: Creates a DataFrame from the dictionary, representing the synthetic weather dataset.

Data Aggregation

```
grouped_df = df.groupby(['City',  
'Condition']).agg({  
    'Temperature': ['mean', 'max', 'min'],  
    'Humidity': ['mean', 'max', 'min']  
}).reset_index()  
grouped_df.columns = ['_'.join(col).strip() for  
col in grouped_df.columns.values]  
grouped_df.rename(columns={'City_': 'City',  
'Condition_': 'Condition'}, inplace=True)
```

- df.groupby(['City', 'Condition']): Groups the data by city and weather condition.
- .agg(): Aggregates data by calculating the mean, maximum, and minimum of temperature and humidity for each group.
- reset_index(): Resets the index of the grouped DataFrame to default integer index.
- grouped_df.columns: Flattens the MultiIndex columns resulting from the aggregation.
- rename(columns=...): Renames columns for clarity, removing the unnecessary level in the column names.

GUI Setup

```
root = tk.Tk()
```

```
root.title("Weather Data Analysis")
root.geometry("1200x800")

notebook = ttk.Notebook(root)
notebook.pack(fill=tk.BOTH, expand=True)
```

- tk.Tk(): Creates the main application window.
- title("Weather Data Analysis"): Sets the title of the window.
- geometry("1200x800"): Defines the size of the window.
- ttk.Notebook(root): Creates a notebook widget to hold multiple tabs.
- pack(fill=tk.BOTH, expand=True): Configures the notebook to expand and fill available space.

Tab 1: Raw Data

```
tab_raw = ttk.Frame(notebook)
notebook.add(tab_raw, text="Raw Data")

tree_raw = ttk.Treeview(tab_raw,
columns=list(df.columns), show='headings')
tree_raw.pack(side=tk.LEFT, fill=tk.BOTH,
expand=True)

vsb_raw = ttk.Scrollbar(tab_raw,
orient="vertical", command=tree_raw.yview)
vsb_raw.pack(side='right', fill='y')
tree_raw.configure(yscrollcommand=vsb_raw.set)

hsb_raw = ttk.Scrollbar(tab_raw,
orient="horizontal", command=tree_raw.xview)
hsb_raw.pack(side='bottom', fill='x')
tree_raw.configure(xscrollcommand=hsb_raw.set)
```

```
for col in df.columns:  
    tree_raw.heading(col, text=col)  
    tree_raw.column(col, width=120)
```

- `ttk.Frame(notebook)`: Creates a frame to hold widgets for the "Raw Data" tab.
- `notebook.add(tab_raw, text="Raw Data")`: Adds the tab to the notebook with the title "Raw Data".
- `ttk.Treeview(tab_raw, columns=list(df.columns), show='headings')`: Creates a Treeview widget to display the raw data.
- `pack(side=tk.LEFT, fill=tk.BOTH, expand=True)`: Configures the Treeview to fill the available space.
- `ttk.Scrollbar(...)`: Adds vertical and horizontal scrollbars to handle large datasets.
- `tree_raw.heading(col, text=col)`: Sets the column headings in the Treeview.
- `tree_raw.column(col, width=120)`: Sets the column width.

Update Raw Data Treeview

```
def update_raw_treeview(dataframe):  
    for i in tree_raw.get_children():  
        tree_raw.delete(i)  
    for index, row in dataframe.iterrows():  
        tag = 'even' if index % 2 == 0 else 'odd'  
                tree_raw.insert("", "end",  
values=list(row), tags=(tag,))  
                tree_raw.tag_configure('even',  
background='#f5f5f5') # Light grey  
                tree_raw.tag_configure('odd',  
background='#ffffff') # White
```

```
update_raw_treeview(df)
```

- update_raw_treeview(dataframe): Function to update the Treeview with new data.
- tree_raw.get_children(): Retrieves all items currently in the Treeview.
- tree_raw.delete(i): Deletes existing items.
- tree_raw.insert("", "end", values=list(row), tags=(tag,)): Inserts new rows into the Treeview with alternating row colors for better readability.
- tag_configure(...): Configures row colors for alternating rows.

Tab 2: Aggregated Data

```
tab_grouped = ttk.Frame(notebook)
notebook.add(tab_grouped, text="Aggregated Data")

tree_grouped      =      ttk.Treeview(tab_grouped,
columns=list(grouped_df.columns), show='headings')
tree_grouped.pack(side=tk.LEFT,      fill=tk.BOTH,
expand=True)

vsb_grouped      =      ttk.Scrollbar(tab_grouped,
orient="vertical", command=tree_grouped.yview)
vsb_grouped.pack(side='right', fill='y')
tree_grouped.configure(yscrollcommand=vsb_grouped.
set)

hsb_grouped      =      ttk.Scrollbar(tab_grouped,
orient="horizontal", command=tree_grouped.xview)
hsb_grouped.pack(side='bottom', fill='x')
```

```

tree_grouped.configure(xscrollcommand=hsb_grouped.set)

for col in grouped_df.columns:
    tree_grouped.heading(col, text=col)
    tree_grouped.column(col, width=120)

```

- `ttk.Frame(notebook)`: Creates a frame for the "Aggregated Data" tab.
- `notebook.add(tab_grouped, text="Aggregated Data")`: Adds the "Aggregated Data" tab to the notebook.
- `ttk.Treeview(tab_grouped, columns=list(grouped_df.columns), show='headings')`: Creates a Treeview widget for displaying aggregated data.
- `ttk.Scrollbar(...)`: Adds scrollbars to the Treeview.
- `tree_grouped.heading(col, text=col)`: Sets column headings for the aggregated data.
- `tree_grouped.column(col, width=120)`: Sets the column width for the aggregated data Treeview.

Update Grouped Data Treeview

```

def update_grouped_treeview(dataframe):
    for i in tree_grouped.get_children():
        tree_grouped.delete(i)
    for index, row in dataframe.iterrows():
        tag = 'even' if index % 2 == 0 else 'odd'
                tree_grouped.insert("", "end",
values=list(row), tags=(tag,))
                tree_grouped.tag_configure('even',
background='#f5f5f5') # Light grey
                tree_grouped.tag_configure('odd',
background='#ffffff') # White

```

```
update_grouped_treeview(grouped_df)
```

- `update_grouped_treeview(dataframe)`: Function to update the Treeview for aggregated data.
- `tree_grouped.get_children()`: Retrieves existing items.
- `tree_grouped.delete(i)`: Deletes old items.
- `tree_grouped.insert("", "end", values=list(row), tags=(tag,))`: Inserts new rows into the Treeview with alternating row colors for clarity.
- `tag_configure(...)`: Configures row colors for readability.

Tab 3: Input Data

```
tab_input = ttk.Frame(notebook)
notebook.add(tab_input, text="Input Data")

tk.Label(tab_input, text="City:").grid(row=0, column=0, padx=5, pady=5, sticky=tk.E)
city_entry = tk.Entry(tab_input)
city_entry.grid(row=0, column=1, padx=5, pady=5)

tk.Label(tab_input, text="Date      (YYYY-MM-DD HH:MM:SS)").grid(row=1, column=0, padx=5, pady=5, sticky=tk.E)
date_entry = tk.Entry(tab_input)
date_entry.grid(row=1, column=1, padx=5, pady=5)

tk.Label(tab_input, text="Temperature:").grid(row=2, column=0, padx=5, pady=5, sticky=tk.E)
temperature_entry = tk.Entry(tab_input)
temperature_entry.grid(row=2, column=1, padx=5, pady=5)
```

```

tk.Label(tab_input,    text="Humidity:").grid(row=3,
column=0,  padx=5,  pady=5,  sticky=tk.E)
humidity_entry = tk.Entry(tab_input)
humidity_entry.grid(row=3,      column=1,      padx=5,
pady=5)

tk.Label(tab_input,    text="Condition:").grid(row=4,
column=0,  padx=5,  pady=5,  sticky=tk.E)
condition_combobox      =      ttk.Combobox(tab_input,
values=['Sunny',      'Cloudy',      'Rainy',      'Snowy',
'Windy'])
condition_combobox.set('Sunny')
condition_combobox.grid(row=4,      column=1,      padx=5,
pady=5)

```

- `ttk.Frame(notebook)`: Creates a frame for the "Input Data" tab.
- `notebook.add(tab_input, text="Input Data")`: Adds the "Input Data" tab to the notebook.
- `tk.Label(...)`: Creates labels for each input field.
- `tk.Entry(...)`: Creates text entry fields for users to input data.
- `ttk.Combobox(...)`: Creates a dropdown menu for selecting the weather condition.

Add Data Function

```

def add_data():
    try:
        city = city_entry.get()
        date = pd.to_datetime(date_entry.get())
                    temperature      =
float(temperature_entry.get())

```

```

humidity = float(humidity_entry.get())
condition = condition_combobox.get()

global df
new_data = pd.DataFrame({
    'City': [city],
    'Date': [date],
    'Temperature': [temperature],
    'Humidity': [humidity],
    'Condition': [condition]
})
df = pd.concat([df, new_data],
ignore_index=True)

update_raw_treeview(df)

grouped_df = df.groupby(['City',
'Condition']).agg({
    'Temperature': ['mean', 'max', 'min'],
    'Humidity': ['mean', 'max', 'min']
}).reset_index()
grouped_df.columns = ['_'.join(col).strip() for col in grouped_df.columns.values]
grouped_df.rename(columns={'City_':
'City', 'Condition_': 'Condition'}, inplace=True)
update_grouped_treeview(grouped_df)

messagebox.showinfo("Success", "Data added successfully.")
except Exception as e:
    messagebox.showerror("Error", f"Failed to add data: {e}")

```

- `add_data()`: Function to handle data input and update the dataset.
- `city_entry.get(), date_entry.get(), temperature_entry.get(), humidity_entry.get(), condition_combobox.get()`: Retrieve user inputs.
- `pd.to_datetime(...)`: Converts the date input to a Pandas datetime object.
- `float(...)`: Converts temperature and humidity inputs to floats.
- `pd.DataFrame({...})`: Creates a DataFrame with the new data.
- `df = pd.concat([...])`: Appends the new data to the existing DataFrame.
- `update_raw_treeview(df)`: Updates the raw data Treeview with the new data.
- `grouped_df = df.groupby([...])`: Re-calculates aggregated data.
- `update_grouped_treeview(grouped_df)`: Updates the aggregated data Treeview.
- `messagebox.showinfo(...)`: Shows a success message.
- `messagebox.showerror(...)`: Shows an error message if something goes wrong.

Tab 4: Plot Data

```
tab_plot = ttk.Frame(notebook)
notebook.add(tab_plot, text="Plot Data")

frame_plot = tk.Frame(tab_plot, bg='#f8f9f9')
frame_plot.pack(fill=tk.BOTH, expand=True)

def plot_data():
```

```

for widget in frame_plot.winfo_children():
    widget.destroy()

city_counts = df['City'].value_counts()

fig, ax = plt.subplots(figsize=(10, 7))

num_colors = len(city_counts)
colors = cm.viridis(np.linspace(0, 1, num_colors))

bars = ax.bar(city_counts.index, city_counts.values, color=colors)

ax.set_title('Number of Records per City')
ax.set_xlabel('City')
ax.set_ylabel('Number of Records')
ax.tick_params(axis='x', rotation=90)

canvas = FigureCanvasTkAgg(fig, master=frame_plot)
canvas.draw()
canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

btn_plot_data = tk.Button(tab_plot, text="Plot Data", command=plot_data)
btn_plot_data.pack(side=tk.LEFT, padx=5, pady=5)

```

- `ttk.Frame(notebook)`: Creates a frame for the "Plot Data" tab.
- `frame_plot = tk.Frame(tab_plot, bg="#f8f9f9")`: Creates a frame within the "Plot Data" tab for the plot, with a light grey background.

- `pack(fill=tk.BOTH, expand=True)`: Configures the frame to fill available space.

Plot Data Function

```
def plot_data():
    for widget in frame_plot.winfo_children():
        widget.destroy()

    city_counts = df['City'].value_counts()

    fig, ax = plt.subplots(figsize=(10, 7))

    num_colors = len(city_counts)
    colors     = cm.viridis(np.linspace(0, 1,
num_colors))

    bars      = ax.bar(city_counts.index,
city_counts.values, color=colors)

    ax.set_title('Number of Records per City')
    ax.set_xlabel('City')
    ax.set_ylabel('Number of Records')
    ax.tick_params(axis='x', rotation=90)

    canvas     = FigureCanvasTkAgg(fig,
master=frame_plot)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,
expand=True)
```

- `plot_data()`: Function to generate and display the plot.
- `frame_plot.winfo_children()`: Retrieves all widgets in the plot frame.

- `widget.destroy()`: Clears old plot widgets.
- `df['City'].value_counts()`: Counts the number of records for each city.
- `fig, ax = plt.subplots(figsize=(10, 7))`: Creates a Matplotlib figure and axes.
- `cm.viridis(np.linspace(0, 1, num_colors))`: Generates a color map for the bars.
- `ax.bar(...)`: Creates a bar chart with city counts.
- `FigureCanvasTkAgg(fig, master=frame_plot)`: Embeds the Matplotlib figure into the Tkinter frame.
- `canvas.draw()`: Draws the plot.
- `canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)`: Packs the plot into the frame.

Finalizing the Application

```
root.mainloop()
```

- `root.mainloop()`: Starts the Tkinter event loop, making the application responsive to user interactions.

This code provides a full-featured Tkinter application for analyzing synthetic weather data, including data input, display, and visualization functionalities.

Weather Data Analysis

Raw Data Aggregated Data Input Data Plot Data

| City | Date | Temperature | Humidity | Condition |
|---------|---------------------|-------------|----------|-----------|
| City_44 | 2023-01-01 00:00:00 | -3.3 | 67.7 | Rainy |
| City_47 | 2023-01-01 01:00:00 | -1.5 | 69.3 | Snowy |
| City_0 | 2023-01-01 02:00:00 | 2.4 | 71.7 | Windy |
| City_3 | 2023-01-01 03:00:00 | 34.7 | 93.7 | Cloudy |
| City_3 | 2023-01-01 04:00:00 | -8.1 | 94.8 | Rainy |
| City_39 | 2023-01-01 05:00:00 | 2.1 | 84.3 | Cloudy |
| City_9 | 2023-01-01 06:00:00 | 9.0 | 33.5 | Sunny |
| City_19 | 2023-01-01 07:00:00 | 32.0 | 56.3 | Sunny |
| City_21 | 2023-01-01 08:00:00 | 2.2 | 43.4 | Windy |
| City_36 | 2023-01-01 09:00:00 | 5.6 | 30.9 | Cloudy |
| City_23 | 2023-01-01 10:00:00 | 12.9 | 38.1 | Cloudy |
| City_6 | 2023-01-01 11:00:00 | -7.0 | 61.9 | Windy |
| City_24 | 2023-01-01 12:00:00 | -5.9 | 84.6 | Windy |
| City_24 | 2023-01-01 13:00:00 | -1.6 | 88.8 | Windy |
| City_12 | 2023-01-01 14:00:00 | 1.8 | 79.6 | Windy |
| City_1 | 2023-01-01 15:00:00 | 13.3 | 36.6 | Rainy |
| City_38 | 2023-01-01 16:00:00 | 14.1 | 55.7 | Snowy |
| City_39 | 2023-01-01 17:00:00 | 24.3 | 67.0 | Cloudy |
| City_23 | 2023-01-01 18:00:00 | 10.3 | 22.5 | Sunny |
| City_46 | 2023-01-01 19:00:00 | 21.7 | 99.7 | Cloudy |
| City_24 | 2023-01-01 20:00:00 | 21.8 | 37.8 | Windy |
| City_17 | 2023-01-01 21:00:00 | 33.3 | 50.5 | Sunny |
| City_37 | 2023-01-01 22:00:00 | 4.9 | 65.2 | Cloudy |
| City_25 | 2023-01-01 23:00:00 | 6.2 | 25.1 | Cloudy |
| City_13 | 2023-01-02 00:00:00 | 21.3 | 32.2 | Cloudy |
| City_8 | 2023-01-02 01:00:00 | 6.8 | 69.4 | Rainy |
| City_9 | 2023-01-02 02:00:00 | 27.3 | 35.7 | Windy |
| City_20 | 2023-01-02 03:00:00 | -6.0 | 24.5 | Windy |
| City_16 | 2023-01-02 04:00:00 | 21.2 | 59.0 | Cloudy |
| City_5 | 2023-01-02 05:00:00 | -3.0 | 92.8 | Windy |

Weather Data Analysis

Raw Data Aggregated Data Input Data Plot Data

| City | Condition | Temperature_mean | Temperature_max | Temperature_min | Humidity_mean | Humidity_max | Humidity_min |
|---------|-----------|----------------------|-----------------|-----------------|--------------------|--------------|--------------|
| City_0 | Cloudy | 14.182978723404256 | 33.4 | -9.7 | 69.3787234042533 | 99.7 | 20.1 |
| City_0 | Rainy | 14.640476190476189 | 34.5 | -6.4 | 59.471428571428575 | 99.3 | 25.1 |
| City_0 | Snowy | 11.810526315789474 | 34.0 | -8.9 | 62.828947368421055 | 97.5 | 23.7 |
| City_0 | Sunny | 10.502777777777778 | 31.8 | -9.3 | 57.486111111111114 | 92.0 | 21.5 |
| City_0 | Windy | 12.801923076923078 | 34.7 | -9.1 | 55.96153846153846 | 99.9 | 20.7 |
| City_1 | Cloudy | 14.172727272727272 | 33.8 | -6.8 | 63.327272727272735 | 99.4 | 20.6 |
| City_1 | Rainy | 11.316279069767443 | 34.6 | -6.9 | 58.86279069767442 | 99.7 | 20.3 |
| City_1 | Snowy | 10.669565217391305 | 33.9 | -8.6 | 55.3260869565217 | 99.6 | 20.2 |
| City_1 | Sunny | 15.75 | 32.7 | -8.2 | 57.661764705882355 | 96.9 | 20.9 |
| City_1 | Windy | 14.48235294117647 | 33.6 | -7.0 | 61.0 | 98.3 | 24.4 |
| City_10 | Cloudy | 12.79736842105641027 | 29.8 | -7.6 | 55.93684210528315 | 98.7 | 21.6 |
| City_10 | Rainy | 11.148780467804878 | 32.4 | -9.6 | 57.92926829266293 | 100.0 | 20.3 |
| City_10 | Snowy | 12.933333333333334 | 32.6 | -7.5 | 57.919444444444444 | 99.2 | 22.0 |
| City_10 | Sunny | 12.29 | 32.4 | -7.8 | 53.19333333333335 | 89.1 | 23.1 |
| City_10 | Windy | 15.02857142857143 | 33.5 | -9.6 | 64.10238095238095 | 95.4 | 26.2 |
| City_11 | Cloudy | 12.504000000000001 | 34.3 | -9.3 | 64.568 | 99.9 | 20.9 |
| City_11 | Rainy | 9.725641025641027 | 33.4 | -9.7 | 59.535846153846156 | 92.3 | 22.8 |
| City_11 | Snowy | 14.580000000000002 | 33.4 | -7.9 | 65.38 | 97.9 | 26.1 |
| City_11 | Sunny | 13.011627906976743 | 34.2 | -9.5 | 64.48139534883721 | 98.4 | 21.2 |
| City_11 | Windy | 14.248888888888889 | 34.8 | -8.8 | 55.21333333333333 | 97.8 | 21.4 |
| City_12 | Cloudy | 13.545454545454545 | 32.1 | -9.9 | 60.42424242424242 | 100.0 | 21.1 |
| City_12 | Rainy | 13.46904761904762 | 34.2 | -9.4 | 50.53809523809524 | 93.8 | 20.1 |
| City_12 | Snowy | 12.973170731707317 | 33.7 | -7.5 | 61.7463414634147 | 99.8 | 25.9 |
| City_12 | Sunny | 10.129787234042555 | 34.8 | -9.5 | 58.55744680851063 | 96.6 | 20.8 |
| City_12 | Windy | 10.692105263157895 | 32.9 | -9.9 | 59.231578947368426 | 96.8 | 21.7 |
| City_13 | Cloudy | 8.697222222222223 | 29.2 | -9.8 | 56.44166666666667 | 96.2 | 20.2 |
| City_13 | Rainy | 9.210526315789474 | 35.0 | -9.8 | 55.123684210526314 | 98.6 | 20.1 |
| City_13 | Snowy | 10.783333333333333 | 31.8 | -9.3 | 54.69166666666667 | 99.8 | 20.5 |
| City_13 | Sunny | 11.623333333333333 | 34.7 | -9.4 | 60.91818181818182 | 98.8 | 22.6 |

Weather Data Analysis

Raw Data Aggregated Data Input Data Plot Data

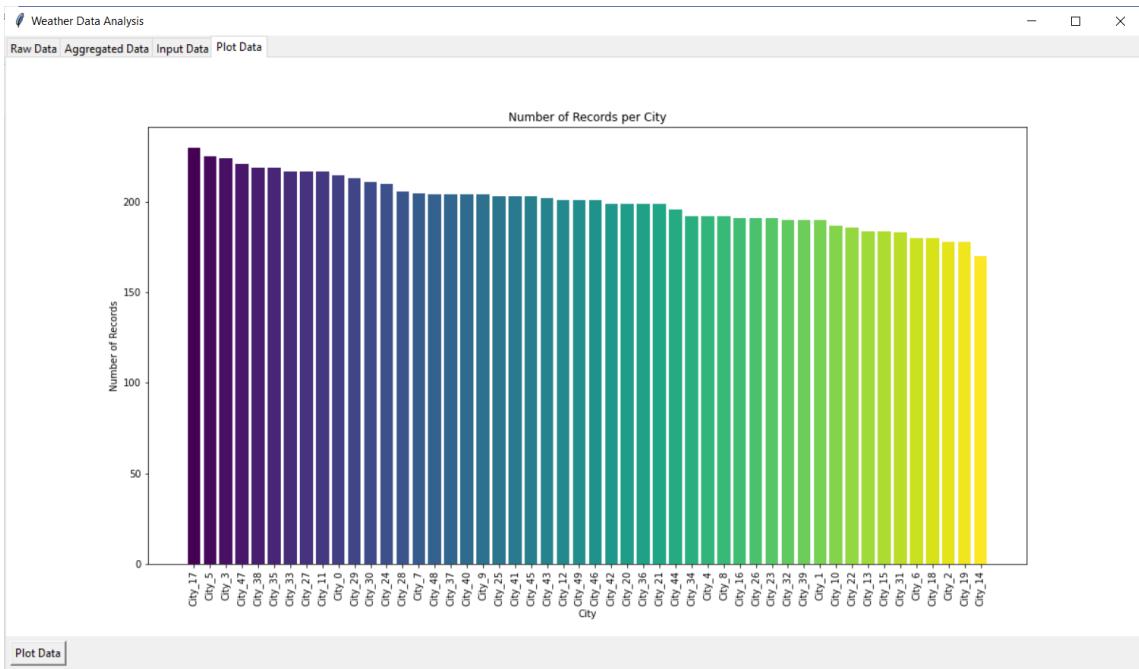
City:

Date (YYYY-MM-DD HH:MM:SS):

Temperature:

Humidity:

Condition:



**DATAFRAME
PIVOTING AND RERSHAPING
DATAFRAME
PIVOTING AND RERSHAPING**

PIVOTING

Pivoting a DataFrame in pandas refers to reshaping the data in such a way that one or more columns are transformed into new columns or indexes. This operation is useful when you need to rearrange data to make it easier to analyze, especially when dealing with data that's organized in a "long" format and you want to switch it to a "wide" format or vice versa.

Let's break down the key concepts and steps involved in DataFrame pivoting:

1. Understanding the Data Format

Long Format: This format is often seen in datasets where each row represents a single observation, and multiple rows may share the same category but have different values for a particular variable. For example:

| Country | Year | Value |
|---------|------|-------|
| USA | 2020 | 500 |
| USA | 2021 | 600 |
| Canada | 2020 | 300 |
| Canada | 2021 | 350 |

Wide Format: This format organizes the data so that each unique value in a category (e.g., Year) becomes a separate column, and each row represents a different category (e.g., Country). For example:

| Country | 2020 | 2021 |
|---------|------|------|
|---------|------|------|

| | | |
|--------|-----|-----|
| USA | 500 | 600 |
| Canada | 300 | 350 |

2. Pivoting with pivot()

The pivot() function in pandas is used to reshape data. It turns unique values from one column into columns, with other columns becoming the index or values.

Syntax:

```
DataFrame.pivot(index=None,           columns=None,  
values=None)
```

- index: Column(s) to set as the index of the resulting DataFrame.
- columns: Column(s) whose unique values will become the columns of the resulting DataFrame.
- values: Column(s) to fill in the new DataFrame with values.

Example:

Given the DataFrame df:

```
df = pd.DataFrame({  
    'Country': ['USA', 'USA', 'Canada', 'Canada'],  
    'Year': [2020, 2021, 2020, 2021],  
    'Value': [500, 600, 300, 350]  
})
```

Pivot the data to make Year the columns and Value the data:

```
pivoted_df = df.pivot(index='Country',  
columns='Year', values='Value')
```

This would result in:

| Year | 2020 | 2021 |
|---------|------|------|
| Country | | |
| Canada | 300 | 350 |
| USA | 500 | 600 |

3. Handling Duplicates with pivot_table()

The pivot() function works well when you have unique index/column combinations. However, if there are duplicates (e.g., multiple rows with the same index and column), pivot() will raise an error.

To handle duplicates, use the pivot_table() function, which allows you to aggregate the data using a function (like mean, sum, etc.).

Syntax:

```
DataFrame.pivot_table(index=None,           columns=None,  
values=None, aggfunc='mean', ...)
```

- aggfunc: The aggregation function to apply (e.g., mean, sum, etc.).

Example:

```
df = pd.DataFrame({  
    'Country': ['USA', 'USA', 'Canada', 'Canada',  
    'Canada'],  
    'Year': [2020, 2020, 2020, 2021, 2021],  
    'Value': [500, 520, 300, 350, 360]  
})  
pivot_table_df = df.pivot_table(index='Country',  
columns='Year', values='Value', aggfunc='sum')
```

This would result in:

| Year | 2020 | 2021 |
|---------|------|------|
| Country | | |
| Canada | 300 | 710 |
| USA | 1020 | NaN |

4. Resetting the Index

After pivoting, you may want to reset the index to turn the index back into a column:

```
pivoted_df.reset_index(inplace=True)
```

This will turn the Country index back into a column.

5. Unpivoting with melt()

The opposite of pivoting is unpivoting, which turns columns into rows. This can be done using the melt() function.

Example:

```
df_unpivoted = pivoted_df.melt(id_vars='Country',  
var_name='Year', value_name='Value')
```

This would convert the wide format back into a long format:

| Country | Year | Value |
|---------|------|-------|
| USA | 2020 | 500 |
| USA | 2021 | 600 |
| Canada | 2020 | 300 |
| Canada | 2021 | 350 |

6. Practical Use Cases

- Time Series Data: Pivoting is commonly used to rearrange time series data for easier comparison across different time periods.
- Survey Data: Pivoting can help transform survey responses to show how different groups responded to various questions.
- Sales Data: Pivoting is often used to compare sales figures across different regions or products over time.

Understanding how to pivot and unpivot DataFrames effectively is key to exploring and analyzing data in various formats.

RESHAPING

Reshaping a DataFrame in pandas involves changing its structure or layout, often to make the data easier to analyze or visualize. The primary operations for reshaping a DataFrame include pivoting, stacking, unstacking, melting, and concatenating. Each of these operations serves different purposes, depending on how you want to organize the data.

1. Pivoting

Pivoting involves converting rows into columns or vice versa. It's useful when you want to transform a dataset from a "long" format (where each row is an observation) to a "wide" format (where each row is a category and columns represent different variables).

Key Functions:

- pivot()
- pivot_table()

Example:

```
import pandas as pd

data = {
    'Date': ['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-02'],
    'City': ['New York', 'Los Angeles', 'New York', 'Los Angeles'],
    'Temperature': [32, 75, 28, 80]
}
df = pd.DataFrame(data)

pivoted_df = df.pivot(index='Date',
columns='City', values='Temperature')
```

This will reshape the DataFrame so that cities become columns, and temperatures are their values.

2. Melting

Melting is the opposite of pivoting; it converts a wide format DataFrame into a long format by turning columns into rows. This is useful when you have multiple columns that represent different categories or variables, and you want to analyze them as part of a single category.

Key Function:

- melt()

Example:

```
melted_df =  
pivoted_df.reset_index().melt(id_vars='Date',  
var_name='City', value_name='Temperature')
```

This will convert the DataFrame back into its original long format.

3. Stacking

Stacking is a more advanced reshaping operation that pivots the columns of a DataFrame into rows, resulting in a hierarchical index (a MultiIndex). This operation is particularly useful when dealing with multi-dimensional data.

Key Function:

- stack()

Example:

```
stacked_df = pivoted_df.stack()
```

This will create a Series with a MultiIndex, where each level represents different dimensions of the data.

MultiIndex Example:

```
data = {  
    'City': ['New York', 'New York', 'Los Angeles', 'Los Angeles'],  
    'Year': [2023, 2023, 2023, 2023],  
    'Month': ['Jan', 'Feb', 'Jan', 'Feb'],  
    'Temperature': [30, 25, 75, 70]  
}  
df = pd.DataFrame(data)  
  
stacked_df = df.set_index(['City', 'Year', 'Month']).stack()
```

4. Unstacking

Unstacking is the inverse of stacking. It pivots the innermost level of a MultiIndex into columns. This operation is useful when you want to spread out hierarchical data into a wider format.

Key Function:

- unstack()

Example:

```
unstacked_df = stacked_df.unstack()
```

This will reverse the stacking operation, converting rows back into columns.

5. Concatenating

Concatenation is the process of joining multiple DataFrames along a particular axis (either rows or columns). This operation is useful when you need to combine datasets that share a similar structure but contain different data points.

Key Function:

- concat()

Example:

```
df1 = pd.DataFrame({  
    'A': ['A0', 'A1', 'A2'],  
    'B': ['B0', 'B1', 'B2']  
})  
  
df2 = pd.DataFrame({  
    'A': ['A3', 'A4', 'A5'],  
    'B': ['B3', 'B4', 'B5']  
})  
  
concatenated_df = pd.concat([df1, df2], axis=0)
```

This will combine df1 and df2 vertically, appending the rows of df2 to the rows of df1.

6. Merging and Joining

Merging and joining are operations used to combine DataFrames based on a key or set of keys. They are similar to SQL joins and are

used when you want to combine different datasets that have some common columns.

Key Functions:

- merge()
- join()

Example:

```
left = pd.DataFrame({  
    'key': ['A', 'B', 'C'],  
    'value': [1, 2, 3]  
})  
  
right = pd.DataFrame({  
    'key': ['A', 'B', 'D'],  
    'value': [4, 5, 6]  
})  
  
merged_df = pd.merge(left, right, on='key',  
how='inner')
```

This will merge the two DataFrames on the key column, keeping only the rows with matching keys.

7. Transposing

Transposing involves flipping the rows and columns of a DataFrame. This is useful when you want to reorient the data for better readability or analysis.

Key Function:

- transpose() or .T

Example:

```
transposed_df = df.T
```

This operation will swap the rows and columns, effectively flipping the DataFrame.

8. Changing the Index with `set_index()` and `reset_index()`

Setting an index involves designating one or more columns to be used as the index (row labels), which can change the structure of the DataFrame. Resetting the index can turn the index back into a regular column.

Key Functions:

- `set_index()`
- `reset_index()`

Example:

```
indexed_df = df.set_index('City')
reset_df = indexed_df.reset_index()
```

This operation will set City as the index and later revert it back to a regular column.

Practical Use Cases:

- Data Cleaning: Reshaping is essential when tidying up messy datasets, transforming them into a more structured format.
- Time Series Analysis: Reshaping helps in analyzing data over time by pivoting or unstacking time-related columns.

- Multi-Dimensional Analysis: Stacking and unstacking allow you to analyze complex data with multiple dimensions.

Reshaping a DataFrame is a powerful tool that allows you to manipulate the structure of your data to suit your analysis needs. Each method—pivoting, melting, stacking, unstacking, concatenating, merging, and transposing—offers different ways to organize and view your data. Understanding these operations is key to effective data manipulation and analysis in pandas.

EXAMPLE 5.1

Pivoting Dataframe with Synthetic Gold Dataset

The purpose of this project is to generate and analyze a synthetic dataset representing gold production data across various countries, years, and regions. The dataset is designed to be large and complex, providing a realistic simulation of real-world data used for analysis and decision-making. By creating a DataFrame with multiple attributes such as country, year, region, and gold production quantities, the project aims to simulate scenarios where detailed data analysis is required.

The core of the project involves pivoting this dataset to summarize and aggregate the data effectively. Using the pivot_table method, the project transforms the raw data into a more insightful format where gold production metrics are aggregated by country and region across different years. This transformation helps in identifying trends and patterns in gold production, such as which countries and regions are leading in gold output and how production varies over time.

Finally, the project ensures that the results are accessible and shareable by saving both the original and pivoted datasets to Excel files. This makes it easier for stakeholders to review and interact with the data. The saved files serve as a valuable resource for further analysis, visualization, or reporting, providing a foundation for more in-depth exploration of gold production trends and contributing to informed decision-making in fields related to mining and resource management.

```
import pandas as pd
import numpy as np

# Seed for reproducibility
np.random.seed(42)

# Generate a more complex, larger synthetic
dataset
large_data = {
    'Country': np.random.choice(['USA', 'Canada',
'Australia', 'South Africa', 'Russia'],
size=10000),
    'Year': np.random.choice(range(2000, 2025),
size=10000),
    'Region': np.random.choice(['North', 'South',
'East', 'West'], size=10000),
    'Mine': np.random.choice(['Mine_A', 'Mine_B',
'Mine_C', 'Mine_D'], size=10000),
    'Gold_Production_Tonnes':
np.random.randint(50, 500, size=10000)
}

large_df = pd.DataFrame(large_data)

# Pivot the DataFrame using multiple columns to
ensure more unique combinations
```

```
large_pivoted_df = large_df.pivot_table(index=['Country', 'Region'], columns='Year',
values='Gold_Production_Tonnes', aggfunc='sum')

# Save the original and pivoted DataFrame to Excel
large_df.to_excel('large_synthetic_gold_data_original_complex.xlsx', sheet_name='Original_Data',
index=False)
large_pivoted_df.to_excel('large_synthetic_gold_data_pivoted_complex.xlsx',
sheet_name='Pivoted_Data')

print("Excel files saved as
'large_synthetic_gold_data_original_complex.xlsx'
and
'large_synthetic_gold_data_pivoted_complex.xlsx'.")
```

Here's a detailed breakdown of each part of the code:

1. Import Libraries

```
import pandas as pd
import numpy as np
```

- `pandas as pd`: Imports the pandas library and gives it the alias `pd`. Pandas is used for data manipulation and analysis, providing data structures like `DataFrames`.
- `numpy as np`: Imports the numpy library and gives it the alias `np`. Numpy is used for numerical operations and working with arrays.

2. Seed for Reproducibility

```
np.random.seed(42)
```

- np.random.seed(42): Sets the seed for the random number generator used by numpy. This ensures that the random numbers generated are the same each time the code is run, which is useful for reproducibility.

3. Generate a More Complex, Larger Synthetic Dataset

```
large_data = {
    'Country': np.random.choice(['USA', 'Canada',
'Australia', 'South Africa', 'Russia'],
size=10000),
    'Year': np.random.choice(range(2000, 2025),
size=10000),
    'Region': np.random.choice(['North', 'South',
'East', 'West'], size=10000),
    'Mine': np.random.choice(['Mine_A', 'Mine_B',
'Mine_C', 'Mine_D'], size=10000),
    'Gold_Production_Tonnes':
np.random.randint(50, 500, size=10000)
}
```

- np.random.choice(..., size=10000): Randomly selects 10,000 entries from the provided list. This is done for each column ('Country', 'Year', 'Region', 'Mine'), creating a large synthetic dataset with these values.
- np.random.randint(50, 500, size=10000): Generates 10,000 random integers between 50 and 500 for the 'Gold_Production_Tonnes' column.

```
large_df = pd.DataFrame(large_data)
```

- pd.DataFrame(large_data): Converts the large_data dictionary into a DataFrame, a tabular data structure in pandas.

4. Pivot the DataFrame

```
large_pivoted_df      =    large_df.pivot_table(index=
['Country',           'Region'],          columns='Year',
values='Gold_Production_Tonnes', aggfunc='sum')
```

- large_df.pivot_table(...): Creates a pivot table from the DataFrame. Here's how the parameters work:
- index=['Country', 'Region']: Defines the rows of the pivot table. Each row will be a combination of 'Country' and 'Region'.
- columns='Year': Defines the columns of the pivot table. Each year will become a column.
- values='Gold_Production_Tonnes': Specifies the data to aggregate in the table.
- aggfunc='sum': Aggregates the data by summing up the 'Gold_Production_Tonnes' for each combination of 'Country', 'Region', and 'Year'.

5. Save the Original and Pivoted DataFrame to Excel

```
large_df.to_excel('large_synthetic_gold_data_original_complex.xlsx', sheet_name='Original_Data',
index=False)
```

large_df.to_excel(...): Saves the original DataFrame to an Excel file.

- 'large_synthetic_gold_data_original_complex.xlsx': The filename for the Excel file.
- sheet_name='Original_Data': The name of the sheet in the Excel file.
- index=False: Excludes the DataFrame index from the Excel file.

```
large_pivoted_df.to_excel('large_synthetic_gold_data_pivoted_complex.xlsx',
sheet_name='Pivoted_Data')
```

- large_pivoted_df.to_excel(...): Saves the pivoted DataFrame to a different Excel file.
- 'large_synthetic_gold_data_pivoted_complex.xlsx': The filename for the Excel file.
- sheet_name='Pivoted_Data': The name of the sheet in the Excel file.

6. Print Confirmation

```
print("Excel files saved as
'large_synthetic_gold_data_original_complex.xlsx'
and
'large_synthetic_gold_data_pivoted_complex.xlsx'.")
```

- print(...): Outputs a message confirming that the Excel files have been saved successfully.

EXAMPLE 5.2

GUI Tkinter for Pivoting Dataframe with Synthetic Gold Dataset

This code is designed to create a rich and interactive graphical user interface (GUI) using Python's Tkinter library. The primary purpose is to visualize and interact with a large synthetic dataset related to gold production, which includes data on countries, regions, mines, and yearly production. By generating this dataset with pandas and numpy, the code simulates real-world data handling scenarios, making it useful for exploring and analyzing gold production trends.

The GUI is organized into multiple tabs, each serving a distinct purpose. These tabs include views for the original dataset, a pivoted version of the data, and various summary statistics by country, region, and year. Additionally, there are graphical visualizations in the form of bar charts, which provide a clear and intuitive way to understand the distribution of gold production across different dimensions. The use of matplotlib allows these charts to be embedded directly within the Tkinter interface, making the application not only functional but also visually appealing.

Overall, the code exemplifies how to build a comprehensive data analysis tool in Python, integrating data processing, filtering, and visualization within a user-friendly interface. It demonstrates the power of combining Tkinter with other libraries like pandas and matplotlib to create a cohesive and interactive experience for exploring complex datasets. This makes it particularly valuable for educational purposes, data analysis tasks, or any scenario where users need to interact with and analyze large volumes of data effectively.

```
import tkinter as tk
from tkinter import ttk
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Seed for reproducibility
np.random.seed(42)

# Generate a more complex, larger synthetic
dataset
large_data = {
    'Country': np.random.choice(['USA', 'Canada',
'Australia', 'South Africa', 'Russia'],
size=10000),
    'Year': np.random.choice(range(2000, 2025),
size=10000),
    'Region': np.random.choice(['North', 'South',
'East', 'West'], size=10000),
    'Mine': np.random.choice(['Mine_A', 'Mine_B',
'Mine_C', 'Mine_D'], size=10000),
    'Gold_Production_Tonnes':
np.random.randint(50, 500, size=10000)
}

large_df = pd.DataFrame(large_data)
large_pivoted_df = large_df.pivot_table(index=
['Country', 'Region'], columns='Year',
values='Gold_Production_Tonnes', aggfunc='sum')

# Summary DataFrames
country_summary_df = large_df.groupby('Country')
['Gold_Production_Tonnes'].sum().reset_index()
region_summary_df = large_df.groupby('Region')
['Gold_Production_Tonnes'].sum().reset_index()
year_summary_df = large_df.groupby('Year')
['Gold_Production_Tonnes'].sum().reset_index()
```

```
class RichTkinterApp:  
    def __init__(self, root):  
        self.root = root  
        self.root.title("Rich Tkinter GUI for Gold  
Production Data")  
        self.root.geometry("1200x800")  
  
        # Notebook (Tabs)  
        self.notebook = ttk.Notebook(root)  
        self.notebook.pack(expand=True,  
fill='both')  
  
        # Create Tabs  
        self.create_original_data_tab()  
        self.create_pivoted_data_tab()  
        self.create_country_summary_tab()  
        self.create_region_summary_tab()  
        self.create_year_summary_tab()  
        self.create_country_distribution_tab()  
        self.create_region_distribution_tab()  
        self.create_year_distribution_tab()  
  
    def create_original_data_tab(self):  
        frame = ttk.Frame(self.notebook)  
        self.notebook.add(frame, text='Original  
Data')  
  
        # Country Filter  
        ttk.Label(frame, text="Filter by  
Country:").pack(pady=10)  
        self.country_combobox =  
        ttk.Combobox(frame, values=['All'] +  
list(large_df['Country'].unique()))  
        self.country_combobox.set('All')  
        self.country_combobox.pack(pady=5)
```

```
        self.country_combobox.bind("
<<ComboboxSelected>>",
self.update_original_data_table)

        # Table
        self.original_table = ttk.Treeview(frame,
columns=list(large_df.columns), show='headings')
        self.original_table.pack(expand=True,
fill='both')

        # Scrollbars
        vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.original_table.yview)
        vsb.pack(side='right', fill='y')

        self.original_table.configure(yscrollcommand=vsb.
set)

        hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.original_table.xview)
        hsb.pack(side='bottom', fill='x')

        self.original_table.configure(xscrollcommand=hsb.
set)

        # Define columns
        for col in large_df.columns:
            self.original_table.heading(col,
text=col)
            self.original_table.column(col,
width=100)

            self.update_original_data_table()  #
Initial load
```

```
def create_pivoted_data_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Pivoted
Data')

    # Table
    self.pivoted_table = ttk.Treeview(frame,
columns=list(large_pivoted_df.columns),
show='headings')
    self.pivoted_table.pack(expand=True,
fill='both')

    # Scrollbars
    vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.pivoted_table.yview)
    vsb.pack(side='right', fill='y')

    self.pivoted_table.configure(yscrollcommand=vsb.s
et)

    hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.pivoted_table.xview)
    hsb.pack(side='bottom', fill='x')

    self.pivoted_table.configure(xscrollcommand=hsb.s
et)

    # Define columns
    for col in large_pivoted_df.columns:
        self.pivoted_table.heading(col,
text=col)
        self.pivoted_table.column(col,
width=100)
```

```
    self.update_pivoted_data_table()  #
Initial load

def create_country_summary_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Country
Summary')

    # Table
    self.country_summary_table =
ttk.Treeview(frame,
columns=list(country_summary_df.columns),
show='headings')

    self.country_summary_table.pack(expand=True,
fill='both')

    # Scrollbars
    vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.country_summary_table.yview)
    vsb.pack(side='right', fill='y')

    self.country_summary_table.configure(yscrollcomm
nd=vsb.set)

    hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.country_summary_table.xview)
    hsb.pack(side='bottom', fill='x')

    self.country_summary_table.configure(xscrollcomm
nd=hsb.set)

    # Define columns
```

```
    for col in country_summary_df.columns:

        self.country_summary_table.heading(col, text=col)
        self.country_summary_table.column(col,
width=100)

            self.update_country_summary_table() # Initial load

    def create_region_summary_tab(self):
        frame = ttk.Frame(self.notebook)
        self.notebook.add(frame, text='Region
Summary')

        # Table
        self.region_summary_table =
ttk.Treeview(frame,
columns=list(region_summary_df.columns),
show='headings')

        self.region_summary_table.pack(expand=True,
fill='both')

        # Scrollbars
        vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.region_summary_table.yview)
        vsb.pack(side='right', fill='y')

        self.region_summary_table.configure(yscrollcommand=
vsb.set)

        hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.region_summary_table.xview)
        hsb.pack(side='bottom', fill='x')
```

```
    self.region_summary_table.configure(xscrollcommand=hsb.set)

        # Define columns
        for col in region_summary_df.columns:
            self.region_summary_table.heading(col, text=col)
            self.region_summary_table.column(col, width=100)

        self.update_region_summary_table() # Initial load

    def create_year_summary_tab(self):
        frame = ttk.Frame(self.notebook)
        self.notebook.add(frame, text='Year Summary')

        # Table
        self.year_summary_table = ttk.Treeview(frame,
columns=list(year_summary_df.columns),
show='headings')
        self.year_summary_table.pack(expand=True, fill='both')

        # Scrollbars
        vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.year_summary_table.yview)
        vsb.pack(side='right', fill='y')

        self.year_summary_table.configure(yscrollcommand=vsb.set)
```

```
        hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.year_summary_table.xview)
        hsb.pack(side='bottom', fill='x')

    self.year_summary_table.configure(xscrollcommand=
hsb.set)

    # Define columns
    for col in year_summary_df.columns:
        self.year_summary_table.heading(col,
text=col)
        self.year_summary_table.column(col,
width=100)

    self.update_year_summary_table() #  
Initial load

def create_country_distribution_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Country  
Distribution')

    # Create figure for plotting
    fig, ax = plt.subplots(figsize=(10, 6))
    ax.bar(country_summary_df['Country'],
country_summary_df['Gold_Production_Tonnes'],
color='skyblue')
    ax.set_title('Total Gold Production by  
Country')
    ax.set_xlabel('Country')
    ax.set_ylabel('Total Gold Production  
(Tonnes)')

    ax.set_xticklabels(country_summary_df['Country'],
rotation=45, ha='right')
```

```
# Display figure in Tkinter
    canvas = FigureCanvasTkAgg(fig,
master=frame)
        canvas.draw()
        canvas.get_tk_widget().pack(expand=True,
fill='both')

    def create_region_distribution_tab(self):
        frame = ttk.Frame(self.notebook)
        self.notebook.add(frame, text='Region
Distribution')

        # Create figure for plotting
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.bar(region_summary_df['Region'],
region_summary_df['Gold_Production_Tonnes'],
color='lightgreen')
        ax.set_title('Total Gold Production by
Region')
        ax.set_xlabel('Region')
        ax.set_ylabel('Total Gold Production
(Tonnes)')

        # Display figure in Tkinter
        canvas = FigureCanvasTkAgg(fig,
master=frame)
        canvas.draw()
        canvas.get_tk_widget().pack(expand=True,
fill='both')

    def create_year_distribution_tab(self):
        frame = ttk.Frame(self.notebook)
        self.notebook.add(frame, text='Year
Distribution')
```

```
# Create figure for plotting
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(year_summary_df['Year'],
year_summary_df['Gold_Production_Tonnes'],
marker='o', color='coral')
    ax.set_title('Total Gold Production by
Year')
    ax.set_xlabel('Year')
    ax.set_ylabel('Total Gold Production
(Tonnes)')
    ax.grid(True)

# Display figure in Tkinter
canvas = FigureCanvasTkAgg(fig,
master=frame)
    canvas.draw()
    canvas.get_tk_widget().pack(expand=True,
fill='both')

    def update_original_data_table(self,
event=None):
        # Clear current data
        for row in
self.original_table.get_children():
            self.original_table.delete(row)

        # Get selected country
        selected_country =
self.country_combobox.get()

        # Filter data
        if selected_country == 'All':
            filtered_df = large_df
        else:
```

```
        filtered_df =
large_df[large_df['Country'] == selected_country]

        # Insert new data
        for _, row in filtered_df.iterrows():
            self.original_table.insert('', 'end',
values=list(row))

    def update_pivoted_data_table(self):
        # Clear current data
        for row in
self.pivoted_table.get_children():
            self.pivoted_table.delete(row)

        # Insert new data
        for _, row in large_pivoted_df.iterrows():
            self.pivoted_table.insert('', 'end',
values=list(row))

    def update_country_summary_table(self):
        # Clear current data
        for row in
self.country_summary_table.get_children():
            self.country_summary_table.delete(row)

        # Insert new data
        for _, row in
country_summary_df.iterrows():
            self.country_summary_table.insert('', 'end',
values=list(row))

    def update_region_summary_table(self):
        # Clear current data
        for row in
self.region_summary_table.get_children():
            self.region_summary_table.delete(row)
```

```

        # Insert new data
        for _, row in
region_summary_df.iterrows():
            self.region_summary_table.insert('', 'end', values=list(row))

    def update_year_summary_table(self):
        # Clear current data
        for row in
self.year_summary_table.get_children():
            self.year_summary_table.delete(row)

        # Insert new data
        for _, row in year_summary_df.iterrows():
            self.year_summary_table.insert('', 'end', values=list(row))

# Main loop
if __name__ == "__main__":
    root = tk.Tk()
    app = RichTkinterApp(root)
    root.mainloop()

```

This code creates a rich graphical user interface (GUI) using the Tkinter library to visualize and interact with a synthetic dataset about gold production. The dataset is generated with pandas and numpy, and the visualization components, including graphs and tables, are integrated using matplotlib. Here's a detailed explanation of each part of the code:

Importing Required Libraries

```

import tkinter as tk
from tkinter import ttk

```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from    matplotlib.backends.backend_tkagg      import
FigureCanvasTkAgg
```

- tkinter: The standard Python library for creating GUI applications.
- ttk: A submodule of tkinter that provides themed widgets, making the GUI more visually appealing.
- pandas: A powerful library for data manipulation and analysis.
- numpy: A library for numerical operations, especially useful for generating synthetic data.
- matplotlib.pyplot: A plotting library for creating static, animated, and interactive visualizations in Python.
- FigureCanvasTkAgg: A module from matplotlib that allows embedding plots in Tkinter applications.

Generating Synthetic Dataset

```
# Seed for reproducibility
np.random.seed(42)

# Generate a more complex, larger synthetic
dataset
large_data = {
    'Country': np.random.choice(['USA', 'Canada',
'Australia', 'South Africa', 'Russia'],
size=10000),
    'Year': np.random.choice(range(2000, 2025),
size=10000),
```

```
'Region': np.random.choice(['North', 'South',
'East', 'West'], size=10000),
'Mine': np.random.choice(['Mine_A', 'Mine_B',
'Mine_C', 'Mine_D'], size=10000),
'Gold_Production_Tonnes':
np.random.randint(50, 500, size=10000)
}
```

- Seed for reproducibility: Sets a random seed to ensure the synthetic data is the same every time the code runs.
- large_data: A dictionary containing arrays of random data generated for different fields (Country, Year, Region, Mine, Gold_Production_Tonnes). Each field contains 10,000 entries to simulate a large dataset.

Creating DataFrames

```
large_df = pd.DataFrame(large_data)
large_pivoted_df = large_df.pivot_table(index=
['Country', 'Region'], columns='Year',
values='Gold_Production_Tonnes', aggfunc='sum')
```

- large_df: Converts the synthetic data dictionary into a pandas DataFrame.
- large_pivoted_df: Creates a pivot table from large_df, aggregating the gold production data by Country and Region for each Year.

Creating Summary DataFrames

```
# Summary DataFrames
country_summary_df = large_df.groupby('Country')
['Gold_Production_Tonnes'].sum().reset_index()
```

```
region_summary_df = large_df.groupby('Region')  
['Gold_Production_Tonnes'].sum().reset_index()  
year_summary_df = large_df.groupby('Year')  
['Gold_Production_Tonnes'].sum().reset_index()
```

- country_summary_df: Summarizes total gold production by Country.
- region_summary_df: Summarizes total gold production by Region.
- year_summary_df: Summarizes total gold production by Year.

Defining the RichTkinterApp Class

The RichTkinterApp class is responsible for building and managing the GUI.

```
Initialization (__init__ Method)  
python  
Copy code  
class RichTkinterApp:  
    def __init__(self, root):  
        self.root = root  
        self.root.title("Rich Tkinter GUI for Gold  
Production Data")  
        self.root.geometry("1200x800")  
  
        # Notebook (Tabs)  
        self.notebook = ttk.Notebook(root)  
        self.notebook.pack(expand=True,  
fill='both')  
  
        # Create Tabs  
        self.create_original_data_tab()  
        self.create_pivoted_data_tab()
```

```
    self.create_country_summary_tab()
    self.create_region_summary_tab()
    self.create_year_summary_tab()
    self.create_country_distribution_tab()
    self.create_region_distribution_tab()
    self.create_year_distribution_tab()
```

- root: The root window of the Tkinter application.
- root.title(): Sets the title of the window.
- root.geometry(): Sets the size of the window.
- ttk.Notebook: Creates a tabbed interface (Notebook) where each tab contains different data views.
- self.notebook.pack(): Ensures the notebook expands to fill the available space.
- Tab creation methods: Calls methods to create various tabs in the GUI.

Creating Tabs

Each of the following methods defines the layout and functionality of a specific tab in the notebook.

1. *Original Data Tab*

```
def create_original_data_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Original Data')

    # Country Filter
    ttk.Label(frame,      text="Filter      by
Country:").pack(pady=10)
    self.country_combobox = ttk.Combobox(frame,
values=['All'] + +
list(large_df['Country'].unique()))
```

```
    self.country_combobox.set('All')
    self.country_combobox.pack(pady=5)
                self.country_combobox.bind("
<<ComboboxSelected>>",
self.update_original_data_table)

    # Table
        self.original_table = ttk.Treeview(frame,
columns=list(large_df.columns), show='headings')
        self.original_table.pack(expand=True,
fill='both')

    # Scrollbars
        vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.original_table.yview)
        vsb.pack(side='right', fill='y')

self.original_table.configure(yscrollcommand=vsb.s
et)

        hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.original_table.xview)
        hsb.pack(side='bottom', fill='x')

self.original_table.configure(xscrollcommand=hsb.s
et)

    # Define columns
    for col in large_df.columns:
        self.original_table.heading(col, text=col)
        self.original_table.column(col, width=100)

    self.update_original_data_table()    # Initial
load
```

- `ttk.Frame`: Creates a new frame within the notebook for this tab.
- Country Filter: A `ttk.Combobox` widget allows the user to filter the data by country.
- `ttk.Treeview`: A widget that displays the original data in a tabular format with headings for each column.
- Scrollbars: Vertical and horizontal scrollbars are added to handle large datasets.
- `update_original_data_table()`: Populates the table with data based on the selected country filter.

2. **Pivoted Data Tab**

Similar to the Original Data Tab, but displays the pivoted data.

```
def create_pivoted_data_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Pivoted Data')

    # Table
        self.pivoted_table = ttk.Treeview(frame,
columns=list(large_pivoted_df.columns),
show='headings')
        self.pivoted_table.pack(expand=True,
fill='both')

    # Scrollbars
        vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.pivoted_table.yview)
        vsb.pack(side='right', fill='y')

    self.pivoted_table.configure(yscrollcommand=vsb.set)
```

```

        hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.pivoted_table.xview)
hsb.pack(side='bottom', fill='x')

self.pivoted_table.configure(xscrollcommand=hsb.set)

# Define columns
for col in large_pivoted_df.columns:
    self.pivoted_table.heading(col, text=col)
    self.pivoted_table.column(col, width=100)

    self.update_pivoted_data_table() # Initial
load

```

3. Country Summary Tab

Displays a summary of total gold production by country.

```

def create_country_summary_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Country
Summary')

    # Table
    self.country_summary_table =
ttk.Treeview(frame,
columns=list(country_summary_df.columns),
show='headings')
    self.country_summary_table.pack(expand=True,
fill='both')

    # Scrollbars
    vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.country_summary_table.yview)

```

```

        vsb.pack(side='right', fill='y')

self.country_summary_table.configure(yscrollcommand=vsb.set)

        hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.country_summary_table.xview)
        hsb.pack(side='bottom', fill='x')

self.country_summary_table.configure(xscrollcommand=hsb.set)

# Define columns
for col in country_summary_df.columns:
    self.country_summary_table.heading(col,
text=col)
    self.country_summary_table.column(col,
width=100)

    self.update_country_summary_table() # Initial load

```

4. Region Summary Tab

Displays a summary of total gold production by region.

```

def create_region_summary_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Region
Summary')

    # Table
    self.region_summary_table =
ttk.Treeview(frame,

```

```

columns=list(region_summary_df.columns),
show='headings')
    self.region_summary_table.pack(expand=True,
fill='both')

    # Scrollbars
    vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.region_summary_table.yview)
    vsb.pack(side='right', fill='y')

self.region_summary_table.configure(yscrollcommand
=vsb.set)

    hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.region_summary_table.xview)
    hsb.pack(side='bottom', fill='x')

self.region_summary_table.configure(xscrollcommand
=hsb.set)

    # Define columns
    for col in region_summary_df.columns:
        self.region_summary_table.heading(col,
text=col)
        self.region_summary_table.column(col,
width=100)

    self.update_region_summary_table()  # Initial
load

```

5. Year Summary Tab

Displays a summary of total gold production by year.

```
def create_year_summary_tab(self):
```

```
frame = ttk.Frame(self.notebook)
self.notebook.add(frame, text='Year Summary')

# Table
    self.year_summary_table = ttk.Treeview(frame,
columns=list(year_summary_df.columns),
show='headings')
        self.year_summary_table.pack(expand=True,
fill='both')

# Scrollbars
    vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.year_summary_table.yview)
    vsb.pack(side='right', fill='y')

self.year_summary_table.configure(yscrollcommand=vsb.set)

    hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.year_summary_table.xview)
    hsb.pack(side='bottom', fill='x')

self.year_summary_table.configure(xscrollcommand=hsb.set)

# Define columns
for col in year_summary_df.columns:
    self.year_summary_table.heading(col,
text=col)
    self.year_summary_table.column(col,
width=100)

    self.update_year_summary_table() # Initial
load
```

6. Country Distribution Tab

Displays a bar chart of the distribution of gold production by country.

```
def create_country_distribution_tab(self):
    frame = ttk.Frame(self.notebook)
        self.notebook.add(frame,   text='Country
Distribution')

    # Bar Chart
    fig, ax = plt.subplots()
        ax.bar(country_summary_df['Country'],
country_summary_df['Gold_Production_Tonnes'],
color='skyblue')
        ax.set_title('Gold Production by Country')
        ax.set_xlabel('Country')
        ax.set_ylabel('Gold Production (Tonnes)')

    # Embed in Tkinter
    canvas = FigureCanvasTkAgg(fig, master=frame)
        canvas.get_tk_widget().pack(expand=True,
fill='both')
    canvas.draw()
```

7. Region Distribution Tab

Displays a bar chart of the distribution of gold production by region.

```
def create_region_distribution_tab(self):
    frame = ttk.Frame(self.notebook)
        self.notebook.add(frame,   text='Region
Distribution')

    # Bar Chart
    fig, ax = plt.subplots()
```

```

        ax.bar(region_summary_df['Region'],
region_summary_df['Gold_Production_Tonnes'],
color='lightgreen')
        ax.set_title('Gold Production by Region')
        ax.set_xlabel('Region')
        ax.set_ylabel('Gold Production (Tonnes)')

# Embed in Tkinter
canvas = FigureCanvasTkAgg(fig, master=frame)
        canvas.get_tk_widget().pack(expand=True,
fill='both')
        canvas.draw()

```

8. Year Distribution Tab

Displays a bar chart of the distribution of gold production by year.

```

def create_year_distribution_tab(self):
    frame = ttk.Frame(self.notebook)
        self.notebook.add(frame,      text='Year
Distribution')

    # Bar Chart
    fig, ax = plt.subplots()
        ax.bar(year_summary_df['Year'],
year_summary_df['Gold_Production_Tonnes'],
color='lightcoral')
        ax.set_title('Gold Production by Year')
        ax.set_xlabel('Year')
        ax.set_ylabel('Gold Production (Tonnes)')

    # Embed in Tkinter
    canvas = FigureCanvasTkAgg(fig, master=frame)
        canvas.get_tk_widget().pack(expand=True,
fill='both')
        canvas.draw()

```

Update Methods

These methods are responsible for updating the tables when data changes.

- `update_original_data_table()`: Updates the original data table based on the selected country filter.
- `update_pivoted_data_table()`: Updates the pivoted data table.
- `update_country_summary_table()`: Updates the country summary table.
- `update_region_summary_table()`: Updates the region summary table.
- `update_year_summary_table()`: Updates the year summary table.

Running the Application

```
if __name__ == "__main__":
    root = tk.Tk()
    app = RichTkinterApp(root)
    root.mainloop()
```

This block runs the application, creating an instance of `tk.Tk()` as the root window, initializing the `RichTkinterApp`, and starting the main event loop with `root.mainloop()`.

Rich Tkinter GUI for Gold Production Data

Original Data Pivoted Data Country Summary Region Summary Year Summary Country Distribution Region Distribution Year Distribution

Filter by Country:

All

| Country | Year | Region | Mine | Gold_Production_Tonnes |
|--------------|------|--------|--------|------------------------|
| South Africa | 2006 | East | Mine_C | 281 |
| Russia | 2003 | West | Mine_A | 91 |
| Australia | 2023 | West | Mine_B | 141 |
| Russia | 2019 | North | Mine_D | 310 |
| Russia | 2006 | East | Mine_C | 97 |
| Canada | 2003 | North | Mine_B | 458 |
| Australia | 2011 | South | Mine_A | 331 |
| Australia | 2004 | West | Mine_D | 123 |
| Australia | 2022 | West | Mine_A | 265 |
| Russia | 2018 | North | Mine_B | 495 |
| South Africa | 2004 | East | Mine_A | 270 |
| Australia | 2018 | South | Mine_C | 158 |
| Russia | 2020 | East | Mine_B | 184 |
| Canada | 2002 | East | Mine_C | 477 |
| South Africa | 2010 | North | Mine_D | 392 |
| Canada | 2006 | East | Mine_C | 287 |
| South Africa | 2016 | North | Mine_C | 204 |
| Russia | 2024 | West | Mine_A | 475 |
| USA | 2008 | West | Mine_A | 378 |
| South Africa | 2020 | West | Mine_B | 105 |
| Canada | 2008 | South | Mine_D | 436 |
| Russia | 2009 | South | Mine_B | 431 |
| South Africa | 2012 | East | Mine_D | 446 |
| USA | 2014 | North | Mine_D | 247 |
| USA | 2021 | East | Mine_C | 364 |

Rich Tkinter GUI for Gold Production Data

Original Data Pivoted Data Country Summary Region Summary Year Summary Country Distribution Region Distribution Year Distribution

Filter by Country:

South Africa

| Country | Year | Region | Mine | Gold_Production_Tonnes |
|--------------|------|--------|--------|------------------------|
| South Africa | 2006 | East | Mine_C | 281 |
| South Africa | 2004 | East | Mine_A | 270 |
| South Africa | 2010 | North | Mine_D | 392 |
| South Africa | 2016 | North | Mine_C | 204 |
| South Africa | 2020 | West | Mine_B | 105 |
| South Africa | 2012 | East | Mine_D | 446 |
| South Africa | 2019 | East | Mine_D | 266 |
| South Africa | 2009 | West | Mine_A | 280 |
| South Africa | 2002 | East | Mine_A | 306 |
| South Africa | 2021 | West | Mine_C | 483 |
| South Africa | 2017 | South | Mine_C | 133 |
| South Africa | 2001 | South | Mine_D | 382 |
| South Africa | 2024 | West | Mine_A | 167 |
| South Africa | 2017 | North | Mine_A | 186 |
| South Africa | 2019 | North | Mine_D | 116 |
| South Africa | 2024 | South | Mine_B | 440 |
| South Africa | 2016 | West | Mine_C | 172 |
| South Africa | 2024 | North | Mine_B | 277 |
| South Africa | 2019 | West | Mine_A | 295 |
| South Africa | 2001 | East | Mine_A | 458 |
| South Africa | 2024 | South | Mine_D | 364 |
| South Africa | 2013 | North | Mine_B | 151 |
| South Africa | 2022 | East | Mine_A | 407 |
| South Africa | 2007 | West | Mine_C | 392 |
| South Africa | 2001 | East | Mine_B | 172 |

Rich Tkinter GUI for Gold Production Data

| | Original Data | Pivoted Data | Country Summary | Region Summary | Year Summary | Country Distribution | Region Distribution | Year Distribution | | | | |
|------|---------------|--------------|-----------------|----------------|--------------|----------------------|---------------------|-------------------|------|------|------|------|
| | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 |
| 6082 | 5121 | 4384 | 3757 | 6183 | 5237 | 5548 | 6254 | 6004 | 5829 | 2456 | 5371 | |
| 5192 | 4554 | 4213 | 5974 | 4740 | 5066 | 5373 | 3378 | 5787 | 5625 | 6271 | 5035 | |
| 4038 | 4706 | 3551 | 4971 | 3562 | 4051 | 5307 | 4634 | 5489 | 4291 | 3323 | 4121 | |
| 3580 | 4312 | 4669 | 5362 | 4842 | 4578 | 4737 | 6040 | 4060 | 4037 | 3131 | 8192 | |
| 5544 | 5119 | 6084 | 5753 | 2941 | 5487 | 6748 | 5788 | 3786 | 5348 | 3506 | 3894 | |
| 5809 | 8969 | 4353 | 5349 | 5854 | 4779 | 4445 | 8172 | 5652 | 6553 | 5478 | 5249 | |
| 4482 | 3787 | 4275 | 2810 | 6220 | 5790 | 4802 | 4869 | 4488 | 4200 | 3478 | 6048 | |
| 5132 | 6045 | 4021 | 3720 | 6332 | 4649 | 5970 | 4364 | 3804 | 8668 | 5343 | 6341 | |
| 5669 | 4724 | 6551 | 7239 | 7742 | 3873 | 3740 | 4883 | 5886 | 3671 | 3890 | 5157 | |
| 6054 | 5413 | 5111 | 5977 | 5330 | 6679 | 7066 | 6420 | 2907 | 4108 | 5078 | 4261 | |
| 7300 | 5202 | 5561 | 3718 | 5896 | 8536 | 4936 | 4146 | 5925 | 7011 | 4273 | 6751 | |
| 2629 | 7241 | 5944 | 4724 | 4466 | 4518 | 6306 | 6348 | 7614 | 7453 | 5093 | 4012 | |
| 4265 | 5491 | 2633 | 6760 | 5033 | 4513 | 6400 | 4079 | 4518 | 6133 | 1928 | 6422 | |
| 3804 | 3172 | 4888 | 6607 | 6158 | 3549 | 5979 | 7814 | 4829 | 5660 | 3411 | 4649 | |
| 4323 | 6715 | 6261 | 5205 | 2630 | 5342 | 4964 | 5165 | 4486 | 5644 | 5374 | 5927 | |
| 4779 | 4202 | 4821 | 6684 | 5097 | 9046 | 5713 | 5727 | 8851 | 4522 | 5799 | 4203 | |
| 6345 | 7185 | 4366 | 3853 | 3544 | 6424 | 5157 | 5568 | 7341 | 4855 | 4468 | 6761 | |
| 4733 | 6195 | 5697 | 4918 | 5677 | 6411 | 6417 | 7143 | 4017 | 5882 | 4819 | 5037 | |
| 5571 | 3968 | 4929 | 5431 | 4582 | 5087 | 5227 | 6958 | 5266 | 3655 | 4096 | 6013 | |
| 6756 | 5741 | 7732 | 6310 | 7886 | 5365 | 5424 | 5172 | 4578 | 6688 | 6721 | 4133 | |

Rich Tkinter GUI for Gold Production Data

| | Original Data | Pivoted Data | Country Summary | Region Summary | Year Summary | Country Distribution | Region Distribution | Year Distribution | | | |
|--------------|---------------|--------------|-----------------|----------------|--------------|----------------------|---------------------|-------------------|------------------------|--|--|
| | Country | | | | | | | | Gold_Production_Tonnes | | |
| Australia | | | | | | | | | 522678 | | |
| Canada | | | | | | | | | 549812 | | |
| Russia | | | | | | | | | 557670 | | |
| South Africa | | | | | | | | | 546963 | | |
| USA | | | | | | | | | 553675 | | |

Rich Tkinter GUI for Gold Production Data

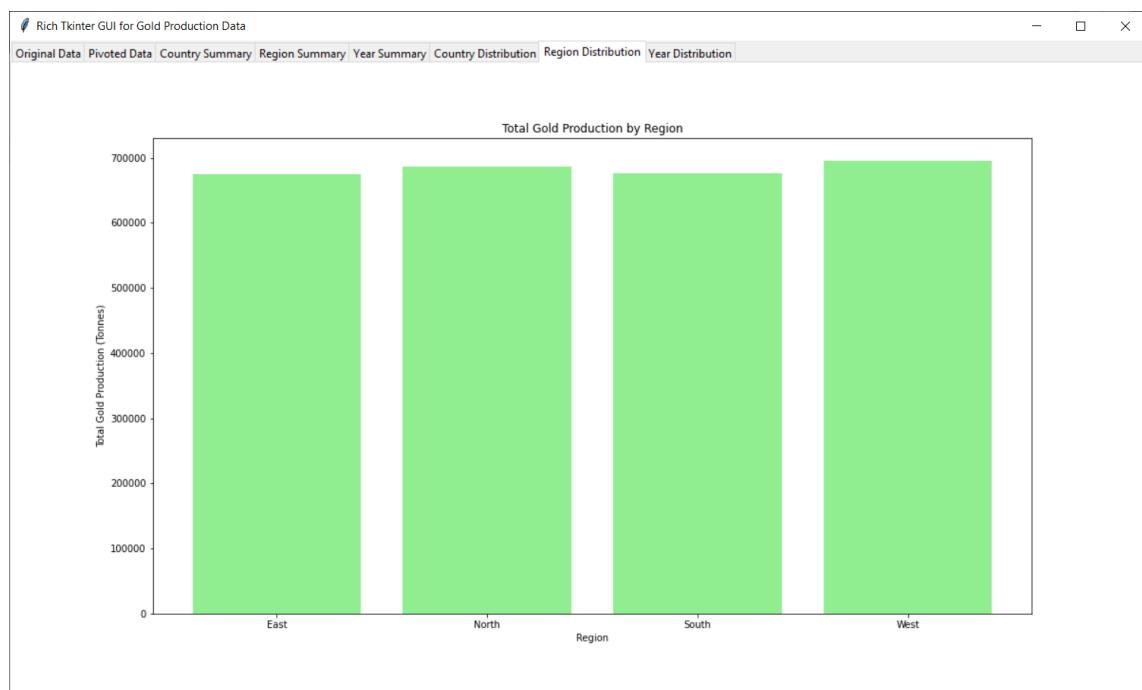
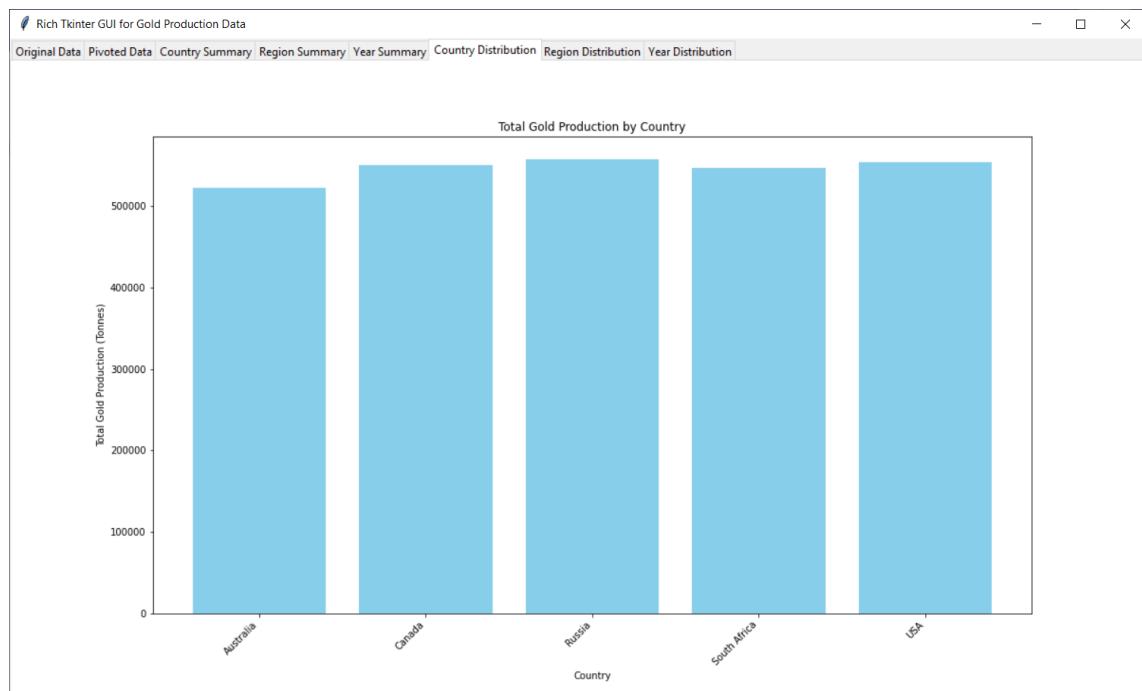
Original Data Pivoted Data Country Summary Region Summary Year Summary Country Distribution Region Distribution Year Distribution

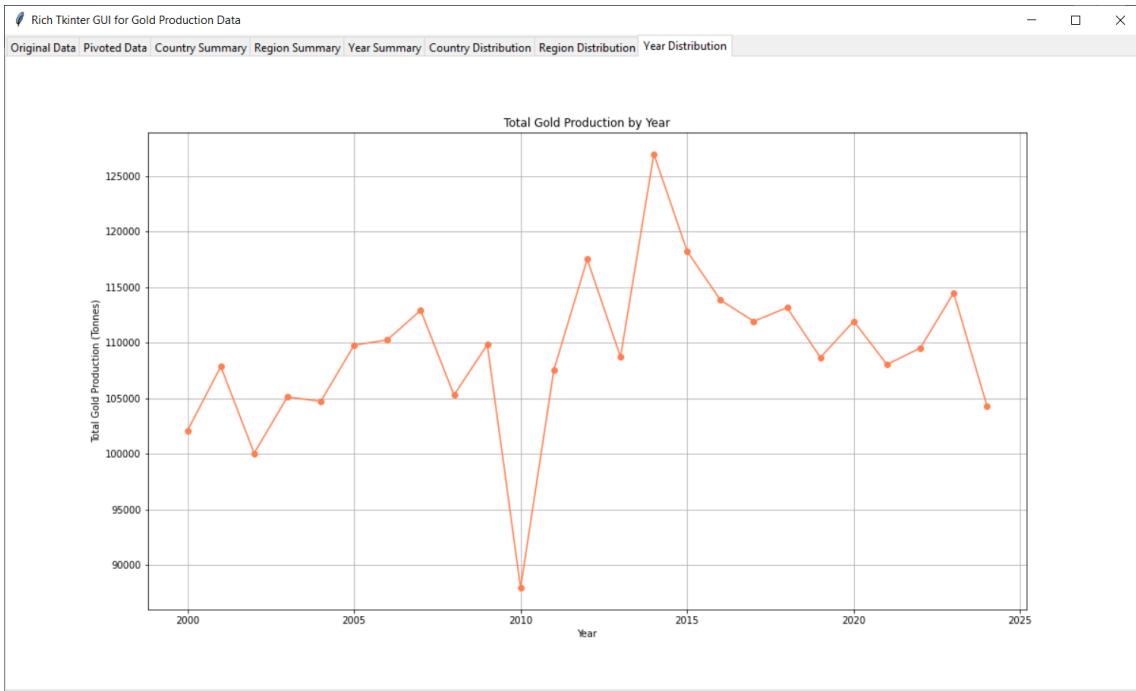
| Region | Gold_Production_Tonnes |
|--------|------------------------|
| East | 673920 |
| North | 685803 |
| South | 675647 |
| West | 695428 |

Rich Tkinter GUI for Gold Production Data

Original Data Pivoted Data Country Summary Region Summary Year Summary Country Distribution Region Distribution Year Distribution

| Year | Gold_Production_Tonnes |
|------|------------------------|
| 2000 | 102087 |
| 2001 | 107862 |
| 2002 | 100044 |
| 2003 | 105122 |
| 2004 | 104715 |
| 2005 | 109780 |
| 2006 | 110259 |
| 2007 | 112922 |
| 2008 | 105288 |
| 2009 | 109833 |
| 2010 | 87936 |
| 2011 | 107577 |
| 2012 | 117509 |
| 2013 | 108718 |
| 2014 | 126998 |
| 2015 | 118232 |
| 2016 | 113840 |
| 2017 | 111931 |
| 2018 | 113157 |
| 2019 | 108674 |
| 2020 | 111933 |
| 2021 | 108039 |
| 2022 | 109529 |
| 2023 | 114489 |
| 2024 | 104324 |





EXAMPLE 5.3

UnPivoting Dataframe with Synthetic Stock Dataset

The purpose of the code is to create a large synthetic dataset simulating stock prices for multiple companies over an extended period, and then to transform and save this data in different formats. The dataset includes daily stock prices for companies like AAPL, GOOG, AMZN, MSFT, TSLA, and META over 10,000 days. This is achieved using random number generation to create a realistic range of stock prices. The resulting DataFrame, `large_stock_df`, represents the data in a wide format where each company's stock prices are in separate columns.

The code then reshapes the DataFrame using the `pd.melt()` function to convert it from a wide format to a long format, which is often more useful for certain types of data analysis and visualization. In this unpivoted format, each row represents a single date, stock, and its price, which makes it easier to analyze trends across stocks and

dates. Finally, the code saves both the original and unpivoted DataFrames to separate Excel files, facilitating further analysis or reporting in Excel.

```
import pandas as pd
import numpy as np

# Seed for reproducibility
np.random.seed(42)
N = 10000

# Create a larger synthetic dataset
data = {
    'Date': pd.date_range(start='2023-01-01',
periods=N),
    'AAPL': np.random.uniform(150, 170, N),
    'GOOG': np.random.uniform(2800, 3000, N),
    'AMZN': np.random.uniform(3300, 3500, N),
    'MSFT': np.random.uniform(250, 270, N),
    'TSLA': np.random.uniform(600, 800, N),
    'META': np.random.uniform(300, 350, N)
}

# Convert to DataFrame
large_stock_df = pd.DataFrame(data)

# Unpivot the DataFrame
unpivoted_large_df = pd.melt(large_stock_df,
id_vars=['Date'], var_name='Stock',
value_name='Price')

# Save both the original and unpivoted DataFrames
# to separate Excel sheets
large_stock_df.to_excel('large_stock_data_original
.xlsx', sheet_name='Original_Data', index=False)
```

```
unpivoted_large_df.to_excel('large_stock_data_unpivoted.xlsx', sheet_name='Unpivoted_Data', index=False)

print("Data saved to
'large_stock_data_original.xlsx' and
'large_stock_data_unpivoted.xlsx'.")
```

Let's break down each part of the code in detail:

1. Importing Libraries

```
import pandas as pd
import numpy as np
```

- pandas (pd): A powerful library for data manipulation and analysis, especially with tabular data like DataFrames.
- numpy (np): A library for numerical computing in Python, used here to generate random numbers.

2. Setting a Seed for Reproducibility

```
np.random.seed(42)
N = 10000
```

- np.random.seed(42): Ensures that the random numbers generated by numpy are reproducible. The same seed will produce the same set of random numbers every time the code is run.
- N = 10000: Specifies the size of the dataset, in this case, generating 10,000 data points for each stock.

3. Creating a Synthetic Dataset

```
data = {  
    'Date': pd.date_range(start='2023-01-01',  
periods=N),  
    'AAPL': np.random.uniform(150, 170, N),  
    'GOOG': np.random.uniform(2800, 3000, N),  
    'AMZN': np.random.uniform(3300, 3500, N),  
    'MSFT': np.random.uniform(250, 270, N),  
    'TSLA': np.random.uniform(600, 800, N),  
    'META': np.random.uniform(300, 350, N)  
}
```

- `pd.date_range(start='2023-01-01', periods=N)`: Creates a sequence of dates starting from January 1, 2023, for 10,000 periods (days).
- `np.random.uniform(...)`: Generates 10,000 random numbers within a specified range for each stock. For example:
 - AAPL: Random prices between \$150 and \$170.
 - GOOG: Random prices between \$2800 and \$3000.
 - Similar ranges are defined for AMZN, MSFT, TSLA, and META.
- `data dictionary`: Combines the dates and generated stock prices into a single data structure. Each key in the dictionary represents a column in the DataFrame.

4. Converting to a DataFrame

```
large_stock_df = pd.DataFrame(data)
```

- `pd.DataFrame(data)`: Converts the data dictionary into a DataFrame, where each key in the dictionary becomes a

column, and each list of values becomes a row.

5. Unpivoting the DataFrame

```
unpivoted_large_df      =      pd.melt(large_stock_df,
id_vars=['Date'],
var_name='Stock',
value_name='Price')
```

- `pd.melt(...)`: Unpivots or reshapes the DataFrame from a wide format to a long format. In this context:
 - `id_vars=['Date']`: Specifies that the 'Date' column should remain fixed.
 - `var_name='Stock'`: All other column names (AAPL, GOOG, AMZN, etc.) are combined into a single 'Stock' column.
 - `value_name='Price'`: The corresponding values (prices) for each stock are placed in a 'Price' column.
- This transformation turns the DataFrame into a format where each row represents a single date, stock, and its price, making it easier to work with for certain types of analysis.

6. Saving DataFrames to Excel

```
large_stock_df.to_excel('large_stock_data_original
.xlsx', sheet_name='Original_Data', index=False)
unpivoted_large_df.to_excel('large_stock_data_unpi
voted.xlsx',
sheet_name='Unpivoted_Data',
index=False)
```

- `to_excel(...)`: Saves the DataFrames to Excel files.
 - '`large_stock_data_original.xlsxOriginal_Data`'.
 - '`large_stock_data_unpivoted.xlsxUnpivoted_Data`'.
 - `index=False`: Omits the DataFrame index from being written to the Excel file.

7. Printing Confirmation

```
print("Data           saved          to
'large_stock_data_original.xlsx'           and
'large_stock_data_unpivoted.xlsx'.")
```

- `print(...)`: Outputs a confirmation message to the console, indicating that the data has been successfully saved to the specified Excel files.

EXAMPLE 5.4

GUI Tkinter for UnPivoting Dataframe with Synthetic Stock Dataset

The purpose of this project is to create a visually rich and interactive graphical user interface (GUI) using Tkinter to analyze and visualize stock market data. The core functionality of the project involves working with a large synthetic dataset of stock prices, which is generated to simulate real-world financial data. The dataset includes stock prices for various companies over a period, and is processed to create both an original and an unpivoted

version, which are then saved to Excel files. This allows users to view and interact with both the raw and transformed data.

The GUI is designed to be highly functional and aesthetically pleasing, featuring multiple tabs for different types of data visualization. The main tabs include "Original Data," which displays the raw stock price data in a table format; "Unpivoted Data," which shows the transformed data where stock prices are melted into a long format; "Summary," which provides statistical summaries such as average, maximum, and minimum prices for each stock; and "Graphs," which presents graphical visualizations of the stock prices and their distributions. The use of alternating row colors in tables enhances readability, while the integration of Matplotlib charts provides a clear and engaging way to analyze trends and distributions in the data.

Overall, the project aims to offer a comprehensive tool for exploring and analyzing stock market data through a user-friendly interface. By combining data manipulation, statistical analysis, and visualization in a single application, users can gain insights into stock performance and market trends with ease. The use of Tkinter for the GUI ensures that the application is accessible and intuitive, making it suitable for both casual users and those with more advanced data analysis needs.

```
import tkinter as tk
from tkinter import ttk
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
```

```
# Seed for reproducibility
np.random.seed(42)
N = 10000

# Create a larger synthetic dataset
data = {
    'Date': pd.date_range(start='2023-01-01',
periods=N),
    'AAPL': np.random.uniform(150, 170, N),
    'GOOG': np.random.uniform(2800, 3000, N),
    'AMZN': np.random.uniform(3300, 3500, N),
    'MSFT': np.random.uniform(250, 270, N),
    'TSLA': np.random.uniform(600, 800, N),
    'META': np.random.uniform(300, 350, N)
}

# Convert to DataFrame
large_stock_df = pd.DataFrame(data)

# Unpivot the DataFrame
unpivoted_large_df = pd.melt(large_stock_df,
id_vars=['Date'], var_name='Stock',
value_name='Price')

# Calculate summary statistics
summary_data = {
    'Stock': ['AAPL', 'GOOG', 'AMZN', 'MSFT',
'TSLA', 'META'],
    'Average Price': [
        large_stock_df['AAPL'].mean(),
        large_stock_df['GOOG'].mean(),
        large_stock_df['AMZN'].mean(),
        large_stock_df['MSFT'].mean(),
        large_stock_df['TSLA'].mean(),
        large_stock_df['META'].mean()
    ],
}
```

```

    'Max Price': [
        large_stock_df['AAPL'].max(),
        large_stock_df['GOOG'].max(),
        large_stock_df['AMZN'].max(),
        large_stock_df['MSFT'].max(),
        large_stock_df['TSLA'].max(),
        large_stock_df['META'].max()
    ],
    'Min Price': [
        large_stock_df['AAPL'].min(),
        large_stock_df['GOOG'].min(),
        large_stock_df['AMZN'].min(),
        large_stock_df['MSFT'].min(),
        large_stock_df['TSLA'].min(),
        large_stock_df['META'].min()
    ]
}
summary_df = pd.DataFrame(summary_data)

# Save both the original and unpivoted DataFrames
# to separate Excel sheets
large_stock_df.to_excel('large_stock_data_original.xlsx', sheet_name='Original_Data', index=False)
unpivoted_large_df.to_excel('large_stock_data_unpivoted.xlsx', sheet_name='Unpivoted_Data',
index=False)

print("Data saved to
'large_stock_data_original.xlsx' and
'large_stock_data_unpivoted.xlsx'.")
```

```

class RichTkinterApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Rich Tkinter GUI for
Stock Data")
```

```
self.root.geometry("1200x800")

# Notebook (Tabs)
self.notebook = ttk.Notebook(root)
self.notebook.pack(expand=True,
fill='both')

# Create Tabs
self.create_original_data_tab()
self.create_unpivoted_data_tab()
self.create_summary_tab()
self.create_graphs_tab()

def create_original_data_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Original
Data')

    # Table
    self.original_table = ttk.Treeview(frame,
columns=list(large_stock_df.columns),
show='headings')
    self.original_table.pack(expand=True,
fill='both')

    # Scrollbars
    vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.original_table.yview)
    vsb.pack(side='right', fill='y')

    self.original_table.configure(yscrollcommand=vsb.
set)

    hsb = ttk.Scrollbar(frame,
orient="horizontal",
```

```
command=self.original_table.xview)
    hsb.pack(side='bottom', fill='x')

    self.original_table.configure(xscrollcommand=hsb.
set)

        # Define columns
        for col in large_stock_df.columns:
            self.original_table.heading(col,
text=col)
            self.original_table.column(col,
width=150)

            self.update_original_data_table() # Initial load

    def create_unpivoted_data_tab(self):
        frame = ttk.Frame(self.notebook)
        self.notebook.add(frame, text='Unpivoted Data')

        # Table
        self.unpivoted_table = ttk.Treeview(frame,
columns=list(unpivoted_large_df.columns),
show='headings')
        self.unpivoted_table.pack(expand=True,
fill='both')

        # Scrollbars
        vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.unpivoted_table.yview)
        vsb.pack(side='right', fill='y')

        self.unpivoted_table.configure(yscrollcommand=vsb.
set)
```

```
        hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.unpivoted_table.xview)
        hsb.pack(side='bottom', fill='x')

    self.unpivoted_table.configure(xscrollcommand=hsb
.set)

        # Define columns
        for col in unpivoted_large_df.columns:
            self.unpivoted_table.heading(col,
text=col)
            self.unpivoted_table.column(col,
width=150)

        self.update_unpivoted_data_table() # Initial load

    def create_summary_tab(self):
        frame = ttk.Frame(self.notebook)
        self.notebook.add(frame, text='Summary')

        # Table
        self.summary_table = ttk.Treeview(frame,
columns=list(summary_df.columns), show='headings')
        self.summary_table.pack(expand=True,
fill='both')

        # Scrollbars
        vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.summary_table.yview)
        vsb.pack(side='right', fill='y')

    self.summary_table.configure(yscrollcommand=vsb.s
```

```
et)

        hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.summary_table.xview)
        hsb.pack(side='bottom', fill='x')

    self.summary_table.configure(xscrollcommand=hsb.s
et)

    # Define columns
    for col in summary_df.columns:
        self.summary_table.heading(col,
text=col)
        self.summary_table.column(col,
width=150)

    self.update_summary_table() # Initial
load

def create_graphs_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Graphs')

    # Create and display figures
    self.plot_stock_prices(frame)
    self.plot_stock_price_distributions(frame)

def plot_stock_prices(self, frame):
    fig, ax = plt.subplots(figsize=(12, 6))
    for stock in ['AAPL', 'GOOG', 'AMZN',
'MSFT', 'TSLA', 'META']:
        ax.plot(large_stock_df['Date'],
large_stock_df[stock], label=stock)
    ax.set_title('Stock Prices Over Time')
    ax.set_xlabel('Date')
```

```
        ax.set_ylabel('Price')
        ax.legend()

        # Display figure in Tkinter
        canvas = FigureCanvasTkAgg(fig,
master=frame)
        canvas.draw()
        canvas.get_tk_widget().pack(expand=True,
fill='both')

    def plot_stock_price_distributions(self,
frame):
        fig, axs = plt.subplots(2, 3, figsize=(18,
12))
        stocks = ['AAPL', 'GOOG', 'AMZN', 'MSFT',
'TSLA', 'META']
        for ax, stock in zip(axs.flatten(),
stocks):
            ax.hist(large_stock_df[stock],
bins=30, alpha=0.7, label=stock)
            ax.set_title(f'{stock} Price
Distribution')
            ax.set_xlabel('Price')
            ax.set_ylabel('Frequency')
            ax.legend()

        # Display figure in Tkinter
        canvas = FigureCanvasTkAgg(fig,
master=frame)
        canvas.draw()
        canvas.get_tk_widget().pack(expand=True,
fill='both')

    def update_original_data_table(self):
        # Clear current data
```

```
        for row in
self.original_table.get_children():
    self.original_table.delete(row)

        # Insert new data with alternating row
colors
        for i, row in
enumerate(large_stock_df.itertuples(index=False)):
            item_id =
self.original_table.insert('', 'end', values=row)
            if i % 2 == 0:

                self.original_table.tag_configure('evenrow',
background='#f0f0f0')
                    self.original_table.item(item_id,
tags='evenrow')
            else:

                self.original_table.tag_configure('oddrow',
background='#ffffff')
                    self.original_table.item(item_id,
tags='oddrow')
```

```

def update_unpivoted_data_table(self):
    # Clear current data
    for row in
self.unpivoted_table.get_children():
    self.unpivoted_table.delete(row)

        # Insert new data with alternating row
colors
    for i, row in
enumerate(unpivoted_large_df.itertuples(index=False)):
        item_id =
    self.unpivoted_table.insert('', 'end',
values=row)
        if i % 2 == 0:

            self.unpivoted_table.tag_configure('evenrow',
background='#f0f0f0')
                self.unpivoted_table.item(item_id,
tags='evenrow')
            else:

                self.unpivoted_table.tag_configure('oddrow',
background='#ffffff')
                    self.unpivoted_table.item(item_id,
tags='oddrow')

def update_summary_table(self):
    # Clear current data
    for row in
self.summary_table.get_children():
    self.summary_table.delete(row)

        # Insert new data

```

```

        for row in
summary_df.itertuples(index=False):
            self.summary_table.insert('', 'end',
values=row)

if __name__ == "__main__":
    root = tk.Tk()
    app = RichTkinterApp(root)
    root.mainloop()

```

Here's the explanation of each part of the code:

Importing Libraries

```

import tkinter as tk
from tkinter import ttk
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from    matplotlib.backends.backend_tkagg    import
FigureCanvasTkAgg

```

- **tkinter** and **ttk**: These modules are part of the Python standard library for creating graphical user interfaces (GUIs). **tkinter** provides the base classes and functions for creating windows, dialogs, and widgets. **ttk** (Themed Tkinter) offers more advanced and aesthetically pleasing widgets compared to the standard Tkinter widgets.
- **pandas**: This library is used for data manipulation and analysis. It provides data structures like **DataFrames**, which are useful for handling structured data efficiently.
- **numpy**: A fundamental library for numerical operations in Python. It provides support for large, multi-dimensional arrays and matrices, and it includes mathematical functions to operate on these arrays.

- `matplotlib.pyplot`: A plotting library for creating static, interactive, and animated visualizations in Python. `pyplot` provides a MATLAB-like interface for creating plots and charts.
- `FigureCanvasTkAgg`: This class from `matplotlib.backends.backend_tkagg` allows Matplotlib figures to be embedded in Tkinter applications. It acts as a bridge between Matplotlib's plotting functions and Tkinter's GUI framework.

Data Preparation

```
# Seed for reproducibility
np.random.seed(42)
N = 10000

# Create a larger synthetic dataset
data = {
    'Date': pd.date_range(start='2023-01-01',
periods=N),
    'AAPL': np.random.uniform(150, 170, N),
    'GOOG': np.random.uniform(2800, 3000, N),
    'AMZN': np.random.uniform(3300, 3500, N),
    'MSFT': np.random.uniform(250, 270, N),
    'TSLA': np.random.uniform(600, 800, N),
    'META': np.random.uniform(300, 350, N)
}

# Convert to DataFrame
large_stock_df = pd.DataFrame(data)
```

- `np.random.seed(42)`: Sets the seed for the random number generator. This ensures that every time the code is run,

the same set of random numbers is generated, which is useful for debugging and reproducibility.

- $N = 10000$: Defines the number of data points to generate. In this case, 10,000 data points are created for each stock.
- `data`: A dictionary where:
 - 'Date' is generated using `pd.date_range()`, creating a sequence of dates starting from January 1, 2023, for 10,000 periods.
 - Each stock's prices ('AAPL', 'GOOG', 'AMZN', 'MSFT', 'TSLA', and 'META') are generated using `np.random.uniform()`, which creates a uniform distribution of prices within specified ranges.
- `large_stock_df`: Converts the data dictionary into a pandas DataFrame. Each key in the dictionary becomes a column in the DataFrame, and each value becomes the data in that column.

Data Transformation

```
# Unpivot the DataFrame
unpivoted_large_df      =      pd.melt(large_stock_df,
id_vars=['Date'],
var_name='Stock',
value_name='Price')
```

- `pd.melt()`: This function reshapes the DataFrame from a wide format to a long format. In the wide format, each stock has its own column, while in the long format:
 - 'Date' is kept as an identifier variable (`id_vars`).
 - The stock names become a single column ('Stock'), with their corresponding prices in another column ('Price').

- unpivoted_large_df: The resulting DataFrame from pd.melt(), which is now easier to plot and analyze in a long format.

Summary Statistics

```
# Calculate summary statistics
summary_data = {
    'Stock': ['AAPL', 'GOOG', 'AMZN', 'MSFT',
'TSLA', 'META'],
    'Average Price': [
        large_stock_df['AAPL'].mean(),
        large_stock_df['GOOG'].mean(),
        large_stock_df['AMZN'].mean(),
        large_stock_df['MSFT'].mean(),
        large_stock_df['TSLA'].mean(),
        large_stock_df['META'].mean()
    ],
    'Max Price': [
        large_stock_df['AAPL'].max(),
        large_stock_df['GOOG'].max(),
        large_stock_df['AMZN'].max(),
        large_stock_df['MSFT'].max(),
        large_stock_df['TSLA'].max(),
        large_stock_df['META'].max()
    ],
    'Min Price': [
        large_stock_df['AAPL'].min(),
        large_stock_df['GOOG'].min(),
        large_stock_df['AMZN'].min(),
        large_stock_df['MSFT'].min(),
        large_stock_df['TSLA'].min(),
        large_stock_df['META'].min()
    ]
}
```

```
summary_df = pd.DataFrame(summary_data)
```

- summary_data: A dictionary that holds summary statistics for each stock. It includes:
 - 'Stock': List of stock tickers.
 - 'Average Price': The average price of each stock, computed using the .mean() method on the respective columns.
 - 'Max Price': The maximum price of each stock, computed using the .max() method.
 - 'Min Price': The minimum price of each stock, computed using the .min() method.
- summary_df: Converts the summary_data dictionary into a pandas DataFrame for easier display and analysis.

Saving Data to Excel

```
# Save both the original and unpivoted DataFrames  
# to separate Excel sheets  
large_stock_df.to_excel('large_stock_data_original.xlsx', sheet_name='Original_Data', index=False)  
unpivoted_large_df.to_excel('large_stock_data_unpivoted.xlsx', sheet_name='Unpivoted_Data',  
index=False)  
  
print("Data saved to  
'large_stock_data_original.xlsx' and  
'large_stock_data_unpivoted.xlsx'.")
```

- large_stock_df.to_excel(): Saves the original DataFrame to an Excel file named 'large_stock_data_original.xlsx', with the sheet name 'Original_Data'. The index=False argument prevents saving the DataFrame index to the file.

- `unpivoted_large_df.to_excel()`: Saves the unpivoted DataFrame to an Excel file named 'large_stock_data_unpivoted.xlsx', with the sheet name 'Unpivoted_Data'.
- `print()`: Outputs a confirmation message that the data has been saved to the specified Excel files.

Tkinter Application Class: RichTkinterApp

```
class RichTkinterApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Rich Tkinter GUI for Stock Data")
        self.root.geometry("1200x800")

        # Notebook (Tabs)
        self.notebook = ttk.Notebook(root)
        self.notebook.pack(expand=True,
fill='both')

        # Create Tabs
        self.create_original_data_tab()
        self.create_unpivoted_data_tab()
        self.create_summary_tab()
        self.create_graphs_tab()
```

- `__init__(self, root)`: The initializer method for the RichTkinterApp class. It sets up the main window and GUI components.
 - `self.root.title()`: Sets the title of the main window.
 - `self.root.geometry()`: Sets the size of the main window (1200x800 pixels).

- `ttk.Notebook()`: Creates a tabbed interface (Notebook) for organizing multiple views or tabs in the GUI.
- `self.notebook.pack()`: Adds the Notebook to the main window, expanding to fill both width and height.

Creating Tabs

```
def create_original_data_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Original Data')

    # Table
    self.original_table = ttk.Treeview(frame,
columns=list(large_stock_df.columns),
show='headings')
    self.original_table.pack(expand=True,
fill='both')

    # Scrollbars
    vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.original_table.yview)
    vsb.pack(side='right', fill='y')

    self.original_table.configure(yscrollcommand=vsb.s
et)

    hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.original_table.xview)
    hsb.pack(side='bottom', fill='x')

    self.original_table.configure(xscrollcommand=hsb.s
```

```

        et)

        # Define columns
        for col in large_stock_df.columns:
            self.original_table.heading(col, text=col)
            self.original_table.column(col, width=150)

        self.update_original_data_table()    # Initial
load

```

- `ttk.Frame(self.notebook)`: Creates a new frame (container) for the tab's contents.
- `self.notebook.add(frame, text='Original Data')`: Adds the frame as a new tab in the Notebook with the title 'Original Data'.
- `ttk.Treeview(frame, columns=list(large_stock_df.columns), show='headings')`: Creates a table view widget to display the DataFrame. The columns parameter specifies the column names, and `show='headings'` indicates that only the column headers should be shown, not the default tree view.
- `self.original_table.pack()`: Adds the table to the frame and configures it to expand and fill both the width and height.
- `ttk.Scrollbar(frame, orient="vertical", command=self.original_table.yview)`: Creates a vertical scrollbar for the table. `command=self.original_table.yview` links the scrollbar to the table's vertical scroll.
- `vsb.pack()`: Adds the vertical scrollbar to the frame.
- `self.original_table.configure(yscrollcommand=vsb.set)`: Configures the table to update the scrollbar position.
- `ttk.Scrollbar(frame, orient="horizontal", command=self.original_table.xview)`: Creates a

- horizontal scrollbar for the table.
- `hsb.pack()`: Adds the horizontal scrollbar to the frame.
 - `self.original_table.configure(xscrollcommand=hsb.set)`: Configures the table to update the horizontal scrollbar position.
 - `self.original_table.heading()` and `self.original_table.column()`: Define the headings and column widths of the table.
 - `self.update_original_data_table()`: Calls a method to populate the table with data.

Creating Unpivoted Data Tab

```
def create_unpivoted_data_tab(self):  
    frame = ttk.Frame(self.notebook)  
    self.notebook.add(frame, text='Unpivoted  
Data')  
  
    # Table  
    self.unpivoted_table = ttk.Treeview(frame,  
columns=list(unpivoted_large_df.columns),  
show='headings')  
    self.unpivoted_table.pack(expand=True,  
fill='both')  
  
    # Scrollbars  
    vsb = ttk.Scrollbar(frame, orient="vertical",  
command=self.unpivoted_table.yview)  
    vsb.pack(side='right', fill='y')  
  
    self.unpivoted_table.configure(yscrollcommand=vsb.  
set)
```

```

        hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.unpivoted_table.xview)
hsb.pack(side='bottom', fill='x')

self.unpivoted_table.configure(xscrollcommand=hsb.
set)

# Define columns
for col in unpivoted_large_df.columns:
    self.unpivoted_table.heading(col,
text=col)
    self.unpivoted_table.column(col,
width=150)

    self.update_unpivoted_data_table() # Initial
load

```

- This method is similar to `create_original_data_tab()`, but it creates a tab for the unpivoted DataFrame. It follows the same steps: creating a frame, adding a table view with scrollbars, and configuring column headings and widths.

Creating Summary Tab

```

def create_summary_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Summary')

    # Table
        self.summary_table = ttk.Treeview(frame,
columns=list(summary_df.columns), show='headings')
        self.summary_table.pack(expand=True,
fill='both')

```

```

# Scrollbars
    vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.summary_table.yview)
    vsb.pack(side='right', fill='y')

self.summary_table.configure(yscrollcommand=vsb.set)

        hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.summary_table.xview)
    hsb.pack(side='bottom', fill='x')

self.summary_table.configure(xscrollcommand=hsb.set)

# Define columns
for col in summary_df.columns:
    self.summary_table.heading(col, text=col)
    self.summary_table.column(col, width=150)

self.update_summary_table() # Initial load

```

- This method creates a tab for displaying summary statistics. It follows the same structure as the previous methods, but uses `summary_df` for the table's data.

Creating Graphs Tab

```

def create_graphs_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Graphs')

```

```
# Create and display figures
self.plot_stock_prices(frame)
self.plot_stock_price_distributions(frame)
```

- self.plot_stock_prices(frame) and
self.plot_stock_price_distributions(frame): Calls methods to create and display graphs in the 'Graphs' tab.

Plotting Stock Prices

```
def plot_stock_prices(self, frame):
    fig, ax = plt.subplots(figsize=(12, 6))
    for stock in ['AAPL', 'GOOG', 'AMZN', 'MSFT',
'TSLA', 'META']:
        ax.plot(large_stock_df['Date'],
large_stock_df[stock], label=stock)
    ax.set_title('Stock Prices Over Time')
    ax.set_xlabel('Date')
    ax.set_ylabel('Price')
    ax.legend()

    # Display figure in Tkinter
    canvas = FigureCanvasTkAgg(fig, master=frame)
    canvas.draw()
    canvas.get_tk_widget().pack(expand=True,
fill='both')
```

- plt.subplots(figsize=(12, 6)): Creates a Matplotlib figure and axes with a specified size (12x6 inches).
- ax.plot(): Plots the stock prices for each stock on the axes.
- ax.set_title(), ax.set_xlabel(), ax.set_ylabel(): Sets the title and labels for the plot.

- `ax.legend()`: Adds a legend to the plot to identify each stock.
- `FigureCanvasTkAgg(fig, master=frame)`: Embeds the Matplotlib figure into a Tkinter frame.
- `canvas.draw()`: Renders the figure.
- `canvas.get_tk_widget().pack()`: Adds the canvas widget to the Tkinter frame and configures it to expand and fill both dimensions.

Plotting Stock Price Distributions

```
def plot_stock_price_distributions(self, frame):
    fig, axs = plt.subplots(2, 3, figsize=(18, 12))
    stocks = ['AAPL', 'GOOG', 'AMZN', 'MSFT',
    'TSLA', 'META']
    for ax, stock in zip(axs.flatten(), stocks):
        ax.hist(large_stock_df[stock], bins=30,
        alpha=0.7, label=stock)
        ax.set_title(f'{stock} Price
Distribution')
        ax.set_xlabel('Price')
        ax.set_ylabel('Frequency')
        ax.legend()

    # Display figure in Tkinter
    canvas = FigureCanvasTkAgg(fig, master=frame)
    canvas.draw()
    canvas.get_tk_widget().pack(expand=True,
    fill='both')
```

- `plt.subplots(2, 3, figsize=(18, 12))`: Creates a Matplotlib figure with a 2x3 grid of subplots, each with a size of

18x12 inches.

- ax.hist(): Creates histograms for the stock prices on each subplot.
- ax.set_title(), ax.set_xlabel(), ax.set_ylabel(): Sets the title and labels for each subplot.
- ax.legend(): Adds a legend to each subplot.
- FigureCanvasTkAgg(fig, master=frame): Embeds the figure into the Tkinter frame.
- canvas.draw() and canvas.get_tk_widget().pack(): Renders the figure and adds it to the Tkinter frame.

Updating Tables

```
def update_original_data_table(self):  
    # Clear current data  
    for row in self.original_table.get_children():  
        self.original_table.delete(row)  
  
    # Insert new data with alternating row colors  
    for i, row in enumerate(large_stock_df.itertuples(index=False)):  
        item_id = self.original_table.insert('',  
'end', values=row)  
        if i % 2 == 0:  
  
            self.original_table.tag_configure('evenrow',  
background='#f0f0f0')  
            self.original_table.item(item_id,  
tags='evenrow')  
        else:  
  
            self.original_table.tag_configure('oddrow',  
background='#ffffff')
```

```
        self.original_table.item(item_id,
tags='oddrow' )
```

- `self.original_table.get_children()`: Retrieves all current rows in the table.
- `self.original_table.delete(row)`: Deletes each row from the table.
- `enumerate(large_stock_df.itertuples(index=False))`: Iterates over DataFrame rows without including the index.
- `self.original_table.insert()`: Inserts each row into the table.
- `self.original_table.tag_configure()`: Configures row tags for alternating row colors.
- `self.original_table.item()`: Applies the row tag (color) to each row.

```
def update_unpivoted_data_table(self):
    # Clear current data
                for         row         in
self.unpivoted_table.get_children():
    self.unpivoted_table.delete(row)

    # Insert new data with alternating row colors
                for         i,         row         in
enumerate(unpivoted_large_df.itertuples(index=False)):
        item_id = self.unpivoted_table.insert('', 'end', values=row)
        if i % 2 == 0:

    self.unpivoted_table.tag_configure('evenrow',
background='#f0f0f0')
```

```
                self.unpivoted_table.item(item_id,
tags='evenrow')
            else:

self.unpivoted_table.tag_configure('oddrow',
background='#ffffff')
                self.unpivoted_table.item(item_id,
tags='oddrow')
```

- This method is similar to `update_original_data_table()`, but it updates the unpivoted DataFrame table.

```
def update_summary_table(self):
    # Clear current data
    for row in self.summary_table.get_children():
        self.summary_table.delete(row)

    # Insert new data with alternating row colors
    for i, row in enumerate(summary_df.itertuples(index=False)):
        item_id = self.summary_table.insert('', 'end', values=row)
        if i % 2 == 0:

self.summary_table.tag_configure('evenrow',
background="#f0f0f0")
                self.summary_table.item(item_id,
tags='evenrow')
            else:

self.summary_table.tag_configure('oddrow',
background='#ffffff')
                self.summary_table.item(item_id,
tags='oddrow')
```

- This method updates the summary statistics table, following the same pattern as the previous methods.

| Rich Tkinter GUI for Stock Data | | | | | | |
|---------------------------------|---------------------|--------------------|---------------------|--------------------|-------------------|--------------------|
| | Original Data | Unpivoted Data | Summary | Graphs | | |
| Date | AAPL | GOOG | AMZN | MSFT | TSLA | META |
| 2023-01-01 00:00:00 | 157.49080237694724 | 2874.72816369334 | 3445.999662197994 | 262.7628913674435 | 659.7824082408833 | 342.3618288482097 |
| 2023-01-02 00:00:00 | 169.0142861281983 | 2866.5824192462806 | 3336.9023991197423 | 259.18584907068384 | 618.963550324116 | 324.7258521859685 |
| 2023-01-03 00:00:00 | 164.6398788362281 | 2835.230782500572 | 3369.327938873977 | 269.28997049727883 | 625.271844994432 | 309.7732805316093 |
| 2023-01-04 00:00:00 | 161.97316968394074 | 2921.4533340202975 | 3432.656127371564 | 254.37956901968477 | 636.1342253631844 | 336.83208933050463 |
| 2023-01-05 00:00:00 | 153.12037280884874 | 2895.324832101726 | 3396.417868904896 | 261.7571283199788 | 640.706668729897 | 320.9339067841648 |
| 2023-01-06 00:00:00 | 153.11989040672404 | 2973.140198464801 | 3447.77142077269787 | 264.00420029980444 | 648.4523618421084 | 329.73137417262484 |
| 2023-01-07 00:00:00 | 151.161672243364 | 2806.4219160964117 | 3492.241580244572 | 265.51128978690764 | 651.0920057926988 | 305.36326450869694 |
| 2023-01-08 00:00:00 | 167.3235229154987 | 2928.7735855110254 | 3323.309337543094 | 258.13942060856016 | 691.1432690975072 | 331.5791994757996 |
| 2023-01-09 00:00:00 | 162.0223002348642 | 2952.589775702312 | 3441.913543279612 | 267.3784381531496 | 701.9146381652082 | 318.6775244725625 |
| 2023-01-10 00:00:00 | 164.1614516559209 | 2951.8973138376614 | 3346.608831126423 | 256.0640294276857 | 661.7758416252711 | 316.70948748592474 |
| 2023-01-11 00:00:00 | 150.41168988591605 | 2977.2147934191053 | 3382.8953454815137 | 258.70952184723643 | 781.4592839041456 | 348.99577893889006 |
| 2023-01-12 00:00:00 | 169.3981970432399 | 2945.806748565898 | 3306.572545159028 | 267.912542624068 | 710.627946677909 | 321.62153213801236 |
| 2023-01-13 00:00:00 | 166.64885281600843 | 2985.5620123295394 | 3327.7184764585397 | 268.9881521866191 | 757.4953856066813 | 333.9019189057867 |
| 2023-01-14 00:00:00 | 154.24678221356552 | 2866.5313199641782 | 3363.95545708978 | 264.3763469385453 | 784.8745664816687 | 331.38637677869207 |
| 2023-01-15 00:00:00 | 153.636499344142 | 2900.6418833503085 | 3368.397161057597 | 257.48707291515237 | 632.6270229151611 | 305.21517754425787 |
| 2023-01-16 00:00:00 | 153.66809019706866 | 2802.8159326565783 | 3479.9170432942193 | 266.40115579949327 | 685.5748967431269 | 337.3963768532158 |
| 2023-01-17 00:00:00 | 156.08484485919075 | 2801.391517460658 | 3448.362381853303 | 250.5173416345084 | 686.6420803429218 | 316.15796017167884 |
| 2023-01-18 00:00:00 | 160.4951286324476 | 2848.025324090304 | 3494.4365933674084 | 252.05491821078843 | 670.8944500876709 | 339.9469774598368 |
| 2023-01-19 00:00:00 | 158.63890037284233 | 2820.161439427817 | 3419.814302067048 | 263.22747472750723 | 601.781173310328 | 330.3820391307691 |
| 2023-01-20 00:00:00 | 155.82458280396085 | 2852.0422735871507 | 3348.3427464182637 | 255.82497150540115 | 763.194421555619 | 315.8545744976745 |
| 2023-01-21 00:00:00 | 162.2370578944476 | 2835.4086589056797 | 3365.5310059843287 | 269.02772436800956 | 622.3118643107806 | 317.8146480165535 |
| 2023-01-22 00:00:00 | 152.78987721304082 | 2805.70400496773 | 3363.9671846583933 | 258.72122418461 | 799.0961195843518 | 336.575368483318 |
| 2023-01-23 00:00:00 | 155.84289297070436 | 2981.86082928121 | 3365.298865326503 | 258.5872964077835 | 754.5916983822364 | 327.5029657561441 |
| 2023-01-24 00:00:00 | 157.327233686587383 | 2801.644626277778 | 3434.59234623636 | 265.56159020381864 | 716.0020035213029 | 333.41755203356576 |
| 2023-01-25 00:00:00 | 159.12139968434073 | 2947.216428867471 | 3406.1615856180833 | 265.3513712461238 | 630.9212835483195 | 307.9159160537594 |
| 2023-01-26 00:00:00 | 165.70351922786028 | 2830.4295773839035 | 3377.0848356173287 | 259.5111747625027 | 703.0785942786495 | 339.3488336114209 |
| 2023-01-27 00:00:00 | 153.9934756431672 | 2982.44592439992 | 3329.817953745743 | 254.28001803423447 | 652.2217481715616 | 316.01956594575176 |
| 2023-01-28 00:00:00 | 160.28468876827225 | 2978.5591879179774 | 3304.4278325884293 | 265.4661728545124 | 650.4645049086139 | 330.99306335788367 |
| 2023-01-29 00:00:00 | 161.84829137724086 | 2930.780229738515 | 3343.7669355175385 | 256.40702549747255 | 765.9058489764971 | 305.73399126210046 |
| 2023-01-30 00:00:00 | 150.92000825439995 | 2934.4468299601212 | 3321.4942807937364 | 253.50221538587715 | 676.1288847918869 | 316.07101205784835 |

Main Application

```
if __name__ == "__main__":
    root = tk.Tk()
    app = RichTkinterApp(root)
    root.mainloop()
```

- if `__name__ == "__main__"`:: Ensures that the code block runs only when the script is executed directly, not when imported as a module.
- `tk.Tk()::` Creates the main Tkinter window.
- `RichTkinterApp(root)`: Initializes the application with the main window.

- `root.mainloop()`: Starts the Tkinter event loop, making the application responsive to user interactions.

Rich Tkinter GUI for Stock Data

Original Data Unpivoted Data Summary Graphs

| Date | Stock | Price |
|---------------------|-------|--------------------|
| 2023-01-01 00:00:00 | AAPL | 157.49080237694724 |
| 2023-01-02 00:00:00 | AAPL | 169.0142861281983 |
| 2023-01-03 00:00:00 | AAPL | 164.6398788362281 |
| 2023-01-04 00:00:00 | AAPL | 161.97316968394074 |
| 2023-01-05 00:00:00 | AAPL | 153.12037280884874 |
| 2023-01-06 00:00:00 | AAPL | 153.11989040672404 |
| 2023-01-07 00:00:00 | AAPL | 151.161672243364 |
| 2023-01-08 00:00:00 | AAPL | 167.3235229154987 |
| 2023-01-09 00:00:00 | AAPL | 162.0223002348642 |
| 2023-01-10 00:00:00 | AAPL | 164.1614515559209 |
| 2023-01-11 00:00:00 | AAPL | 150.41168988591605 |
| 2023-01-12 00:00:00 | AAPL | 169.3981970432399 |
| 2023-01-13 00:00:00 | AAPL | 166.64885281600843 |
| 2023-01-14 00:00:00 | AAPL | 154.24678221356552 |
| 2023-01-15 00:00:00 | AAPL | 153.636499344142 |
| 2023-01-16 00:00:00 | AAPL | 153.66809019706866 |
| 2023-01-17 00:00:00 | AAPL | 156.08484485919075 |
| 2023-01-18 00:00:00 | AAPL | 160.49512863264476 |
| 2023-01-19 00:00:00 | AAPL | 158.63890037284233 |
| 2023-01-20 00:00:00 | AAPL | 155.82458280396085 |
| 2023-01-21 00:00:00 | AAPL | 162.2370578944476 |
| 2023-01-22 00:00:00 | AAPL | 152.78987721304082 |
| 2023-01-23 00:00:00 | AAPL | 155.84289297070436 |
| 2023-01-24 00:00:00 | AAPL | 157.32723686587383 |
| 2023-01-25 00:00:00 | AAPL | 159.12139968434073 |
| 2023-01-26 00:00:00 | AAPL | 165.70351922786028 |
| 2023-01-27 00:00:00 | AAPL | 153.9934756431672 |
| 2023-01-28 00:00:00 | AAPL | 160.28468876827225 |
| 2023-01-29 00:00:00 | AAPL | 161.84829137724086 |
| 2023-01-30 00:00:00 | AAPL | 150.92900825439995 |

Rich Tkinter GUI for Stock Data

Original Data Unpivoted Data Summary Graphs

| Stock | Average Price | Max Price | Min Price |
|-------|--------------------|--------------------|--------------------|
| AAPL | 159.88319115368597 | 169.9943534657226 | 150.00023269510731 |
| GOOG | 2900.905975336107 | 2999.984965366635 | 2800.0315489156114 |
| AMZN | 3400.0100789157955 | 3499.9801954018462 | 3300.0096247788624 |
| MSFT | 259.9742676605365 | 269.99578743625193 | 250.0001107351476 |
| TSLA | 699.3727611087585 | 799.9944294735965 | 600.003347251587 |
| META | 325.1572294578585 | 349.9969848022859 | 300.00042161056524 |



EXAMPLE 5.5

Stacking Dataframe with Synthetic Heart Disease Dataset

The purpose of this project is to generate, transform, and save a synthetic dataset related to heart disease. Initially, the code creates a synthetic dataset with 1,000 records containing various attributes such as patient ID, age, gender, cholesterol level, blood pressure, heart disease status, and diagnosis date. This dataset mimics real-world medical data, providing a foundation for analysis and testing. The `np.random.seed(42)` ensures that the random data generation is reproducible, so the same dataset can be generated each time the code runs.

The code then transforms the dataset by "stacking" it, which involves converting certain columns (cholesterol level and blood pressure) from a wide format into a long format. This transformation simplifies data management and analysis by

consolidating related measurements into a single column, thus making it easier to work with the data in various analytical contexts. Finally, the code saves both the original and transformed datasets into separate sheets of an Excel file. This allows users to access and examine both the raw and modified versions of the dataset conveniently.

```
import pandas as pd
import numpy as np

# Seed for reproducibility
np.random.seed(42)

# Step 1: Generate a Synthetic Heart Disease
Dataset
num_records = 1000

# Generate synthetic data
data = {
    'Patient_ID': np.arange(1, num_records + 1),
    'Age': np.random.randint(30, 80,
size=num_records),
    'Gender': np.random.choice(['Male', 'Female'],
size=num_records),
    'Cholesterol_Level': np.random.randint(150,
300, size=num_records),
    'Blood_Pressure': np.random.randint(90, 180,
size=num_records),
    'Heart_Disease': np.random.choice(['No',
'Yes'], size=num_records),
    'Diagnosis_Date': pd.date_range(start='2022-
01-01', periods=num_records, freq='D')
}

# Create DataFrame
```

```
df = pd.DataFrame(data)

# Step 2: Stack the DataFrame
# To stack the DataFrame, we will convert some
# columns into a more compact format
# Here, we'll stack 'Cholesterol_Level' and
# 'Blood_Pressure' into a long format

# Create a DataFrame for stacking
# 'Cholesterol_Level' and 'Blood_Pressure'
df_stacked = pd.melt(df, id_vars=['Patient_ID',
    'Age', 'Gender', 'Heart_Disease',
    'Diagnosis_Date'],
    value_vars=
    ['Cholesterol_Level', 'Blood_Pressure'],
    var_name='Metric',
    value_name='Value')

# Step 3: Save the DataFrame to Excel
excel_file = 'heart_disease_dataset.xlsx'

# Save both the original and stacked DataFrames to
# separate sheets
with pd.ExcelWriter(excel_file) as writer:
    df.to_excel(writer,
    sheet_name='Original_Data', index=False)
    df_stacked.to_excel(writer,
    sheet_name='Stacked_Data', index=False)

print(f"Data saved to '{excel_file}'")
```

Here's the explanation of each part of the code:

1. Importing Libraries

```
import pandas as pd  
import numpy as np
```

- import pandas as pd: Imports the pandas library with the alias pd. pandas is used for data manipulation and analysis. It provides data structures like DataFrames for handling structured data.
- import numpy as np: Imports the numpy library with the alias np. numpy is used for numerical operations and array handling. It provides functions for generating random numbers and performing mathematical operations.

2. Seed for Reproducibility

```
np.random.seed(42)
```

- np.random.seed(42): Sets the seed for the random number generator in numpy. By setting a specific seed value (42 in this case), the random number generation becomes deterministic. This ensures that the random numbers (and thus the synthetic dataset) generated are the same each time the code is run, making the results reproducible.

3. Generating Synthetic Heart Disease Dataset

```
num_records = 1000
```

- num_records = 1000: Defines the number of records (rows) to generate in the synthetic dataset. This variable determines the size of the dataset.

3. Generating Synthetic Data

```
data = {
    'Patient_ID': np.arange(1, num_records + 1),
    'Age': np.random.randint(30, 80,
size=num_records),
    'Gender': np.random.choice(['Male', 'Female'],
size=num_records),
    'Cholesterol_Level': np.random.randint(150,
300, size=num_records),
    'Blood_Pressure': np.random.randint(90, 180,
size=num_records),
    'Heart_Disease': np.random.choice(['No',
'Yes'], size=num_records),
    'Diagnosis_Date': pd.date_range(start='2022-
01-01', periods=num_records, freq='D')
}
```

- data = {}: Creates a dictionary where each key represents a column in the dataset, and the corresponding value is an array or series of data for that column.
 - 'Patient_ID': np.arange(1, num_records + 1): Generates a sequence of integers from 1 to 1000. This will be used as unique patient identifiers.
 - 'Age': np.random.randint(30, 80, size=num_records): Generates 1,000 random integers between 30 and 80, representing the ages of the patients.
 - 'Gender': np.random.choice(['Male', 'Female'], size=num_records): Randomly selects between 'Male' and 'Female' for each patient, with 1,000 selections.
 - 'Cholesterol_Level': np.random.randint(150, 300, size=num_records): Generates 1,000 random integers between 150 and 300, representing cholesterol levels.

- 'Blood_Pressure': np.random.randint(90, 180, size=num_records): Generates 1,000 random integers between 90 and 180, representing blood pressure measurements.
- 'Heart_Disease': np.random.choice(['No', 'Yes'], size=num_records): Randomly selects between 'No' and 'Yes' for whether the patient has heart disease.
- 'Diagnosis_Date': pd.date_range(start='2022-01-01', periods=num_records, freq='D'): Creates a range of dates starting from January 1, 2022, for 1,000 days, representing the diagnosis dates.

Creating DataFrame

```
df = pd.DataFrame(data)
```

- df = pd.DataFrame(data): Converts the data dictionary into a pandas DataFrame. Each key in the dictionary becomes a column in the DataFrame, and the values become the rows for that column.

4. Stacking the DataFrame

```
df_stacked = pd.melt(df, id_vars=['Patient_ID',  
'Age', 'Gender', 'Heart_Disease',  
'Diagnosis_Date'],  
value_vars=['Cholesterol_Level', 'Blood_Pressure'],  
var_name='Metric',  
value_name='Value')
```

- pd.melt(): Transforms the DataFrame from a wide format (where each variable is in its own column) to a long format (where multiple variables are stacked into a single column).
 - id_vars=['Patient_ID', 'Age', 'Gender', 'Heart_Disease', 'Diagnosis_Date']: Specifies the columns that should remain as identifier variables in the long format. These columns will be repeated for each value in the stacked columns.
 - value_vars=['Cholesterol_Level', 'Blood_Pressure']: Specifies the columns to be stacked into a long format. These columns will be combined into a single 'Metric' column, with their values appearing in the 'Value' column.
 - var_name='Metric': Defines the name of the new column that will contain the names of the original columns being stacked.
 - value_name='Value': Defines the name of the new column that will contain the values from the stacked columns.

5. Saving the DataFrame to Excel

```
excel_file = 'heart_disease_dataset.xlsx'
```

- excel_file = 'heart_disease_dataset.xlsx': Specifies the name of the Excel file to save the DataFrames.

Saving DataFrames to Excel

```
with pd.ExcelWriter(excel_file) as writer:
```

```
df.to_excel(writer,  
sheet_name='Original_Data', index=False)  
df_stacked.to_excel(writer,  
sheet_name='Stacked_Data', index=False)
```

- with pd.ExcelWriter(excel_file) as writer: Opens an Excel writer object, allowing multiple DataFrames to be saved into different sheets of the same Excel file.
 - df.to_excel(writer, sheet_name='Original_Data', index=False): Saves the original DataFrame to the sheet named 'Original_Data'. index=False prevents saving the DataFrame index as a separate column.
 - df_stacked.to_excel(writer, sheet_name='Stacked_Data', index=False): Saves the stacked DataFrame to the sheet named 'Stacked_Data'.

6. Confirmation Message

```
print(f"Data saved to '{excel_file}'.")
```

- print(f"Data saved to '{excel_file}' . "): Prints a message confirming that the data has been saved to the specified Excel file. The file name is dynamically inserted into the message using an f-string.

EXAMPLE 5.6

GUI Tkinter for Stacking Dataframe with Synthetic Heart Disease Dataset

This project is designed to create an interactive graphical user interface (GUI) for analyzing a synthetic heart disease dataset using

Tkinter and matplotlib. The purpose of this project is to provide a comprehensive tool for exploring and visualizing health data through various tabs and graphical representations. The application is built with Tkinter, a popular Python library for GUI development, and incorporates pandas for data manipulation, numpy for generating synthetic data, and matplotlib for visualizations.

The core functionality of the application includes several tabs that facilitate different types of data exploration. The "Original Data" tab displays the dataset in a tabular format, allowing users to view the raw data. The "Stacked Data" tab presents the data in a long format, which is useful for certain types of analyses. The "Summary" tab provides key statistical summaries of the dataset, such as average, maximum, and minimum values for selected metrics. These summaries help in understanding the central tendencies and spread of the data.

Additionally, the "Graphs" tab uses matplotlib to generate and display visualizations, including histograms for age distribution, cholesterol levels, blood pressure, and heart disease status. These graphical representations help users to quickly grasp the distributions and patterns within the dataset. Overall, this project aims to create a rich and interactive environment for analyzing and visualizing health data, making it easier for users to derive insights and make informed decisions based on the provided data.

```
import tkinter as tk
from tkinter import ttk
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

# Seed for reproducibility
np.random.seed(42)

# Generate synthetic heart disease dataset
num_records = 1000
data = {
    'Patient_ID': np.arange(1, num_records + 1),
    'Age': np.random.randint(30, 80,
size=num_records),
    'Gender': np.random.choice(['Male', 'Female'],
size=num_records),
    'Cholesterol_Level': np.random.randint(150,
300, size=num_records),
    'Blood_Pressure': np.random.randint(90, 180,
size=num_records),
    'Heart_Disease': np.random.choice(['No',
'Yes'], size=num_records),
    'Diagnosis_Date': pd.date_range(start='2022-
01-01', periods=num_records, freq='D')
}
df = pd.DataFrame(data)
df_stacked = pd.melt(df, id_vars=['Patient_ID',
'Age', 'Gender', 'Heart_Disease',
'Diagnosis_Date'],
value_vars=
['Cholesterol_Level', 'Blood_Pressure'],
var_name='Metric',
value_name='Value')

class HeartDiseaseApp:
    def __init__(self, root):
        self.root = root
```

```
        self.root.title("Heart Disease Data
Analysis")
        self.root.geometry("1200x800")

        # Notebook (Tabs)
        self.notebook = ttk.Notebook(root)
        self.notebook.pack(expand=True,
fill='both')

        # Create Tabs
        self.create_original_data_tab()
        self.create_stacked_data_tab()
        self.create_summary_tab()
        self.create_graphs_tab()

    def create_original_data_tab(self):
        frame = ttk.Frame(self.notebook)
        self.notebook.add(frame, text='Original
Data')

        # Table
        self.original_table = ttk.Treeview(frame,
columns=list(df.columns), show='headings')
        self.original_table.pack(expand=True,
fill='both')

        # Scrollbars
        vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.original_table.yview)
        vsb.pack(side='right', fill='y')

        self.original_table.configure(yscrollcommand=vsb.
set)
```

```
        hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.original_table.xview)
        hsb.pack(side='bottom', fill='x')

    self.original_table.configure(xscrollcommand=hsb.
set)

        # Define columns
        for col in df.columns:
            self.original_table.heading(col,
text=col)
            self.original_table.column(col,
width=150)

    self.update_original_data_table()

def create_stacked_data_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Stacked
Data')

    # Table
    self.stacked_table = ttk.Treeview(frame,
columns=list(df_stacked.columns), show='headings')
    self.stacked_table.pack(expand=True,
fill='both')

    # Scrollbars
    vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.stacked_table.yview)
    vsb.pack(side='right', fill='y')

    self.stacked_table.configure(yscrollcommand=vsb.s
et)
```

```
        hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.stacked_table.xview)
        hsb.pack(side='bottom', fill='x')

    self.stacked_table.configure(xscrollcommand=hsb.set)

        # Define columns
        for col in df_stacked.columns:
            self.stacked_table.heading(col,
text=col)
            self.stacked_table.column(col,
width=150)

    self.update_stacked_data_table()

def create_summary_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Summary')

        # Table
        self.summary_table = ttk.Treeview(frame,
columns=['Metric', 'Average', 'Max', 'Min'],
show='headings')
        self.summary_table.pack(expand=True,
fill='both')

        # Scrollbars
        vsb = ttk.Scrollbar(frame,
orient="vertical",
command=self.summary_table.yview)
        vsb.pack(side='right', fill='y')

    self.summary_table.configure(yscrollcommand=vsb.s
```

```
et)

        hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.summary_table.xview)
        hsb.pack(side='bottom', fill='x')

    self.summary_table.configure(xscrollcommand=hsb.s
et)

        # Define columns
        self.summary_table.heading('Metric',
text='Metric')
        self.summary_table.heading('Average',
text='Average')
        self.summary_table.heading('Max',
text='Max')
        self.summary_table.heading('Min',
text='Min')
        self.summary_table.column('Metric',
width=150)
        self.summary_table.column('Average',
width=150)
        self.summary_table.column('Max',
width=150)
        self.summary_table.column('Min',
width=150)

    self.update_summary_table()

def create_graphs_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Graphs')

    # Create and display figures
    self.plot_distributions(frame)
```

```
def plot_distributions(self, frame):
    fig, axs = plt.subplots(2, 2, figsize=(14, 10))

        # Distribution of Age
        axs[0, 0].hist(df['Age'], bins=20,
color='skyblue', edgecolor='black')
        axs[0, 0].set_title('Age Distribution')
        axs[0, 0].set_xlabel('Age')
        axs[0, 0].set_ylabel('Frequency')

        # Distribution of Cholesterol Level
        axs[0, 1].hist(df['Cholesterol_Level'],
bins=20, color='lightgreen', edgecolor='black')
        axs[0, 1].set_title('Cholesterol Level
Distribution')
        axs[0, 1].set_xlabel('Cholesterol Level')
        axs[0, 1].set_ylabel('Frequency')

        # Distribution of Blood Pressure
        axs[1, 0].hist(df['Blood_Pressure'],
bins=20, color='salmon', edgecolor='black')
        axs[1, 0].set_title('Blood Pressure
Distribution')
        axs[1, 0].set_xlabel('Blood Pressure')
        axs[1, 0].set_ylabel('Frequency')

        # Distribution of Heart Disease
        heart_disease_counts =
df['Heart_Disease'].value_counts()
        axs[1, 1].bar(heart_disease_counts.index,
heart_disease_counts.values, color='lightcoral')
        axs[1, 1].set_title('Heart Disease
Distribution')
        axs[1, 1].set_xlabel('Heart Disease')
```

```

        axs[1, 1].set_ylabel('Count')

        # Display figure in Tkinter
        canvas = FigureCanvasTkAgg(fig,
master=frame)
        canvas.draw()
        canvas.get_tk_widget().pack(expand=True,
fill='both')

    def update_original_data_table(self):
        # Clear current data
        for row in
self.original_table.get_children():
            self.original_table.delete(row)

        # Insert new data with alternating row
colors
        for i, row in
enumerate(df.itertuples(index=False)):
            item_id =
self.original_table.insert('', 'end', values=row)
            if i % 2 == 0:

                self.original_table.tag_configure('evenrow',
background='#f0f0f0')
                    self.original_table.item(item_id,
tags='evenrow')
            else:

                self.original_table.tag_configure('oddrow',
background='#ffffff')
                    self.original_table.item(item_id,
tags='oddrow' )

    def update_stacked_data_table(self):
        # Clear current data

```

```

        for row in
self.stacked_table.get_children():
            self.stacked_table.delete(row)

            # Insert new data with alternating row
colors
        for i, row in
enumerate(df_stacked.itertuples(index=False)):
            item_id =
self.stacked_table.insert(' ', 'end', values=row)
            if i % 2 == 0:

                self.stacked_table.tag_configure('evenrow',
background='#f0f0f0')
                    self.stacked_table.item(item_id,
tags='evenrow')
                else:

                    self.stacked_table.tag_configure('oddrow',
background='#ffffff')
                    self.stacked_table.item(item_id,
tags='oddrow')

    def update_summary_table(self):
        # Clear current data
        for row in
self.summary_table.get_children():
            self.summary_table.delete(row)

        # Calculate and display summary statistics
        summary_data = {
            'Metric': ['Age', 'Cholesterol Level',
'Blood Pressure'],
            'Average': [df['Age'].mean(),
df['Cholesterol_Level'].mean(),
df['Blood_Pressure'].mean()],

```

```

        'Max': [df['Age'].max(),
df['Cholesterol_Level'].max(),
df['Blood_Pressure'].max()],
        'Min': [df['Age'].min(),
df['Cholesterol_Level'].min(),
df['Blood_Pressure'].min()]
    }
summary_df = pd.DataFrame(summary_data)
for row in
summary_df.itertuples(index=False):
    self.summary_table.insert('', 'end',
values=row)

if __name__ == "__main__":
    root = tk.Tk()
    app = HeartDiseaseApp(root)
    root.mainloop()

```

Here's the explanation of each part of the provided code, which creates a rich Tkinter GUI for analyzing a synthetic heart disease dataset:

Imports and Data Generation

```

import tkinter as tk
from tkinter import ttk
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from    matplotlib.backends.backend_tkagg    import
FigureCanvasTkAgg

```

- **tkinter and ttk:** These are standard libraries for creating GUIs in Python. tkinter provides the basic GUI

components, while ttk offers more advanced and themed widgets.

- pandas: A powerful data manipulation library used for handling data in DataFrames.
- numpy: Used for numerical operations, such as generating random numbers.
- matplotlib: A plotting library used for creating visualizations. FigureCanvasTkAgg allows matplotlib figures to be embedded into Tkinter GUIs.

```
np.random.seed(42)
```

- np.random.seed(42): Sets the seed for random number generation to ensure reproducibility of the synthetic dataset.

```
num_records = 1000
data = {
    'Patient_ID': np.arange(1, num_records + 1),
    'Age': np.random.randint(30, 80,
size=num_records),
    'Gender': np.random.choice(['Male', 'Female'],
size=num_records),
    'Cholesterol_Level': np.random.randint(150,
300, size=num_records),
    'Blood_Pressure': np.random.randint(90, 180,
size=num_records),
    'Heart_Disease': np.random.choice(['No',
'Yes'], size=num_records),
    'Diagnosis_Date': pd.date_range(start='2022-
01-01', periods=num_records, freq='D')
}
df = pd.DataFrame(data)
```

```
df_stacked = pd.melt(df, id_vars=['Patient_ID',  
'Age', 'Gender', 'Heart_Disease',  
'Diagnosis_Date'],  
                      value_vars=  
['Cholesterol_Level', 'Blood_Pressure'],  
                      var_name='Metric',  
value_name='Value')
```

- num_records = 1000: Specifies the number of records in the synthetic dataset.
- data: Dictionary containing columns for the dataset:
 - Patient_ID: Unique identifiers for each record.
 - Age: Random ages between 30 and 80.
 - Gender: Randomly assigned gender (Male or Female).
 - Cholesterol_Level: Random cholesterol levels between 150 and 300.
 - Blood_Pressure: Random blood pressure readings between 90 and 180.
 - Heart_Disease: Randomly assigned (Yes or No).
 - Diagnosis_Date: A range of dates starting from January 1, 2022, with a daily frequency.
- df: A pandas DataFrame created from the data dictionary.
- df_stacked: A melted version of df where Cholesterol_Level and Blood_Pressure are combined into a long format for easier analysis.

GUI Class Definition

```
class HeartDiseaseApp:  
    def __init__(self, root):  
        self.root = root
```

```
        self.root.title("Heart Disease Data Analysis")
        self.root.geometry("1200x800")

        self.notebook = ttk.Notebook(root)
                    self.notebook.pack(expand=True,
fill='both')

        self.create_original_data_tab()
        self.create_stacked_data_tab()
        self.create_summary_tab()
        self.create_graphs_tab()
```

- HeartDiseaseApp: Main class for the application. It initializes the GUI and creates tabs.
- init(self, root): Constructor method initializes the root window, sets its title and size, and creates a ttk.Notebook widget for tabbed navigation.

Creating Tabs

```
def create_original_data_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Original Data')

    self.original_table = ttk.Treeview(frame,
columns=list(df.columns), show='headings')
    self.original_table.pack(expand=True,
fill='both')

    vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.original_table.yview)
    vsb.pack(side='right', fill='y')
```

```

self.original_table.configure(yscrollcommand=vsb.set)

        hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.original_table.xview)
    hsb.pack(side='bottom', fill='x')

self.original_table.configure(xscrollcommand=hsb.set)

for col in df.columns:
    self.original_table.heading(col, text=col)
    self.original_table.column(col, width=150)

self.update_original_data_table()

```

- `create_original_data_tab(self)`: Sets up a tab displaying the original dataset.
- `frame`: A frame widget that holds the table and scrollbars.
- `self.original_table`: A Treeview widget that displays the dataset in a table format.
- `vsb` and `hsb`: Vertical and horizontal scrollbars for navigating through the table.
- `for col in df.columns`: Configures each column of the table with headings and widths.
- `self.update_original_data_table()`: Method to populate the table with data.

```

def create_stacked_data_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Stacked Data')

```

```

        self.stacked_table = ttk.Treeview(frame,
columns=list(df_stacked.columns), show='headings')
        self.stacked_table.pack(expand=True,
fill='both')

        vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.stacked_table.yview)
        vsb.pack(side='right', fill='y')

self.stacked_table.configure(yscrollcommand=vsb.set)

        hsb      =      ttk.Scrollbar(frame,
orient="horizontal",
command=self.stacked_table.xview)
        hsb.pack(side='bottom', fill='x')

self.stacked_table.configure(xscrollcommand=hsb.set)

for col in df_stacked.columns:
    self.stacked_table.heading(col, text=col)
    self.stacked_table.column(col, width=150)

self.update_stacked_data_table()

```

- `create_stacked_data_tab(self)`: Sets up a tab displaying the stacked version of the dataset.
- `self.stacked_table`: Another Treeview widget, but for the stacked data.
- `vsb` and `hsb`: Scrollbars similar to those in the original data tab.
- `for col in df_stacked.columns`: Configures the table columns for the stacked data.

- `self.update_stacked_data_table()`: Method to populate the stacked data table.

```

def create_summary_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Summary')

        self.summary_table = ttk.Treeview(frame,
columns=['Metric', 'Average', 'Max', 'Min'],
show='headings')
        self.summary_table.pack(expand=True,
fill='both')

    vsb = ttk.Scrollbar(frame, orient="vertical",
command=self.summary_table.yview)
    vsb.pack(side='right', fill='y')

self.summary_table.configure(yscrollcommand=vsb.set)

    hsb = ttk.Scrollbar(frame,
orient="horizontal",
command=self.summary_table.xview)
    hsb.pack(side='bottom', fill='x')

self.summary_table.configure(xscrollcommand=hsb.set)

        self.summary_table.heading('Metric',
text='Metric')
        self.summary_table.heading('Average',
text='Average')
        self.summary_table.heading('Max', text='Max')
        self.summary_table.heading('Min', text='Min')
        self.summary_table.column('Metric', width=150)

```

```

        self.summary_table.column('Average',
width=150)
        self.summary_table.column('Max', width=150)
self.summary_table.column('Min', width=150)

self.update_summary_table()

```

- `create_summary_tab(self)`: Sets up a tab displaying summary statistics of the dataset.
- `self.summary_table`: A Treeview widget for showing summary statistics.
- `vsb` and `hsb`: Scrollbars for the summary table.
- `self.summary_table.heading`: Configures column headings and widths.
- `self.update_summary_table()`: Method to populate the summary table with calculated statistics.

```

def create_graphs_tab(self):
    frame = ttk.Frame(self.notebook)
    self.notebook.add(frame, text='Graphs')

    self.plot_distributions(frame)
create_graphs_tab(self): Sets up a tab for
displaying graphical distributions of the data.
self.plot_distributions(frame): Calls a method to
generate and display graphs.

python
Copy code
def plot_distributions(self, frame):
    fig, axs = plt.subplots(2, 2, figsize=(14,
10))

    axs[0, 0].hist(df['Age'], bins=20,
color='skyblue', edgecolor='black')

```

```

        axs[0, 0].set_title('Age Distribution')
        axs[0, 0].set_xlabel('Age')
        axs[0, 0].set_ylabel('Frequency')

            axs[0, 1].hist(df['Cholesterol_Level'],
bins=20, color='lightgreen', edgecolor='black')
            axs[0, 1].set_title('Cholesterol Level
Distribution')
            axs[0, 1].set_xlabel('Cholesterol Level')
            axs[0, 1].set_ylabel('Frequency')

            axs[1, 0].hist(df['Blood_Pressure'], bins=20,
color='salmon', edgecolor='black')
            axs[1, 0].set_title('Blood Pressure
Distribution')
            axs[1, 0].set_xlabel('Blood Pressure')
            axs[1, 0].set_ylabel('Frequency')

                heart_disease_counts      =
df['Heart_Disease'].value_counts()
                axs[1, 1].bar(heart_disease_counts.index,
heart_disease_counts.values, color='lightcoral')
                axs[1, 1].set_title('Heart Disease
Distribution')
                axs[1, 1].set_xlabel('Heart Disease')
                axs[1, 1].set_ylabel('Count')

canvas = FigureCanvasTkAgg(fig, master=frame)
canvas.draw()
    canvas.get_tk_widget().pack(expand=True,
fill='both')

```

- `plot_distributions(self, frame)`: Generates and displays four different plots:
- `fig, axs`: Create a 2x2 grid of subplots.

- axs[0, 0]: Histogram of Age distribution.
- axs[0, 1]: Histogram of Cholesterol_Level distribution.
- axs[1, 0]: Histogram of Blood_Pressure distribution.
- axs[1, 1]: Bar chart of Heart_Disease distribution.
- FigureCanvasTkAgg: Embeds the matplotlib figure into the Tkinter GUI.
- canvas.draw(): Renders the figure.
- canvas.get_tk_widget().pack(expand=True, fill='both'): Places the canvas widget into the Tkinter frame.

Updating Tables

```
def update_original_data_table(self):
    for row in self.original_table.get_children():
        self.original_table.delete(row)

    for i, row in enumerate(df.itertuples(index=False)):
        item_id = self.original_table.insert('', 'end', values=row)
        if i % 2 == 0:
            self.original_table.tag_configure('evenrow',
                                              background='#f0f0f0')
            self.original_table.item(item_id,
                                     tags='evenrow')
        else:
            self.original_table.tag_configure('oddrow',
                                              background='#ffffff')
            self.original_table.item(item_id,
                                     tags='oddrow')
```

- `update_original_data_table(self)`: Updates the original data table with new data.
- `self.original_table.get_children()`: Retrieves all rows currently in the table.
- `self.original_table.delete(row)`: Deletes each row.
- `for i, row in enumerate(df.itertuples(index=False))`: Inserts each row from df into the table.
- `self.original_table.tag_configure`: Configures alternating row colors for better readability.

```
def update_stacked_data_table(self):
    for row in self.stacked_table.get_children():
        self.stacked_table.delete(row)

                for i, row in
enumerate(df_stacked.itertuples(index=False)):
                    item_id = self.stacked_table.insert('', 'end', values=row)
                    if i % 2 == 0:

self.stacked_table.tag_configure('evenrow',
background='#f0f0f0')
                    self.stacked_table.item(item_id,
tags='evenrow')
                    else:

self.stacked_table.tag_configure('oddrow',
background='#ffffff')
                    self.stacked_table.item(item_id,
tags='oddrow' )
```

- `update_stacked_data_table(self)`: Updates the stacked data table similarly to the original data table.

```

def update_summary_table(self):
    for row in self.summary_table.get_children():
        self.summary_table.delete(row)

    summary_data = {
        'Metric': ['Age', 'Cholesterol Level',
'Blood Pressure'],
                    'Average': [df['Age'].mean(),
df['Cholesterol_Level'].mean(),
df['Blood_Pressure'].mean()],
                    'Max': [df['Age'].max(),
df['Cholesterol_Level'].max(),
df['Blood_Pressure'].max()],
                    'Min': [df['Age'].min(),
df['Cholesterol_Level'].min(),
df['Blood_Pressure'].min()]
    }
    summary_df = pd.DataFrame(summary_data)
    for row in summary_df.itertuples(index=False):
        self.summary_table.insert('', 'end',
values=row)

```

- `update_summary_table(self)`: Updates the summary table with calculated statistics.
- `summary_data`: Dictionary containing metrics and their corresponding statistics (Average, Max, Min).
- `summary_df`: DataFrame created from `summary_data`.
- `for row in summary_df.itertuples(index=False)`: Inserts summary statistics into the table.

Main Execution

```

if __name__ == "__main__":

```

```

root = tk.Tk()
app = HeartDiseaseApp(root)
root.mainloop()

```

- if name == "main": Ensures this block runs only if the script is executed directly.
- root = tk.Tk(): Creates the main Tkinter window.
- app = HeartDiseaseApp(root): Initializes the HeartDiseaseApp with the main window.
- root.mainloop(): Starts the Tkinter event loop, waiting for user interaction.

Heart Disease Data Analysis

| | Patient_ID | Age | Gender | Cholesterol_Level | Blood_Pressure | Heart_Disease | Diagnosis_Date |
|----|------------|--------|--------|-------------------|----------------|---------------------|----------------|
| 1 | 68 | Male | 188 | 162 | No | 2022-01-01 00:00:00 | |
| 2 | 58 | Male | 231 | 134 | Yes | 2022-01-02 00:00:00 | |
| 3 | 44 | Male | 249 | 145 | Yes | 2022-01-03 00:00:00 | |
| 4 | 72 | Female | 151 | 173 | No | 2022-01-04 00:00:00 | |
| 5 | 37 | Male | 226 | 171 | No | 2022-01-05 00:00:00 | |
| 6 | 50 | Female | 161 | 146 | No | 2022-01-06 00:00:00 | |
| 7 | 68 | Female | 188 | 110 | No | 2022-01-07 00:00:00 | |
| 8 | 48 | Male | 162 | 157 | Yes | 2022-01-08 00:00:00 | |
| 9 | 52 | Female | 195 | 139 | No | 2022-01-09 00:00:00 | |
| 10 | 40 | Female | 262 | 122 | Yes | 2022-01-10 00:00:00 | |
| 11 | 40 | Female | 166 | 96 | No | 2022-01-11 00:00:00 | |
| 12 | 53 | Male | 253 | 159 | Yes | 2022-01-12 00:00:00 | |
| 13 | 65 | Female | 247 | 104 | Yes | 2022-01-13 00:00:00 | |
| 14 | 69 | Female | 237 | 171 | Yes | 2022-01-14 00:00:00 | |
| 15 | 53 | Female | 258 | 121 | Yes | 2022-01-15 00:00:00 | |
| 16 | 32 | Male | 219 | 164 | No | 2022-01-16 00:00:00 | |
| 17 | 51 | Female | 223 | 109 | No | 2022-01-17 00:00:00 | |
| 18 | 31 | Male | 212 | 177 | Yes | 2022-01-18 00:00:00 | |
| 19 | 53 | Male | 209 | 104 | No | 2022-01-19 00:00:00 | |
| 20 | 73 | Male | 228 | 137 | No | 2022-01-20 00:00:00 | |
| 21 | 59 | Male | 186 | 160 | Yes | 2022-01-21 00:00:00 | |
| 22 | 67 | Male | 171 | 110 | No | 2022-01-22 00:00:00 | |
| 23 | 31 | Male | 297 | 131 | No | 2022-01-23 00:00:00 | |
| 24 | 50 | Female | 245 | 115 | No | 2022-01-24 00:00:00 | |
| 25 | 62 | Male | 217 | 103 | Yes | 2022-01-25 00:00:00 | |
| 26 | 41 | Female | 166 | 146 | No | 2022-01-26 00:00:00 | |
| 27 | 51 | Female | 161 | 126 | Yes | 2022-01-27 00:00:00 | |
| 28 | 73 | Male | 299 | 147 | No | 2022-01-28 00:00:00 | |
| 29 | 54 | Female | 294 | 110 | No | 2022-01-29 00:00:00 | |
| 30 | 78 | Female | 287 | 171 | No | 2022-01-30 00:00:00 | |

Heart Disease Data Analysis

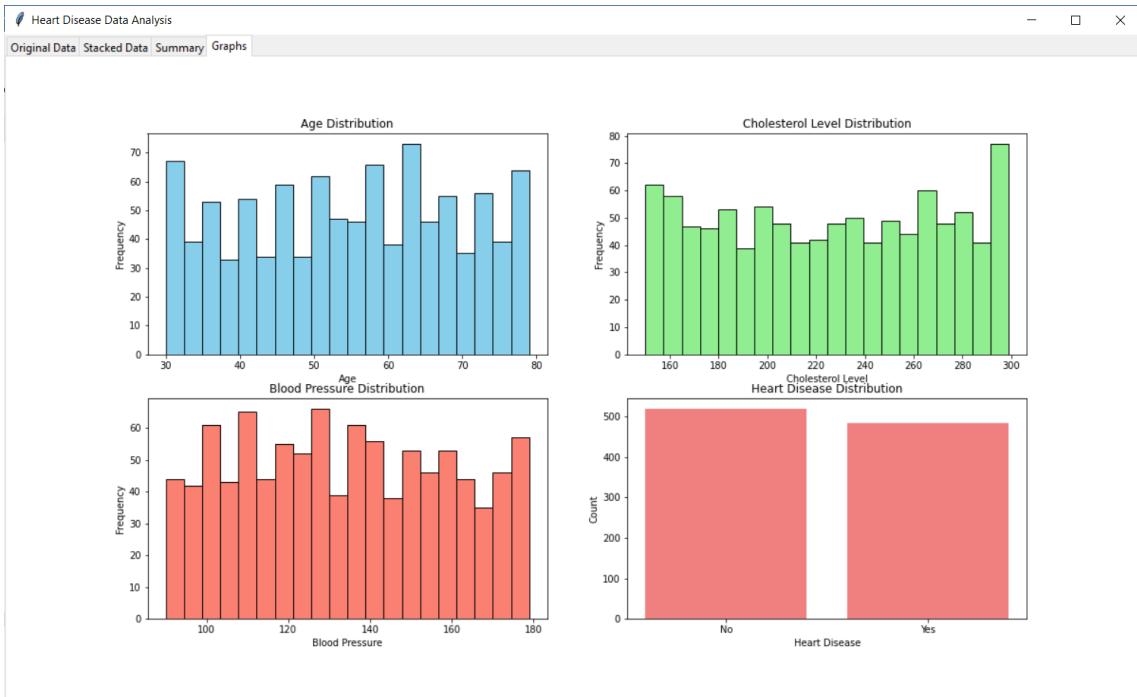
Original Data Stacked Data Summary Graphs

| | Patient_ID | Age | Gender | Heart_Disease | Diagnosis_Date | Metric | Value |
|----|------------|--------|--------|---------------------|-------------------|--------|-------|
| 1 | 68 | Male | No | 2022-01-01 00:00:00 | Cholesterol_Level | 188 | |
| 2 | 58 | Male | Yes | 2022-01-02 00:00:00 | Cholesterol_Level | 231 | |
| 3 | 44 | Male | Yes | 2022-01-03 00:00:00 | Cholesterol_Level | 249 | |
| 4 | 72 | Female | No | 2022-01-04 00:00:00 | Cholesterol_Level | 151 | |
| 5 | 37 | Male | No | 2022-01-05 00:00:00 | Cholesterol_Level | 226 | |
| 6 | 50 | Female | No | 2022-01-06 00:00:00 | Cholesterol_Level | 161 | |
| 7 | 68 | Female | No | 2022-01-07 00:00:00 | Cholesterol_Level | 188 | |
| 8 | 48 | Male | Yes | 2022-01-08 00:00:00 | Cholesterol_Level | 162 | |
| 9 | 52 | Female | No | 2022-01-09 00:00:00 | Cholesterol_Level | 195 | |
| 10 | 40 | Female | Yes | 2022-01-10 00:00:00 | Cholesterol_Level | 262 | |
| 11 | 40 | Female | No | 2022-01-11 00:00:00 | Cholesterol_Level | 166 | |
| 12 | 53 | Male | Yes | 2022-01-12 00:00:00 | Cholesterol_Level | 253 | |
| 13 | 65 | Female | Yes | 2022-01-13 00:00:00 | Cholesterol_Level | 247 | |
| 14 | 69 | Female | Yes | 2022-01-14 00:00:00 | Cholesterol_Level | 237 | |
| 15 | 53 | Female | Yes | 2022-01-15 00:00:00 | Cholesterol_Level | 258 | |
| 16 | 32 | Male | No | 2022-01-16 00:00:00 | Cholesterol_Level | 219 | |
| 17 | 51 | Female | No | 2022-01-17 00:00:00 | Cholesterol_Level | 223 | |
| 18 | 31 | Male | Yes | 2022-01-18 00:00:00 | Cholesterol_Level | 212 | |
| 19 | 53 | Male | No | 2022-01-19 00:00:00 | Cholesterol_Level | 209 | |
| 20 | 73 | Male | No | 2022-01-20 00:00:00 | Cholesterol_Level | 228 | |
| 21 | 59 | Male | Yes | 2022-01-21 00:00:00 | Cholesterol_Level | 186 | |
| 22 | 67 | Male | No | 2022-01-22 00:00:00 | Cholesterol_Level | 171 | |
| 23 | 31 | Male | No | 2022-01-23 00:00:00 | Cholesterol_Level | 297 | |
| 24 | 50 | Female | No | 2022-01-24 00:00:00 | Cholesterol_Level | 245 | |
| 25 | 62 | Male | Yes | 2022-01-25 00:00:00 | Cholesterol_Level | 217 | |
| 26 | 41 | Female | No | 2022-01-26 00:00:00 | Cholesterol_Level | 166 | |
| 27 | 51 | Female | Yes | 2022-01-27 00:00:00 | Cholesterol_Level | 161 | |
| 28 | 73 | Male | No | 2022-01-28 00:00:00 | Cholesterol_Level | 299 | |
| 29 | 54 | Female | No | 2022-01-29 00:00:00 | Cholesterol_Level | 294 | |
| 30 | 78 | Female | No | 2022-01-30 00:00:00 | Cholesterol_Level | 287 | |

Heart Disease Data Analysis

Original Data Stacked Data Summary Graphs

| Metric | Average | Max | Min |
|-------------------|---------|-----|-----|
| Age | 54.83 | 79 | 30 |
| Cholesterol Level | 225.233 | 299 | 150 |
| Blood Pressure | 133.829 | 179 | 90 |



HANDLING MISSING DATA

HANDLING MISSING DATA

INTRODUCTION

Handling missing data in a DataFrame is a crucial aspect of data preprocessing. Missing data can occur for various reasons, such as data entry errors, merging issues, or incomplete data collection. Here's a detailed guide on how to handle missing data in a DataFrame using pandas:

1. Identify Missing Data

a. Detecting Missing Data

Nan Representation:

- In pandas, missing values are often represented as NaN (Not a Number), None, or NaT (Not a Time).
- Different types of missing data can be handled differently depending on their representation.

Detect Missing Values:

- `isna()` or `isnull()`:

```
df.isna()    # Returns a DataFrame of the same  
shape with True for NaN and False otherwise  
df.isnull()  # Alias for isna()
```

This method returns a DataFrame of the same shape where each element is True if the corresponding value is NaN and False otherwise.

- Summing Missing Values:

```
df.isna().sum()    # Returns the number of NaN  
values per column
```

This gives the count of missing values for each column. Summing across the DataFrame can provide the total count.

- Checking Total Missing Values:

```
df.isna().sum().sum()    # Total number of NaN  
values in the DataFrame
```

This provides the total number of missing values in the entire DataFrame.

- Count Missing Values by Row:

```
df.isna().sum(axis=1)    # Number of NaN values  
per row
```

- Visualizing Missing Data:

Tools like missingno can visualize missing data patterns.

```
import missingno as msno  
msno.matrix(df) # Visualizes missing data
```

b. Understanding Missing Data Patterns

- Missing Completely at Random (MCAR):
 - Missing data has no relationship with the observed or unobserved data.
 - No bias introduced.
- Missing at Random (MAR):
 - Missing data is related to observed data but not the missing data itself.
 - For example, income data might be missing more for certain age groups.
- Missing Not at Random (MNAR):
 - Missing data is related to the missing values themselves.
 - For example, people with very high income might be less likely to report their income.

2. Handle Missing Data

a. Removing Missing Data

Drop Rows:

- Drop rows with any NaN values:

```
df.dropna() # Drops rows where any NaN values  
are present
```

- Drop rows with NaN values in specific columns:

```
df.dropna(subset=['column_name'])      # Only  
drops rows where NaN exists in 'column_name'
```

- Drop rows with all NaN values:

```
df.dropna(how='all')    # Drops rows where all  
values are NaN
```

Drop Columns:

- Drop columns with any NaN values:

```
df.dropna(axis=1)    # Drops columns where any  
NaN values are present
```

- Drop columns with all NaN values:

```
df.dropna(axis=1, how='all')    # Drops columns  
where all values are NaN
```

b. Filling Missing Data

- Fill with Constant Values:

Fill NaNs with a specific value:

```
df.fillna(value=0)    # Replace NaN with 0
```

- Forward Fill:

Propagate the last valid observation forward:

```
df.fillna(method='ffill')
```

Useful when data is time series and you want to carry forward the last observed value.

- Backward Fill:

Propagate the next valid observation backward:

```
df.fillna(method='bfill')
```

Useful when you want to fill missing values with the next valid observation.

- Interpolation:

Interpolate missing values:

```
df.interpolate()      # Linear interpolation by default
```

This method estimates missing values by interpolating between existing values. You can specify different methods, such as polynomial interpolation.

- Fill with Statistical Measures:

Mean, Median, or Mode:

```
mean_value = df['column_name'].mean()
df['column_name'].fillna(mean_value,
inplace=True)
median_value = df['column_name'].median()
df['column_name'].fillna(median_value,
inplace=True)
mode_value = df['column_name'].mode()[0]
df['column_name'].fillna(mode_value,
inplace=True)
```

- Contextual Filling:

For categorical data, you might fill missing values with the most frequent category.

- Model-Based Imputation:
- Use machine learning algorithms to predict missing values:
 - Train a model to predict missing values based on other features.
 - Libraries like scikit-learn offer tools for this.

3. Checking the Result

After handling missing data, it's essential to check that your methods have been applied correctly and that the data integrity has been maintained.

Re-check Missing Values:

```
df.isna().sum() # Ensure there are no remaining  
NaNs
```

Inspect Data:

```
print(df.head()) # Verify the changes
```

Statistical Summary:

```
print(df.describe()) # Check if the distribution  
of data is reasonable
```

Example

Here's an enhanced example:

```
import pandas as pd  
import numpy as np
```

```

# Create a DataFrame with missing values
data = {
    'A': [1, np.nan, 3, 4, np.nan],
    'B': [5, 6, np.nan, 8, 9],
    'C': [np.nan, np.nan, 11, 12, 13]
}
df = pd.DataFrame(data)

# Identify missing values
print("Missing values in DataFrame:\n", df.isna())

# Drop rows with any missing values
df_dropped = df.dropna()
print("\nDataFrame after dropping rows with
missing values:\n", df_dropped)

# Fill missing values with the mean of each column
mean_values = df.mean()
df_filled_mean = df.fillna(mean_values)
print("\nDataFrame after filling missing values
with mean:\n", df_filled_mean)

# Interpolate missing values
df_interpolated = df.interpolate()
print("\nDataFrame after interpolation:\n",
df_interpolated)

```

Considerations

Impact on Analysis:

1. Handling missing data should not introduce bias or distort the analysis. Choose methods that preserve the integrity of your data.
2. Domain Knowledge:

Sometimes, the best method for handling missing data depends on the context and domain knowledge. Understanding the reasons behind missing data can guide better handling strategies.

3. Check Assumptions:

Be aware of assumptions made when imputing missing values, such as assuming a linear trend in interpolation.

By understanding and applying these techniques, you can effectively handle missing data in your DataFrame and ensure your analysis is robust and reliable.

EXAMPLE 6.1

Handling Missing Data with Dropping Rows Using Road Traffic Dataset

The project generates a large synthetic road traffic dataset, consisting of 10,000 rows, where each row represents a traffic observation with fields such as Date, Time, Location, Vehicle_Count, Average_Speed, and Incident. To achieve this, it uses various randomization techniques: `np.random.seed(42)` ensures reproducibility, `pd.date_range` generates a range of dates with hourly frequency, and lists are created for time, location, vehicle count, speed, and incidents using random selections. The data is then stored in a pandas DataFrame, which serves as a structured and easily manipulable data format.

After creating the dataset, the code simulates real-world scenarios where some data might be missing by randomly introducing NaN values into 10% of the rows for each column. To clean the dataset, it drops all rows containing any missing values using `dropna()`, ensuring that the remaining data is complete. Finally, the cleaned

dataset is saved to an Excel file named 'cleaned_large_road_traffic_data.xlsx', allowing the data to be exported and used for further analysis or reporting.

```
import pandas as pd
import numpy as np
import random

# Step 1: Generate a larger synthetic road traffic dataset
np.random.seed(42)
rows = 10000
dates = pd.date_range('2023-01-01', periods=rows, freq='H')
times = [f"{random.randint(0, 23)}:02d": f"{random.randint(0, 59)}:02d}" for _ in range(rows)]
locations = [f"Location_{i}" for i in np.random.randint(1, 50, rows)]
vehicle_counts = np.random.randint(50, 500, rows)
average_speeds = np.random.randint(30, 120, rows)
incidents = np.random.choice([0, 1], size=rows)

# Create DataFrame
data = {
    'Date': dates,
    'Time': times,
    'Location': locations,
    'Vehicle_Count': vehicle_counts,
    'Average_Speed': average_speeds,
    'Incident': incidents
}

df = pd.DataFrame(data)

# Introduce some missing data
```

```
for col in df.columns:  
    df.loc[df.sample(frac=0.1).index, col] =  
    np.nan  
  
# Step 2: Handle missing data by dropping rows  
with any missing values  
df_cleaned = df.dropna()  
  
# Step 3: Save the cleaned DataFrame to an Excel  
file  
output_file =  
'cleaned_large_road_traffic_data.xlsx'  
df_cleaned.to_excel(output_file, index=False)  
  
output_file
```

Let's break down the code step by step, explaining each part in detail:

Imports

```
import pandas as pd  
import numpy as np  
import random
```

- import pandas as pd: This line imports the pandas library, a powerful data manipulation tool used for handling structured data. The alias pd is commonly used to refer to pandas functions, making the code more concise.
- import numpy as np: This imports the NumPy library, a fundamental package for numerical computing in Python. NumPy provides support for arrays, matrices, and a large collection of mathematical functions to operate on these arrays.

- import random: This imports Python's built-in random module, which is used to generate random numbers and selections, allowing us to simulate random scenarios such as generating random times, locations, and other data.

Step 1: Generate a Larger Synthetic Road Traffic Dataset

```
np.random.seed(42)
rows = 10000
dates = pd.date_range('2023-01-01', periods=rows,
freq='H')
times = [f"{random.randint(0, 23):02d}":
{random.randint(0, 59):02d}" for _ in range(rows)]
locations = [f"Location_{i}" for i in
np.random.randint(1, 50, rows)]
vehicle_counts = np.random.randint(50, 500, rows)
average_speeds = np.random.randint(30, 120, rows)
incidents = np.random.choice([0, 1], size=rows)
```

- np.random.seed(42): This sets the random seed for NumPy's random number generator. By setting the seed to a fixed number (42 in this case), we ensure that the sequence of random numbers generated is reproducible. This is useful for debugging and consistency in results.
- rows = 10000: This defines a variable rows to represent the number of data points (rows) we want in our dataset. In this case, we're generating a dataset with 10,000 rows.
- dates = pd.date_range('2023-01-01', periods=rows, freq='H'): This line creates a pandas DatetimeIndex containing 10,000 hourly timestamps starting from '2023-01-01'. The pd.date_range() function is used here to generate a sequence of dates/times with a specified

frequency (freq='H' for hourly). The periods=rows argument specifies the number of timestamps to generate.

- times = [f"random.randint(0, 23):02d": {random.randint(0, 59):02d}" for _ in range(rows)]: This creates a list of time strings in the format HH:MM for each row. The list comprehension iterates rows times (10,000 in this case), and for each iteration, it generates a random hour (random.randint(0, 23)) and minute (random.randint(0, 59)), formatted as a zero-padded string. This simulates random times within a day.
- locations = [f"Location_{i}" for i in np.random.randint(1, 50, rows)]: This creates a list of location names in the format Location_X, where X is a random integer between 1 and 49. The np.random.randint(1, 50, rows) generates 10,000 random integers, and the list comprehension constructs a list of location strings using these integers.
- vehicle_counts = np.random.randint(50, 500, rows): This generates an array of 10,000 random integers representing vehicle counts, with values ranging between 50 and 499. These values are used to simulate the number of vehicles recorded at each location and time.
- average_speeds = np.random.randint(30, 120, rows): This generates an array of 10,000 random integers representing the average speed of vehicles, with values ranging between 30 and 119 km/h. These values simulate varying traffic speeds.
- incidents = np.random.choice([0, 1], size=rows): This generates an array of 10,000 random integers, where each integer is either 0 or 1, chosen randomly. The values simulate the occurrence of incidents, with 0 indicating no incident and 1 indicating an incident.

Create DataFrame

```
data = {  
    'Date': dates,  
    'Time': times,  
    'Location': locations,  
    'Vehicle_Count': vehicle_counts,  
    'Average_Speed': average_speeds,  
    'Incident': incidents  
}  
  
df = pd.DataFrame(data)
```

- `data = {...}`: This creates a dictionary where each key is the name of a column (e.g., 'Date', 'Time', etc.), and each value is the corresponding list or array created in the previous step. The dictionary structure allows for easy conversion to a pandas DataFrame.
- `df = pd.DataFrame(data)`: This converts the dictionary data into a pandas DataFrame df. The DataFrame is a two-dimensional labeled data structure with columns corresponding to the keys in the dictionary and rows corresponding to each data point.

Introduce Some Missing Data

```
for col in df.columns:  
    df.loc[df.sample(frac=0.1).index, col] =  
    np.nan
```

- `for col in df.columns`: This initiates a loop that iterates over each column name in the DataFrame df.

- `df.loc[df.sample(frac=0.1).index, col] = np.nan`: For each column, this line randomly selects 10% of the rows (`frac=0.1`) and assigns NaN (Not a Number) to those rows in the current column. The `df.sample(frac=0.1)` function returns a random sample of 10% of the DataFrame's rows, and `.index` retrieves the indices of these rows. The `df.loc[...]` operation assigns NaN to the selected rows in the specified column. This simulates missing data in the dataset.

Step 2: Handle Missing Data by Dropping Rows with Any Missing Values

```
df_cleaned = df.dropna()
```

- `df_cleaned = df.dropna()`: This line removes all rows from the DataFrame `df` that contain any `NaN` values. The `dropna()` function returns a new DataFrame `df_cleaned` that only includes rows with complete data. This step cleans the dataset by discarding incomplete records.

Step 3: Save the Cleaned DataFrame to an Excel File

```
output_file =  
'cleaned_large_road_traffic_data.xlsx'  
df_cleaned.to_excel(output_file, index=False)
```

- `output_file = 'cleaned_large_road_traffic_data.xlsx'`: This defines the filename for the Excel file that will store the cleaned dataset.
- `df_cleaned.to_excel(output_file, index=False)`: This saves the cleaned DataFrame `df_cleaned` to an Excel file with the specified filename. The `index=False` argument ensures that the DataFrame's row indices are not written to the Excel file, keeping the file clean and focused on the data itself.

Final Output

```
output_file
```

- `output_file`: This line simply returns the filename of the saved Excel file, indicating where the cleaned data is stored.

EXAMPLE 6.2

GUI Tkinter for Handling Missing Data with Dropping Rows Using Road Traffic Dataset

This project is designed to generate, clean, and analyze a synthetic road traffic dataset, and then visualize the results using a Tkinter-based graphical user interface (GUI). It starts by creating a large dataset that simulates traffic data, including variables like date, time, location, vehicle count, average speed, and whether an incident occurred. Some data points are randomly removed to introduce missing data, which is then cleaned by dropping rows with any missing values. This cleaned dataset is the focus of further analysis.

The GUI is structured into four main tabs: the original dataset, the cleaned dataset, summary statistics, and graph distributions. The original and cleaned datasets are displayed in separate tabs using Tkinter's Treeview widget, allowing users to explore the data. Summary statistics are computed for the cleaned dataset, providing insights into metrics like mean, standard deviation, and data distribution for each column. These statistics are also displayed in a Treeview in the summary tab.

In the graph distribution tab, various aspects of the traffic data are visualized using Matplotlib. Histograms and bar charts illustrate the distribution of vehicle counts, average speeds, incidents, and the most frequent locations, providing users with a visual summary of the traffic patterns. The combination of these features in a user-

friendly GUI allows for a comprehensive analysis of the synthetic road traffic dataset.

```
import pandas as pd
import numpy as np
import random
import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.pyplot as plt

# Step 1: Generate a larger synthetic road traffic
dataset
np.random.seed(42)
rows = 10000
dates = pd.date_range('2023-01-01', periods=rows,
freq='H')
times = [f"{random.randint(0, 23)}:02d}":
{random.randint(0, 59)}" for _ in range(rows)]
locations = [f"Location_{i}" for i in
np.random.randint(1, 50, rows)]
vehicle_counts = np.random.randint(50, 500, rows)
average_speeds = np.random.randint(30, 120, rows)
incidents = np.random.choice([0, 1], size=rows)

# Create DataFrame
data = {
    'Date': dates,
    'Time': times,
    'Location': locations,
    'Vehicle_Count': vehicle_counts,
    'Average_Speed': average_speeds,
    'Incident': incidents
}
```

```
df = pd.DataFrame(data)

# Introduce some missing data
for col in df.columns:
    df.loc[df.sample(frac=0.1).index, col] =
np.nan

# Step 2: Handle missing data by dropping rows
with any missing values
df_cleaned = df.dropna()

# Function to display a DataFrame in a Treeview
with alternating row colors
def display_dataframe(tab, dataframe):
    tree = ttk.Treeview(tab, columns=[str(i) for i
in dataframe.columns], show='headings')
    tree.pack(expand=True, fill='both')

    # Define headings
    for col in dataframe.columns:
        tree.heading(str(col), text=str(col))
        tree.column(str(col), width=100,
anchor='center')

    # Insert data with alternating row colors
    for i, row in dataframe.iterrows():
        tags = 'evenrow' if i % 2 == 0 else
'oddrow'
        tree.insert("", "end", values=[str(x) for
x in row], tags=(tags,))

        tree.tag_configure('evenrow',
background="#E8E8E8")
        tree.tag_configure('oddrow',
background="#DFDFDF")
```

```

# Function to create summary statistics
def summary_statistics(tab, dataframe):
    # Generate summary statistics and transpose to
    # align with columns
    stats =
    dataframe.describe(include='all').transpose()

    # Insert a 'Statistic' column for the index
    stats.insert(0, 'Statistic', stats.index)

    # Ensure all column names are strings
    columns = [str(col) for col in stats.columns]

    # Create Treeview with the correct columns
    tree = ttk.Treeview(tab, columns=columns,
    show='headings')
    tree.pack(expand=True, fill='both')

    # Define headings
    for col in columns:
        tree.heading(col, text=col)
        tree.column(col, width=120,
    anchor='center')

    # Insert summary statistics into the Treeview
    # without alternating row colors
    for i, row in stats.iterrows():
        values = [str(x) for x in row] # Convert
        each value to string
        tree.insert("", "end", values=values)

# Function to create distribution graphs
def distribution_graphs(tab, dataframe):
    fig, axs = plt.subplots(2, 2, figsize=(10, 8))

```

```
    dataframe['Vehicle_Count'].hist(ax=axs[0, 0],
bins=30, color='blue')
    axs[0, 0].set_title('Vehicle Count
Distribution')

    dataframe['Average_Speed'].hist(ax=axs[0, 1],
bins=30, color='green')
    axs[0, 1].set_title('Average Speed
Distribution')

dataframe['Incident'].value_counts().plot(kind='ba
r', ax=axs[1, 0], color='red')
    axs[1, 0].set_title('Incident Distribution')

dataframe['Location'].value_counts().head(20).plot
(kind='bar', ax=axs[1, 1], color='purple')
    axs[1, 1].set_title('Top 20 Locations by
Count')

plt.tight_layout()

canvas = FigureCanvasTkAgg(fig, master=tab)
canvas.draw()
canvas.get_tk_widget().pack(expand=True,
fill='both')

# Create the main application window
root = tk.Tk()
root.title("Road Traffic Data Analysis")
root.geometry("1200x800")

# Create a Notebook (tabbed interface)
notebook = ttk.Notebook(root)
notebook.pack(expand=True, fill='both')
```

```
# Add tabs
tab_original = ttk.Frame(notebook)
tab_cleaned = ttk.Frame(notebook)
tab_summary = ttk.Frame(notebook)
tab_graphs = ttk.Frame(notebook)

notebook.add(tab_original, text="Original
DataFrame")
notebook.add(tab_cleaned, text="Cleaned
DataFrame")
notebook.add(tab_summary, text="Summary
Statistics")
notebook.add(tab_graphs, text="Graph
Distribution")

# Display dataframes in the respective tabs
display_dataframe(tab_original, df)
display_dataframe(tab_cleaned, df_cleaned)

# Display summary statistics in the summary tab
summary_statistics(tab_summary, df_cleaned)

# Display distribution graphs in the graphs tab
distribution_graphs(tab_graphs, df_cleaned)

# Run the application
root.mainloop()
```

This code is a comprehensive example of how to create a Tkinter-based graphical user interface (GUI) for analyzing and visualizing a large synthetic road traffic dataset. The code is structured into several key steps, from generating the dataset to creating a user-friendly interface for data exploration and visualization.

1. Importing Required Libraries

The code begins by importing several essential Python libraries:

- pandas and numpy are used for data manipulation and numerical operations.
- random is used to generate random times for the dataset.
- tkinter and ttk (Tkinter's themed widgets) are used to build the GUI, including tables and tabs.
- matplotlib and FigureCanvasTkAgg are used to create and display graphs within the Tkinter GUI.

2. Generating the Synthetic Road Traffic Dataset

- Setting the Random Seed: np.random.seed(42) ensures that the random numbers generated are reproducible.
- Creating the Dataset: The dataset contains 10,000 rows, representing hourly road traffic data. The following data points are generated:
 - dates: A series of dates starting from '2023-01-01', with a frequency of one hour for each row.
 - times: Randomly generated times in HH format.
 - locations: Randomly generated location identifiers (e.g., Location_1, Location_2).
 - vehicle_counts: Random integers between 50 and 500 representing the number of vehicles counted.
 - average_speeds: Random integers between 30 and 120 representing the average speed of vehicles.
 - incidents: A binary indicator (0 or 1) representing whether an incident occurred.
- Creating the DataFrame: These variables are combined into a dictionary and passed to pd.DataFrame to create a structured dataset.
- Introducing Missing Data: To simulate real-world data, 10% of the values in each column are randomly set to

NaN (missing values).

3. Cleaning the Dataset

- Dropping Missing Values: The code removes any rows that contain missing values using `df.dropna()`, creating a cleaned version of the dataset (`df_cleaned`). This step is crucial for ensuring that subsequent analysis and visualizations are based on complete data.

4. Creating the Tkinter GUI

- Initializing the Main Window: The main Tkinter window (`root`) is created with a title "Road Traffic Data Analysis" and a size of 1200x800 pixels.
- Adding a Notebook Widget: A `ttk.Notebook` is added to the main window, providing a tabbed interface for organizing different parts of the analysis.
- Creating and Adding Tabs: Four tabs are created:
 - `tab_original`: Displays the original, uncleaned dataset.
 - `tab_cleaned`: Displays the cleaned dataset.
 - `tab_summary`: Displays summary statistics of the cleaned dataset.
 - `tab_graphs`: Displays various distribution graphs based on the cleaned data.

5. Displaying the DataFrames

- `display_dataframe` Function: This function takes a tab and a DataFrame as inputs, and displays the DataFrame in a Treeview widget within the tab. The columns of the DataFrame are used as headings, and each row of data is inserted into the table. The rows are displayed with alternating colors for better readability.

6. Generating and Displaying Summary Statistics

- `summary_statistics` Function: This function calculates summary statistics for the cleaned DataFrame using `dataframe.describe(include='all').transpose()`. The statistics are displayed in a Treeview widget, similar to how the DataFrames are displayed. The statistics include metrics like count, mean, standard deviation, and percentiles for each column.

7. Creating and Displaying Distribution Graphs

- `distribution_graphs` Function: This function generates a series of distribution graphs:
 - A histogram of vehicle counts.
 - A histogram of average speeds.
 - A bar chart of incident counts (0 vs. 1).
 - A bar chart showing the top 20 locations by frequency.
- These graphs are plotted using Matplotlib, and the `FigureCanvasTkAgg` is used to embed the plots into the Tkinter tab.

8. Running the Application

Finally, `root.mainloop()` starts the Tkinter event loop, allowing users to interact with the GUI. The GUI remains responsive, enabling users to switch between tabs and explore the data and visualizations interactively.

| Road Traffic Data Analysis | | | | | | |
|----------------------------|-------------------|--------------------|--------------------|---------------|----------|--|
| Original DataFrame | Cleaned DataFrame | Summary Statistics | Graph Distribution | | | |
| Date | Time | Location | Vehicle_Count | Average_Speed | Incident | |
| 2023-01-01 00:00:00 | 04:52 | nan | 498.0 | 96.0 | nan | |
| 2023-01-01 01:00:00 | 08:55 | nan | 101.0 | 46.0 | 0.0 | |
| 2023-01-01 02:00:00 | 05:37 | Location_15 | nan | 37.0 | 1.0 | |
| 2023-01-01 03:00:00 | 15:55 | Location_43 | 255.0 | 89.0 | 0.0 | |
| 2023-01-01 04:00:00 | 22:50 | Location_8 | 336.0 | 69.0 | 0.0 | |
| 2023-01-01 05:00:00 | 18:26 | Location_21 | 450.0 | 110.0 | 0.0 | |
| 2023-01-01 06:00:00 | 02:36 | Location_39 | 488.0 | 118.0 | 1.0 | |
| 2023-01-01 07:00:00 | 16:40 | nan | 415.0 | 91.0 | 0.0 | |
| NaT | 17:27 | Location_23 | 142.0 | 70.0 | 0.0 | |
| 2023-01-01 09:00:00 | 18:01 | nan | 469.0 | 41.0 | 0.0 | |
| 2023-01-01 10:00:00 | 15:24 | nan | nan | 118.0 | 0.0 | |
| 2023-01-01 11:00:00 | 01:07 | Location_24 | 358.0 | nan | 0.0 | |
| NaT | 08:47 | Location_36 | 194.0 | 118.0 | 0.0 | |
| 2023-01-01 13:00:00 | 06:52 | Location_40 | 135.0 | 40.0 | 0.0 | |
| 2023-01-01 14:00:00 | 12:58 | Location_24 | 202.0 | 89.0 | 1.0 | |
| 2023-01-01 15:00:00 | 09:34 | Location_3 | 51.0 | 97.0 | 0.0 | |
| 2023-01-01 16:00:00 | 19:35 | Location_22 | 391.0 | 107.0 | nan | |
| 2023-01-01 17:00:00 | 11:47 | nan | 438.0 | 75.0 | 1.0 | |
| NaT | nan | Location_24 | 309.0 | nan | 1.0 | |
| 2023-01-01 19:00:00 | 16:56 | Location_44 | nan | 90.0 | 1.0 | |
| 2023-01-01 20:00:00 | 10:11 | Location_30 | 55.0 | 46.0 | 1.0 | |
| 2023-01-01 21:00:00 | 19:47 | Location_38 | nan | 95.0 | 1.0 | |
| 2023-01-01 22:00:00 | 21:16 | Location_2 | 254.0 | 101.0 | 0.0 | |
| 2023-01-01 23:00:00 | 14:53 | Location_21 | 167.0 | 65.0 | 0.0 | |
| 2023-01-02 00:00:00 | nan | Location_33 | 138.0 | 103.0 | 0.0 | |
| 2023-01-02 01:00:00 | 01:52 | Location_12 | nan | 36.0 | 1.0 | |
| 2023-01-02 02:00:00 | 04:40 | Location_22 | 488.0 | 45.0 | 0.0 | |
| 2023-01-02 03:00:00 | 01:16 | Location_44 | 216.0 | 43.0 | 0.0 | |
| 2023-01-02 04:00:00 | 05:34 | Location_25 | 210.0 | 109.0 | nan | |
| 2023-01-02 05:00:00 | 23:11 | Location_49 | 319.0 | 54.0 | 1.0 | |
| 2023-01-02 06:00:00 | 10:59 | Location_27 | nan | 33.0 | 0.0 | |

| Road Traffic Data Analysis | | | | | | |
|----------------------------|-------------------|--------------------|--------------------|---------------|----------|--|
| Original DataFrame | Cleaned DataFrame | Summary Statistics | Graph Distribution | | | |
| Date | Time | Location | Vehicle_Count | Average_Speed | Incident | |
| 2023-01-01 03:00:00 | 15:55 | Location_43 | 255.0 | 89.0 | 0.0 | |
| 2023-01-01 04:00:00 | 22:50 | Location_8 | 336.0 | 69.0 | 0.0 | |
| 2023-01-01 05:00:00 | 18:26 | Location_21 | 450.0 | 110.0 | 0.0 | |
| 2023-01-01 06:00:00 | 02:36 | Location_39 | 488.0 | 118.0 | 1.0 | |
| 2023-01-01 13:00:00 | 06:52 | Location_40 | 135.0 | 40.0 | 0.0 | |
| 2023-01-01 14:00:00 | 12:58 | Location_24 | 202.0 | 89.0 | 1.0 | |
| 2023-01-01 15:00:00 | 09:34 | Location_3 | 51.0 | 97.0 | 0.0 | |
| 2023-01-01 20:00:00 | 10:11 | Location_30 | 55.0 | 46.0 | 1.0 | |
| 2023-01-01 22:00:00 | 21:16 | Location_2 | 254.0 | 101.0 | 0.0 | |
| 2023-01-01 23:00:00 | 14:53 | Location_21 | 167.0 | 65.0 | 0.0 | |
| 2023-01-02 02:00:00 | 04:40 | Location_22 | 488.0 | 45.0 | 0.0 | |
| 2023-01-02 03:00:00 | 01:16 | Location_44 | 216.0 | 43.0 | 0.0 | |
| 2023-01-02 05:00:00 | 23:11 | Location_49 | 319.0 | 54.0 | 1.0 | |
| 2023-01-02 07:00:00 | 19:42 | Location_42 | 277.0 | 58.0 | 1.0 | |
| 2023-01-02 08:00:00 | 22:31 | Location_28 | 386.0 | 85.0 | 0.0 | |
| 2023-01-02 09:00:00 | 23:34 | Location_16 | 198.0 | 35.0 | 1.0 | |
| 2023-01-02 10:00:00 | 00:59 | Location_15 | 95.0 | 30.0 | 0.0 | |
| 2023-01-02 11:00:00 | 18:37 | Location_47 | 193.0 | 68.0 | 0.0 | |
| 2023-01-02 15:00:00 | 06:09 | Location_7 | 389.0 | 96.0 | 1.0 | |
| 2023-01-02 21:00:00 | 19:57 | Location_25 | 324.0 | 77.0 | 1.0 | |
| 2023-01-02 23:00:00 | 11:27 | Location_9 | 115.0 | 83.0 | 1.0 | |
| 2023-01-03 00:00:00 | 05:43 | Location_26 | 128.0 | 33.0 | 1.0 | |
| 2023-01-03 02:00:00 | 15:22 | Location_20 | 173.0 | 84.0 | 1.0 | |
| 2023-01-03 03:00:00 | 00:21 | Location_28 | 251.0 | 73.0 | 1.0 | |
| 2023-01-03 04:00:00 | 20:23 | Location_47 | 199.0 | 118.0 | 1.0 | |
| 2023-01-03 05:00:00 | 13:43 | Location_7 | 95.0 | 41.0 | 0.0 | |
| 2023-01-03 06:00:00 | 19:08 | Location_44 | 162.0 | 110.0 | 0.0 | |
| 2023-01-03 07:00:00 | 11:41 | Location_8 | 140.0 | 108.0 | 0.0 | |
| 2023-01-03 09:00:00 | 15:25 | Location_35 | 338.0 | 81.0 | 0.0 | |
| 2023-01-03 10:00:00 | 22:24 | Location_14 | 438.0 | 105.0 | 0.0 | |
| 2023-01-03 12:00:00 | 00:15 | Location_36 | 126.0 | 94.0 | 0.0 | |

Road Traffic Data Analysis

| Original DataFrame | Cleaned DataFrame | Summary Statistics | Graph Distribution | | | | | | |
|--------------------|-------------------|--------------------|--------------------|------|-----------------------|---------------------|---------------------|---------------------|---------------------|
| Statistic | count | unique | top | freq | mean | min | 25% | 50% | 75% |
| Date | 5317 | nan | nan | nan | 2023-07-27 23:25:10.5 | 2023-01-01 03:00:00 | 2023-04-14 18:00:00 | 2023-07-28 11:00:00 | 2023-11-07 16:00:00 |
| Time | 5317 | 1399 | 17:10 | 12 | nan | nan | nan | nan | nan |
| Location | 5317 | 49 | Location_49 | 126 | nan | nan | nan | nan | nan |
| Vehicle_Count | 5317.0 | nan | nan | nan | 272.7444047395148 | 50.0 | 158.0 | 272.0 | 385.0 |
| Average_Speed | 5317.0 | nan | nan | nan | 74.24205378973105 | 30.0 | 51.0 | 74.0 | 97.0 |
| Incident | 5317.0 | nan | nan | nan | 0.4920067707353771 | 0.0 | 0.0 | 0.0 | 1.0 |



EXAMPLE 6.3

Handling Missing Data by Filling It Using Synthetic Electricity Dataset

The purpose of this project is to simulate a large-scale electricity dataset that mimics real-world data patterns and challenges. By generating synthetic data across multiple regions and a full calendar year, the project creates a realistic scenario for analyzing electricity consumption, temperature fluctuations, and pricing. This synthetic dataset is particularly useful for testing data processing workflows, such as handling missing data, without the need to access sensitive or proprietary real-world data.

In this project, missing data is deliberately introduced to emulate common issues encountered in real-world datasets. The handling of missing data is demonstrated through techniques such as filling missing electricity consumption values with the regional average, forward-filling temperature data, and replacing missing price values with the overall mean. The resulting cleaned dataset is then saved to an Excel file, providing a comprehensive resource for further analysis or as a test bed for developing data analysis algorithms. This approach allows for the exploration of data imputation methods in a controlled environment, aiding in the development of robust data processing skills.

```
import pandas as pd
import numpy as np

# Step 1: Generate the Enlarged Synthetic
Electricity Dataset
np.random.seed(42)
date_range = pd.date_range(start="2023-01-01",
end="2023-12-31", freq='D')
regions = ['North', 'South', 'East', 'West',
'Central', 'Northeast', 'Southeast', 'Northwest',
```

```
'Southwest']\n\ndata = {\n    'Date': np.tile(date_range, len(regions)),\n    'Region': np.repeat(regions, len(date_range)),\n    'Electricity_Consumption':\n        np.random.uniform(200, 1000, len(date_range) *\n            len(regions)),\n    'Temperature': np.random.uniform(-10, 35,\n        len(date_range) * len(regions)),\n    'Price_per_MWh': np.random.uniform(20, 150,\n        len(date_range) * len(regions))\n}\n\ndf = pd.DataFrame(data)\n\n# Introduce missing values\ndf.loc[df.sample(frac=0.1).index,\n    'Electricity_Consumption'] = np.nan\ndf.loc[df.sample(frac=0.05).index, 'Temperature']\n= np.nan\ndf.loc[df.sample(frac=0.05).index,\n    'Price_per_MWh'] = np.nan\n\n# Step 2: Handle Missing Data by Filling It\ndf['Electricity_Consumption'] =\ndf.groupby('Region')\n    ['Electricity_Consumption'].transform(lambda x:\n        x.fillna(x.mean()))\ndf['Temperature'] = df['Temperature'].ffill()\ndf['Price_per_MWh'] =\ndf['Price_per_MWh'].fillna(df['Price_per_MWh'].mea\nn())\n\n# Step 3: Save the DataFrame to Excel
```

```
df.to_excel('synthetic_electricity_data.xlsx',  
index=False)  
  
print("synthetic electricity dataset created and  
saved to 'synthetic_electricity_data.xlsx'")
```

Here's the explanation of each part of the code:

1. Importing Required Libraries

```
import pandas as pd  
import numpy as np
```

- `pandas as pd`: Imports the pandas library, which is essential for data manipulation and analysis. The alias `pd` is a common shorthand used in Python to reference the library.
- `numpy as np`: Imports the numpy library, which provides support for large multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. The alias `np` is a common shorthand used in Python to reference the library.

2. Setting a Random Seed

```
np.random.seed(42)
```

- `np.random.seed(42)`: This sets the seed for NumPy's random number generator. By setting the seed to a specific value (in this case, 42), the sequence of random numbers generated will be the same every time the code is run, ensuring reproducibility of the results.

3. Generating Date Range

```
date_range      = pd.date_range(start="2023-01-01",  
end="2023-12-31", freq='D')
```

- `pd.date_range(start="2023-01-01", end="2023-12-31", freq='D')`: This generates a sequence of dates from January 1, 2023, to December 31, 2023, with a daily frequency ('D'). The result is a DatetimeIndex object containing 365 dates (one for each day of the year).

4. Defining Regions

```
regions = ['North', 'South', 'East', 'West',
'Central', 'Northeast', 'Southeast', 'Northwest',
'Southwest']
```

- regions: A list of region names representing different geographical areas. These regions are used to simulate data for multiple locations across a country or region, making the dataset more realistic and complex.

5. Creating the Data Dictionary

```
data = {
    'Date': np.tile(date_range, len(regions)),
    'Region': np.repeat(regions, len(date_range)),
                           'Electricity_Consumption':
np.random.uniform(200, 1000, len(date_range) * len(regions)),
    'Temperature': np.random.uniform(-10, 35,
len(date_range) * len(regions)),
```

```
'Price_per_MWh': np.random.uniform(20, 150,  
len(date_range) * len(regions)))  
}
```

- `np.tile(date_range, len(regions))`: This repeats the `date_range` array for each region. If there are 365 dates and 9 regions, this creates an array with $365 * 9 = 3285$ elements, where each date appears once for each region.
- `np.repeat(regions, len(date_range))`: This repeats each region name for every date in `date_range`. If there are 9 regions and 365 dates, this also creates an array with 3285 elements, where each region name is repeated 365 times.
- `np.random.uniform(200, 1000, len(date_range) * len(regions))`: This generates 3285 random numbers uniformly distributed between 200 and 1000, simulating electricity consumption in megawatt-hours (MWh).
- `np.random.uniform(-10, 35, len(date_range) * len(regions))`: This generates 3285 random numbers uniformly distributed between -10°C and 35°C, simulating daily temperature variations.
- `np.random.uniform(20, 150, len(date_range) * len(regions))`: This generates 3285 random numbers uniformly distributed between 20 and 150 USD, simulating the price per MWh of electricity.

6. Creating the DataFrame

```
df = pd.DataFrame(data)
```

- `pd.DataFrame(data)`: This converts the dictionary data into a pandas DataFrame, which is a 2D labeled data structure with columns representing the different

variables (Date, Region, Electricity_Consumption, Temperature, and Price_per_MWh) and rows representing individual records.

7. Introducing Missing Values

```
df.loc[df.sample(frac=0.1).index,
'Electricity_Consumption'] = np.nan
df.loc[df.sample(frac=0.05).index, 'Temperature'] = np.nan
df.loc[df.sample(frac=0.05).index, 'Price_per_MWh'] = np.nan
```

- df.sample(frac=0.1): This randomly samples 10% of the rows in the DataFrame.
- .loc[df.sample(frac=0.1).index, 'Electricity_Consumption'] = np.nan: This assigns NaN (Not a Number) to the Electricity_Consumption column for 10% of the rows, introducing missing values to simulate real-world data imperfections.
- df.sample(frac=0.05): Similar to the previous step, but this time sampling 5% of the rows.
- .loc[df.sample(frac=0.05).index, 'Temperature'] = np.nan and .loc[df.sample(frac=0.05).index, 'Price_per_MWh'] = np.nan: This introduces missing values in the Temperature and Price_per_MWh columns for 5% of the rows.

8. Handling Missing Data

```
df['Electricity_Consumption'] =
df.groupby('Region')
```

```
['Electricity_Consumption'].transform(lambda x:  
x.fillna(x.mean()))  
df['Temperature'] = df['Temperature'].ffill()  
df['Price_per_MWh'] =  
df['Price_per_MWh'].fillna(df['Price_per_MWh'].mea  
n())
```

- df.groupby('Region')[['Electricity_Consumption']].transform(lambda x: x.fillna(x.mean())): This groups the DataFrame by the Region column and applies a transformation to the Electricity_Consumption column. For each region, missing values in Electricity_Consumption are filled with the mean consumption value for that region.
- df['Temperature'].ffill(): This forward-fills missing temperature values, meaning that each NaN value is replaced with the last valid observation in the column.
- df['Price_per_MWh'].fillna(df['Price_per_MWh'].mean()): This fills missing Price_per_MWh values with the overall mean price from the entire column.

9. Saving the DataFrame to Excel

```
df.to_excel('synthetic_electricity_data.xlsx',  
index=False)
```

- df.to_excel('synthetic_electricity_data.xlsx', index=False): This saves the DataFrame to an Excel file named synthetic_electricity_data.xlsx. The index=False argument ensures that the DataFrame's index is not included as a separate column in the Excel file.

10. Output Message

```
print("synthetic electricity dataset created and saved to 'synthetic_electricity_data.xlsx'")
```

- `print()`: Outputs a message to the console indicating that the synthetic electricity dataset has been successfully created and saved to an Excel file.

EXAMPLE 6.4

GUI Tkinter for Handling Missing Data by Filling It Using Synthetic Electricity Dataset

This project aims to provide an interactive and comprehensive analysis tool for synthetic electricity consumption data through a graphical user interface (GUI) built with Tkinter. The application is designed to handle a large dataset containing electricity consumption, temperature, and pricing information across various regions. By generating synthetic data with realistic characteristics and introducing missing values, the project demonstrates techniques for data cleaning and preparation, including handling missing data and filling in gaps with appropriate values.

The GUI features a multi-tab interface that enables users to explore and analyze the dataset from different perspectives. The "Original DataFrame" and "Cleaned DataFrame" tabs allow users to view raw and processed data respectively, with the cleaned data being free from missing values. The "Summary Statistics" tab provides a concise statistical overview of the dataset, including measures like mean, median, and standard deviation. The "Graph Distribution" tab offers visual insights into the distribution of key variables such as electricity consumption, temperature, and pricing, using aesthetically enhanced plots from Seaborn and Matplotlib.

Additionally, the "Time-Series Graphs" tab provides an in-depth analysis of how key metrics such as electricity consumption, temperature, and pricing vary over time. These time-series plots are crucial for understanding trends and patterns in the data, offering valuable insights for decision-making and forecasting. Overall, this project integrates data generation, cleaning, and visualization to create a robust tool for analyzing electricity consumption data, showcasing practical applications of data science and GUI development.

```
import warnings
warnings.filterwarnings("ignore",
category=FutureWarning, module="seaborn")

import pandas as pd
import numpy as np
import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.pyplot as plt
import seaborn as sns

# Step 1: Generate the Enlarged Synthetic
Electricity Dataset
np.random.seed(42)
date_range = pd.date_range(start="2023-01-01",
end="2023-12-31", freq='D')
regions = ['North', 'South', 'East', 'West',
'Central', 'Northeast', 'Southeast', 'Northwest',
'Southwest']

data = {
```

```
'Date': np.tile(date_range, len(regions)),
'Region': np.repeat(regions, len(date_range)),
'Electricity_Consumption':
np.random.uniform(200, 1000, len(date_range) *
len(regions)),
'Temperature': np.random.uniform(-10, 35,
len(date_range) * len(regions)),
'Price_per_Mwh': np.random.uniform(20, 150,
len(date_range) * len(regions)))
}

df = pd.DataFrame(data)

# Introduce missing values
df.loc[df.sample(frac=0.1).index,
'Electricity_Consumption'] = np.nan
df.loc[df.sample(frac=0.05).index, 'Temperature'] =
np.nan
df.loc[df.sample(frac=0.05).index,
'Price_per_Mwh'] = np.nan

# Step 2: Handle Missing Data by Filling It
df['Electricity_Consumption'] =
df.groupby('Region')
['Electricity_Consumption'].transform(lambda x:
x.fillna(x.mean())))
df['Temperature'] = df['Temperature'].ffill()
df['Price_per_Mwh'] =
df['Price_per_Mwh'].fillna(df['Price_per_Mwh'].mea
n())

# Step 3: Save the DataFrame to Excel
df.to_excel('synthetic_electricity_data.xlsx',
index=False)
```

```

# Function to display a DataFrame in a Treeview
with alternating row colors
def display_dataframe(tab, dataframe):
    tree = ttk.Treeview(tab, columns=[str(i) for i
in dataframe.columns], show='headings')
    tree.pack(expand=True, fill='both')

    # Define headings
    for col in dataframe.columns:
        tree.heading(str(col), text=str(col))
        tree.column(str(col), width=100,
anchor='center')

    # Insert data with alternating row colors
    for i, row in dataframe.iterrows():
        tags = 'evenrow' if i % 2 == 0 else
'oddrow'
        tree.insert("", "end", values=[str(x) for
x in row], tags=(tags,))

        tree.tag_configure('evenrow',
background="#E8E8E8")
        tree.tag_configure('oddrow',
background="#DFDFDF")

# Function to create summary statistics
def summary_statistics(tab, dataframe):
    # Generate summary statistics and transpose to
align with columns
    stats =
dataframe.describe(include='all').transpose()

    # Insert a 'Statistic' column for the index
    stats.insert(0, 'Statistic', stats.index)

    # Ensure all column names are strings

```

```
columns = [str(col) for col in stats.columns]

# Create Treeview with the correct columns
tree = ttk.Treeview(tab, columns=columns,
show='headings')
tree.pack(expand=True, fill='both')

# Define headings
for col in columns:
    tree.heading(col, text=col)
    tree.column(col, width=120,
anchor='center')

# Insert summary statistics into the Treeview
without alternating row colors
for i, row in stats.iterrows():
    values = [str(x) for x in row] # Convert
each value to string
    tree.insert("", "end", values=values)

# Function to create distribution graphs
def distribution_graphs(tab, dataframe):
    fig, axs = plt.subplots(2, 2, figsize=(12,
10))

    sns.histplot(dataframe['Electricity_Consumption'],
bins=30, ax=axs[0, 0], color='blue', kde=True)
    axs[0, 0].set_title('Electricity Consumption
Distribution')

    sns.histplot(dataframe['Temperature'],
bins=30, ax=axs[0, 1], color='orange', kde=True)
    axs[0, 1].set_title('Temperature
Distribution')
```

```

    sns.histplot(dataframe['Price_per_MWh'],
bins=30, ax=axs[1, 0], color='green', kde=True)
    axs[1, 0].set_title('Price per MWh
Distribution')

    sns.countplot(y=dataframe['Region'],
order=dataframe['Region'].value_counts().index,
ax=axs[1, 1], palette='viridis')
    axs[1, 1].set_title('Region Count')

plt.tight_layout()

canvas = FigureCanvasTkAgg(fig, master=tab)
canvas.draw()
canvas.get_tk_widget().pack(expand=True,
fill='both')

# Function to create time-series graphs
def time_series_graphs(tab, dataframe):
    fig, axs = plt.subplots(3, 1, figsize=(12,
10), sharex=True)

    sns.lineplot(data=dataframe, x='Date',
y='Electricity_Consumption', hue='Region',
ax=axs[0], palette='tab10')
    axs[0].set_title('Electricity Consumption Over
Time')

    sns.lineplot(data=dataframe, x='Date',
y='Temperature', hue='Region', ax=axs[1],
palette='coolwarm')
    axs[1].set_title('Temperature Over Time')

    sns.lineplot(data=dataframe, x='Date',
y='Price_per_MWh', hue='Region', ax=axs[2],
palette='magma')

```

```
    axs[2].set_title('Price per MWh Over Time')

    plt.tight_layout()

    canvas = FigureCanvasTkAgg(fig, master=tab)
    canvas.draw()
    canvas.get_tk_widget().pack(expand=True,
fill='both')

# Create the main application window
root = tk.Tk()
root.title("Electricity Data Analysis")
root.geometry("1200x800")

# Create a Notebook (tabbed interface)
notebook = ttk.Notebook(root)
notebook.pack(expand=True, fill='both')

# Add tabs
tab_original = ttk.Frame(notebook)
tab_cleaned = ttk.Frame(notebook)
tab_summary = ttk.Frame(notebook)
tab_graphs = ttk.Frame(notebook)
tab_time_series = ttk.Frame(notebook)

notebook.add(tab_original, text="Original
DataFrame")
notebook.add(tab_cleaned, text="Cleaned
DataFrame")
notebook.add(tab_summary, text="Summary
Statistics")
notebook.add(tab_graphs, text="Graph
Distribution")
notebook.add(tab_time_series, text="Time-Series
Graphs")
```

```
# Display dataframes in the respective tabs
display_dataframe(tab_original, df)
display_dataframe(tab_cleaned, df)

# Display summary statistics in the summary tab
summary_statistics(tab_summary, df)

# Display distribution graphs in the graphs tab
distribution_graphs(tab_graphs, df)

# Display time-series graphs in the time-series
tab
time_series_graphs(tab_time_series, df)

# Run the application
root.mainloop()
```

Here's the explanation of each part of the code:

Importing Libraries and Setting Up Warnings

```
import warnings
warnings.filterwarnings("ignore",
category=FutureWarning, module="seaborn")

import pandas as pd
import numpy as np
import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.pyplot as plt
import seaborn as sns
```

- import warnings: This imports the warnings module which is used to manage warning messages.


```

regions = ['North', 'South', 'East', 'West',
'Central', 'Northeast', 'Southeast', 'Northwest',
'Southwest']

data = {
    'Date': np.tile(date_range, len(regions)),
    'Region': np.repeat(regions, len(date_range)),
    'Electricity_Consumption':
np.random.uniform(200, 1000, len(date_range) *
len(regions)),
    'Temperature': np.random.uniform(-10, 35,
len(date_range) * len(regions)),
    'Price_per_MWh': np.random.uniform(20, 150,
len(date_range) * len(regions))
}

df = pd.DataFrame(data)

```

- `np.random.seed(42)`: Sets the random seed to 42 for reproducibility of random number generation.
- `date_range = pd.date_range(start="2023-01-01", end="2023-12-31", freq='D')`: Creates a range of dates from January 1, 2023, to December 31, 2023, with daily frequency.
- `regions = [...]`: Defines a list of regions for the dataset.
- `data = {...}`: Creates a dictionary with data for each column:
 - 'Date': Uses `np.tile` to repeat the date range for each region.
 - 'Region': Uses `np.repeat` to repeat each region for each date.
 - 'Electricity_Consumption': Generates random values for electricity consumption between 200 and 1000.

- 'Temperature': Generates random temperature values between -10 and 35 degrees Celsius.
- 'Price_per_MWh': Generates random values for price per MWh between 20 and 150.
- df = pd.DataFrame(data): Creates a DataFrame from the dictionary.

Handling Missing Data

```
# Introduce missing values
df.loc[df.sample(frac=0.1).index,
'Electricity_Consumption'] = np.nan
df.loc[df.sample(frac=0.05).index,    'Temperature'] = np.nan
df.loc[df.sample(frac=0.05).index,
'Price_per_Mwh'] = np.nan

# Step 2: Handle Missing Data by Filling It
df['Electricity_Consumption'] = df.groupby('Region')[['Electricity_Consumption']].transform(lambda x: x.fillna(x.mean()))
df['Temperature'] = df['Temperature'].ffill()
df['Price_per_Mwh'] = df['Price_per_Mwh'].fillna(df['Price_per_Mwh'].mean())
df.loc[df.sample(frac=0.1).index,
'Electricity_Consumption'] = np.nan: Randomly assigns NaN values to 10% of the Electricity_Consumption column.
df.loc[df.sample(frac=0.05).index,    'Temperature'] = np.nan: Randomly assigns NaN values to 5% of the Temperature column.
```

```
df.loc[df.sample(frac=0.05).index,
'Price_per_MWh'] = np.nan: Randomly assigns NaN values to 5% of the Price_per_MWh column.
```

- df['Electricity_Consumption'] = df.groupby('Region')['Electricity_Consumption'].transform(lambda x: x.fillna(x.mean())): Fills missing values in Electricity_Consumption with the mean value of the respective region.
- df['Temperature'] = df['Temperature'].ffill(): Fills missing values in Temperature using forward fill method (propagates the last valid observation forward).
- df['Price_per_MWh'] = df['Price_per_MWh'].fillna(df['Price_per_MWh'].mean()): Fills missing values in Price_per_MWh with the overall mean of the column.

Saving the DataFrame to Excel

```
# Step 3: Save the DataFrame to Excel
df.to_excel('synthetic_electricity_data.xlsx',
index=False)
```

- df.to_excel('synthetic_electricity_data.xlsx', index=False): Saves the DataFrame to an Excel file named synthetic_electricity_data.xlsx without including row indices.

Function Definitions

Displaying DataFrames

```
def display_dataframe(tab, dataframe):
```

```

    tree = ttk.Treeview(tab, columns=[str(i) for i
in dataframe.columns], show='headings')
    tree.pack(expand=True, fill='both')

    # Define headings
    for col in dataframe.columns:
        tree.heading(str(col), text=str(col))
        tree.column(str(col), width=100,
anchor='center')

    # Insert data with alternating row colors
    for i, row in dataframe.iterrows():
        tags = 'evenrow' if i % 2 == 0 else
'oddrow'
        tree.insert("", "end", values=[str(x) for
x in row], tags=(tags,))

        tree.tag_configure('evenrow',
background='#E8E8E8')
        tree.tag_configure('oddrow',
background='#DFDFDF')

```

- def display_dataframe(tab, dataframe):: Defines a function to display a DataFrame in a Treeview widget on a given tab.
- tree = ttk.Treeview(tab, columns=[str(i) for i in dataframe.columns], show='headings'): Creates a Treeview widget with columns corresponding to DataFrame columns.
- tree.pack(expand=True, fill='both'): Packs the Treeview widget to expand and fill the tab.
- for col in dataframe.columns:: Iterates over DataFrame columns to set column headings and widths.

- `for i, row in dataframe.iterrows():` Iterates over DataFrame rows to insert data into the Treeview, alternating row colors.
- `tags = 'evenrow' if i % 2 == 0 else 'oddrow':` Assigns row tags for alternating colors.
- `tree.tag_configure('evenrow', background='#E8E8E8'):` Configures the color for even rows.
- `tree.tag_configure('oddrow', background='#DFDFDF'):` Configures the color for odd rows.

Summary Statistics

```
def summary_statistics(tab, dataframe):
    # Generate summary statistics and transpose to
    align with columns
    stats = dataframe.describe(include='all').transpose()

    # Insert a 'Statistic' column for the index
    stats.insert(0, 'Statistic', stats.index)

    # Ensure all column names are strings
    columns = [str(col) for col in stats.columns]

    # Create Treeview with the correct columns
    tree = ttk.Treeview(tab, columns=columns,
show='headings')
    tree.pack(expand=True, fill='both')

    # Define headings
    for col in columns:
        tree.heading(col, text=col)
        tree.column(col, width=120,
anchor='center')
```

```

# Insert summary statistics into the Treeview
without alternating row colors
for i, row in stats.iterrows():
    values = [str(x) for x in row] # Convert
each value to string
    tree.insert("", "end", values=values)

```

- def summary_statistics(tab, dataframe):: Defines a function to display summary statistics of a DataFrame in a Treeview.
- stats = dataframe.describe(include='all').transpose(): Computes summary statistics for all columns and transposes the result for better alignment.
- stats.insert(0, 'Statistic', stats.index): Inserts an index column labeled 'Statistic' at the beginning of the stats DataFrame.
- columns = [str(col) for col in stats.columns]: Ensures all column names are strings.
- tree = ttk.Treeview(tab, columns=columns, show='headings'): Creates a Treeview widget with columns for the summary statistics.
- for col in columns:: Sets column headings and widths.
- for i, row in stats.iterrows(): Inserts summary statistics into the Treeview.

Distribution Graphs

```

def distribution_graphs(tab, dataframe):
    fig, axs = plt.subplots(2, 2, figsize=(12, 10))

    sns.histplot(dataframe['Electricity_Consumption'],

```

```

bins=30, ax=axs[0, 0], color='blue', kde=True)
    axs[0, 0].set_title('Electricity Consumption
Distribution')

sns.histplot(dataframe['Temperature'],
bins=30, ax=axs[0, 1], color='orange', kde=True)
    axs[0, 1].set_title('Temperature
Distribution')

sns.histplot(dataframe['Price_per_MWh'],
bins=30, ax=axs[1, 0], color='green', kde=True)
    axs[1, 0].set_title('Price per MWh
Distribution')

sns.countplot(y=dataframe['Region'],
order=dataframe['Region'].value_counts().index,
ax=axs[1, 1], palette='viridis')
    axs[1, 1].set_title('Region Count')

plt.tight_layout()

canvas = FigureCanvasTkAgg(fig, master=tab)
canvas.draw()
    canvas.get_tk_widget().pack(expand=True,
fill='both')

```

- def distribution_graphs(tab, dataframe):: Defines a function to create and display distribution graphs in a tkinter tab.
- fig, axs = plt.subplots(2, 2, figsize=(12, 10)): Creates a 2x2 grid of subplots with specified figure size.
- sns.histplot(dataframe['Electricity_Consumption'],
bins=30, ax=axs[0, 0], color='blue', kde=True): Plots a

histogram of electricity consumption with kernel density estimate (KDE) on the top-left subplot.

- `axs[0, 0].set_title('Electricity Consumption Distribution'):` Sets the title for the top-left subplot.
- `sns.histplot(dataframe['Temperature'], bins=30, ax=axs[0, 1], color='orange', kde=True):` Plots a histogram of temperature with KDE on the top-right subplot.
- `axs[0, 1].set_title('Temperature Distribution'):` Sets the title for the top-right subplot.
- `sns.histplot(dataframe['Price_per_MWh'], bins=30, ax=axs[1, 0], color='green', kde=True):` Plots a histogram of price per MWh with KDE on the bottom-left subplot.
- `axs[1, 0].set_title('Price per MWh Distribution'):` Sets the title for the bottom-left subplot.
- `sns.countplot(y=dataframe['Region'], order=dataframe['Region'].value_counts().index, ax=axs[1, 1], palette='viridis'):` Creates a count plot of regions on the bottom-right subplot.
- `axs[1, 1].set_title('Region Count'):` Sets the title for the bottom-right subplot.
- `plt.tight_layout():` Adjusts subplot parameters for a clean layout.
- `canvas = FigureCanvasTkAgg(fig, master=tab):` Creates a FigureCanvasTkAgg object to embed the Matplotlib figure in the tkinter tab.
- `canvas.draw():` Draws the figure on the canvas.
- `canvas.get_tk_widget().pack(expand=True, fill='both'):` Packs the canvas widget to expand and fill the tab.

Time-Series Graphs

```

def time_series_graphs(tab, dataframe):
    fig, axs = plt.subplots(3, 1, figsize=(12, 10), sharex=True)

        sns.lineplot(data=dataframe, x='Date',
y='Electricity_Consumption', hue='Region',
ax=axs[0], palette='tab10')
    axs[0].set_title('Electricity Consumption Over Time')

        sns.lineplot(data=dataframe, x='Date',
y='Temperature', hue='Region', ax=axs[1],
palette='coolwarm')
    axs[1].set_title('Temperature Over Time')

        sns.lineplot(data=dataframe, x='Date',
y='Price_per_MWh', hue='Region', ax=axs[2],
palette='magma')
    axs[2].set_title('Price per MWh Over Time')

    plt.tight_layout()

    canvas = FigureCanvasTkAgg(fig, master=tab)
    canvas.draw()
    canvas.get_tk_widget().pack(expand=True,
fill='both')

```

- def time_series_graphs(tab, dataframe):: Defines a function to create and display time-series graphs in a tkinter tab.
- fig, axs = plt.subplots(3, 1, figsize=(12, 10), sharex=True): Creates a 3x1 grid of subplots with shared x-axis.
- sns.lineplot(data=dataframe, x='Date', y='Electricity_Consumption', hue='Region', ax=axs[0],

palette='tab10'): Plots a time-series line plot of electricity consumption over time on the top subplot.

- axs[0].set_title('Electricity Consumption Over Time'): Sets the title for the top subplot.
- sns.lineplot(data=dataframe, x='Date', y='Temperature', hue='Region', ax=axs[1], palette='coolwarm'): Plots a time-series line plot of temperature over time on the middle subplot.
- axs[1].set_title('Temperature Over Time'): Sets the title for the middle subplot.
- sns.lineplot(data=dataframe, x='Date', y='Price_per_MWh', hue='Region', ax=axs[2], palette='magma'): Plots a time-series line plot of price per MWh over time on the bottom subplot.
- axs[2].set_title('Price per MWh Over Time'): Sets the title for the bottom subplot.
- plt.tight_layout(): Adjusts subplot parameters for a clean layout.
- canvas = FigureCanvasTkAgg(fig, master=tab): Creates a FigureCanvasTkAgg object to embed the Matplotlib figure in the tkinter tab.
- canvas.draw(): Draws the figure on the canvas.
- canvas.get_tk_widget().pack(expand=True, fill='both'): Packs the canvas widget to expand and fill the tab.

Creating and Running the Tkinter Application

```
# Create the main application window
root = tk.Tk()
root.title("Electricity Data Analysis")
root.geometry("1200x800")
```

```
# Create a Notebook (tabbed interface)
notebook = ttk.Notebook(root)
notebook.pack(expand=True, fill='both')

# Add tabs
tab_original = ttk.Frame(notebook)
tab_cleaned = ttk.Frame(notebook)
tab_summary = ttk.Frame(notebook)
tab_graphs = ttk.Frame(notebook)
tab_time_series = ttk.Frame(notebook)

notebook.add(tab_original, text="Original DataFrame")
notebook.add(tab_cleaned, text="Cleaned DataFrame")
notebook.add(tab_summary, text="Summary Statistics")
notebook.add(tab_graphs, text="Graph Distribution")
notebook.add(tab_time_series, text="Time-Series Graphs")

# Display dataframes in the respective tabs
display_dataframe(tab_original, df)
display_dataframe(tab_cleaned, df)

# Display summary statistics in the summary tab
summary_statistics(tab_summary, df)

# Display distribution graphs in the graphs tab
distribution_graphs(tab_graphs, df)

# Display time-series graphs in the time-series tab
time_series_graphs(tab_time_series, df)
```

```
# Run the application
root.mainloop()
```

- root = tk.Tk(): Creates the main application window.
- root.title("Electricity Data Analysis"): Sets the title of the main window.
- root.geometry("1200x800"): Sets the size of the main window.
- notebook = ttk.Notebook(root): Creates a Notebook widget (a tabbed interface) as a child of the main window.
- notebook.pack(expand=True, fill='both'): Packs the Notebook widget to expand and fill the main window.
- tab_original = ttk.Frame(notebook): Creates a frame for the "Original DataFrame" tab.
- tab_cleaned = ttk.Frame(notebook): Creates a frame for the "Cleaned DataFrame" tab.
- tab_summary = ttk.Frame(notebook): Creates a frame for the "Summary Statistics" tab.
- tab_graphs = ttk.Frame(notebook): Creates a frame for the "Graph Distribution" tab.
- tab_time_series = ttk.Frame(notebook): Creates a frame for the "Time-Series Graphs" tab.
- notebook.add(tab_original, text="Original DataFrame"): Adds the "Original DataFrame" tab to the Notebook.
- notebook.add(tab_cleaned, text="Cleaned DataFrame"): Adds the "Cleaned DataFrame" tab to the Notebook.
- notebook.add(tab_summary, text="Summary Statistics"): Adds the "Summary Statistics" tab to the Notebook.
- notebook.add(tab_graphs, text="Graph Distribution"): Adds the "Graph Distribution" tab to the Notebook.

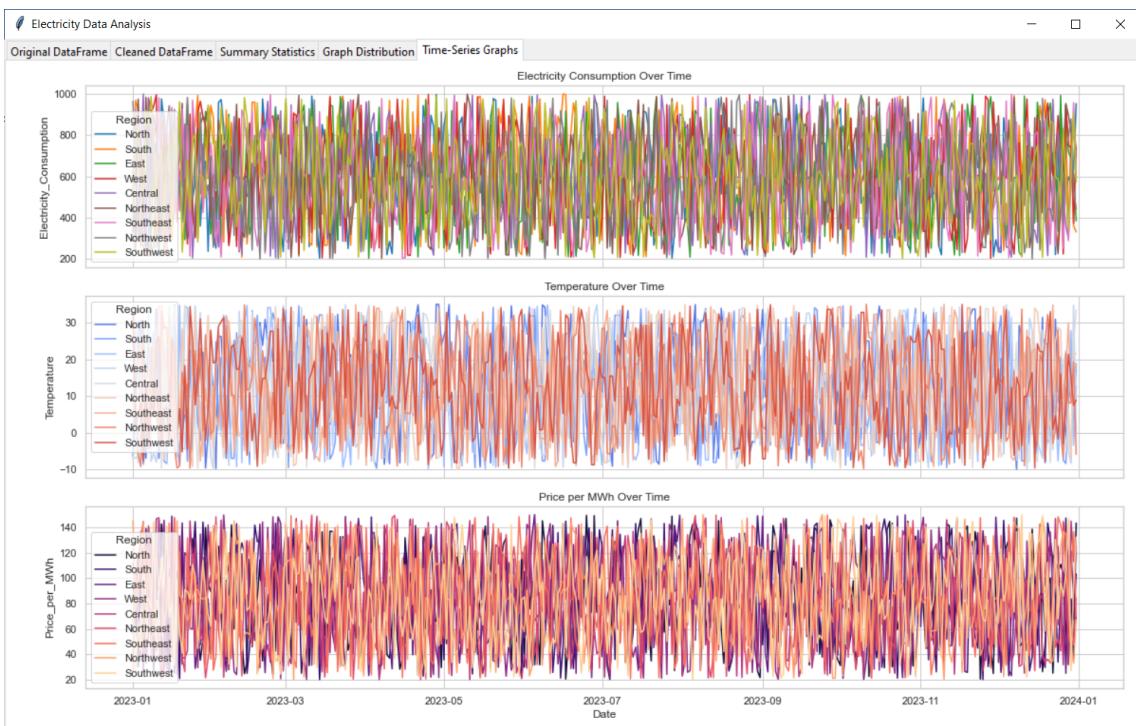
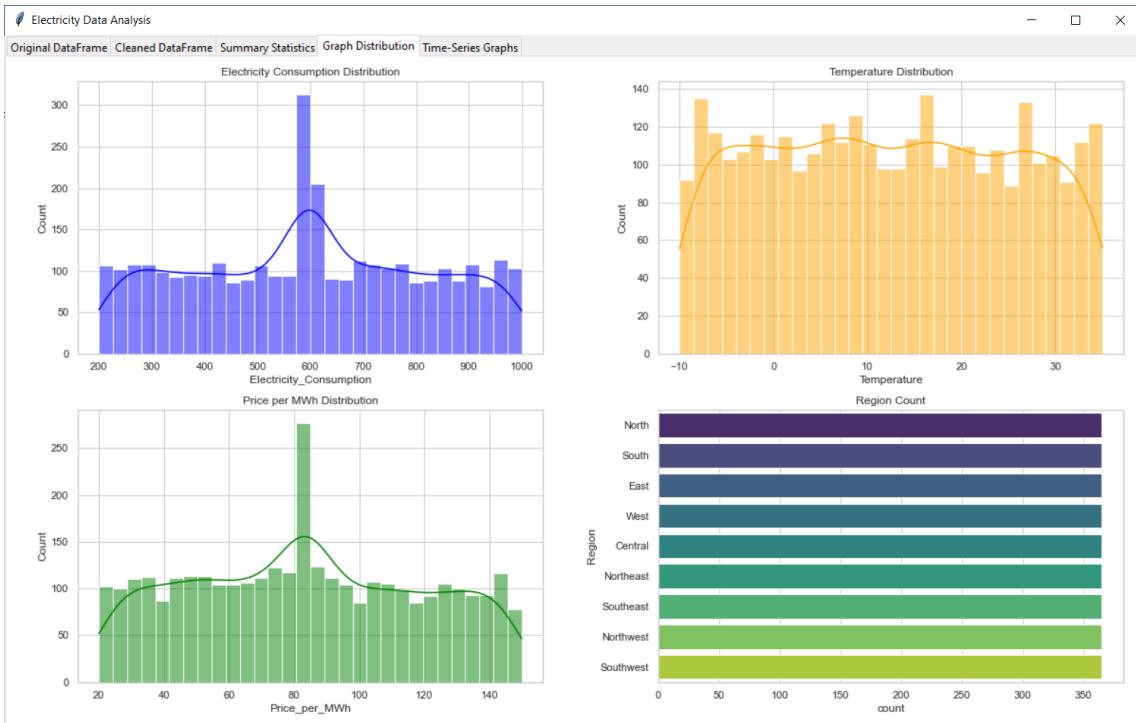
- `notebook.add(tab_time_series, text="Time-Series Graphs")`: Adds the "Time-Series Graphs" tab to the Notebook.
- `display_dataframe(tab_original, df)`: Displays the original DataFrame in the "Original DataFrame" tab.
- `display_dataframe(tab_cleaned, df)`: Displays the cleaned DataFrame in the "Cleaned DataFrame" tab.
- `summary_statistics(tab_summary, df)`: Displays summary statistics in the "Summary Statistics" tab.
- `distribution_graphs(tab_graphs, df)`: Displays distribution graphs in the "Graph Distribution" tab.
- `time_series_graphs(tab_time_series, df)`: Displays time-series graphs in the "Time-Series Graphs" tab.
- `root.mainloop()`: Starts the tkinter event loop to run the application.

Electricity Data Analysis

| Electricity Data Analysis | | | | |
|---------------------------|-------------------|-------------------------|----------------------|--------------------|
| Original DataFrame | Cleaned DataFrame | Summary Statistics | Graph Distribution | Time-Series Graphs |
| Date | Region | Electricity_Consumption | Temperature | Price_per_MWh |
| 2023-01-01 00:00:00 | North | 499.63209507788997 | -6.593512271059564 | 93.07441474101984 |
| 2023-01-02 00:00:00 | North | 598.1797359052366 | 7.452174046261906 | 83.48523176160363 |
| 2023-01-03 00:00:00 | North | 785.59513449124 | 26.159194675817155 | 30.20628289896928 |
| 2023-01-04 00:00:00 | North | 678.9267873576293 | 30.579831284580294 | 134.5635488091915 |
| 2023-01-05 00:00:00 | North | 324.81491235394924 | -0.8437265876081881 | 30.1242685818678 |
| 2023-01-06 00:00:00 | North | 324.79561626896214 | -6.986187458862782 | 83.48523176160363 |
| 2023-01-07 00:00:00 | North | 246.46688973455957 | 29.48077039097774 | 139.60496133417683 |
| 2023-01-08 00:00:00 | North | 892.9409166199481 | 7.52322254093865 | 55.1888685922504 |
| 2023-01-09 00:00:00 | North | 680.89209394567 | 14.379243392040348 | 68.16388681773685 |
| 2023-01-10 00:00:00 | North | 766.4580622368364 | 33.56296154228415 | 83.48523176160363 |
| 2023-01-11 00:00:00 | North | 216.46759545664194 | -7.004587243859971 | 83.48523176160363 |
| 2023-01-12 00:00:00 | North | 975.9278817295955 | 19.174310335344302 | 142.44772634385464 |
| 2023-01-13 00:00:00 | North | 865.9541126403374 | 19.174310335344302 | 116.14322225446485 |
| 2023-01-14 00:00:00 | North | 369.87128854262096 | 6.896101780992517 | 31.26056891110223 |
| 2023-01-15 00:00:00 | North | 345.45997373656805 | 26.1716545337051 | 106.9900402195306 |
| 2023-01-16 00:00:00 | North | 346.72360788274705 | 9.506250474003547 | 73.33948384344606 |
| 2023-01-17 00:00:00 | North | 443.39379436763016 | 34.87320592563277 | 81.191129933838 |
| 2023-01-18 00:00:00 | North | 619.8051453057903 | 15.15766782597098 | 26.389290744873016 |
| 2023-01-19 00:00:00 | North | 545.5560149136926 | 4.45229686617907 | 44.83807811502656 |
| 2023-01-20 00:00:00 | North | 598.1797359052366 | -0.0949903428610915 | 83.48523176160363 |
| 2023-01-21 00:00:00 | North | 689.4823157779035 | 5.773424042877281 | 83.48523176160363 |
| 2023-01-22 00:00:00 | North | 311.59508852163344 | 6.763997319179786 | 83.48523176160363 |
| 2023-01-23 00:00:00 | North | 433.71571882817454 | -6.907884502421756 | 69.92247961544456 |
| 2023-01-24 00:00:00 | North | 493.0894746349534 | 6.630838974204792 | 131.07193673233843 |
| 2023-01-25 00:00:00 | North | 564.8559873736287 | 10.89073388265122 | 117.36987226264439 |
| 2023-01-26 00:00:00 | North | 828.1407691144109 | 10.89073388265122 | 55.40100163354076 |
| 2023-01-27 00:00:00 | North | 359.7390257266878 | 19.552826812472357 | 35.27523922760853 |
| 2023-01-28 00:00:00 | North | 611.387507308893 | 21.8945448531194 | 52.75976881372932 |
| 2023-01-29 00:00:00 | North | 673.931655089634 | -9.62363431554143 | 139.6983386371154 |
| 2023-01-30 00:00:00 | North | 237.16033017599818 | -0.20496701887587854 | 95.6065489204682 |
| 2023-01-31 00:00:00 | North | 686.0358815211507 | 19.7757927977993543 | 89.35242713682177 |
| 2023-02-01 00:00:00 | North | 336.4192889498332 | 11.77952156808622 | 97.53116796002 |
| 2023-02-02 00:00:00 | North | 252.0412743882236 | -9.760893594924433 | 67.58367454704698 |

| Electricity Data Analysis | | | | | |
|---------------------------|-------------------|-------------------------|----------------------|--------------------|--|
| Original DataFrame | Cleaned DataFrame | Summary Statistics | Graph Distribution | Time-Series Graphs | |
| Date | Region | Electricity_Consumption | Temperature | Price_per_MWh | |
| 2023-01-01 00:00:00 | North | 499.63209507788997 | -6.593512271059564 | 93.07441474101984 | |
| 2023-01-02 00:00:00 | North | 598.1797359052366 | 7.452174046261906 | 83.48523176160363 | |
| 2023-01-03 00:00:00 | North | 785.595153449124 | 26.159194673817155 | 30.20628289896928 | |
| 2023-01-04 00:00:00 | North | 678.9267873576293 | 30.579831284580294 | 134.5635488091915 | |
| 2023-01-05 00:00:00 | North | 324.81491235394924 | -0.8437265876081881 | 30.1242685818678 | |
| 2023-01-06 00:00:00 | North | 324.79561626896214 | -6.986187458862782 | 83.48523176160363 | |
| 2023-01-07 00:00:00 | North | 246.4668897455957 | 29.4807703909774 | 139.60496133417683 | |
| 2023-01-08 00:00:00 | North | 892.9409166199481 | 7.523222540493865 | 55.18886855922504 | |
| 2023-01-09 00:00:00 | North | 680.89209394567 | 14.379243392040348 | 68.16388681773685 | |
| 2023-01-10 00:00:00 | North | 766.4580622368364 | 33.56296154228415 | 83.48523176160363 | |
| 2023-01-11 00:00:00 | North | 216.467595453664194 | -7.004587243859971 | 83.48523176160363 | |
| 2023-01-12 00:00:00 | North | 975.9278817295955 | 19.174310335344302 | 142.44772634385464 | |
| 2023-01-13 00:00:00 | North | 865.9541126403374 | 19.174310335344302 | 116.1432225446485 | |
| 2023-01-14 00:00:00 | North | 369.87128854262096 | 6.896101780992517 | 31.26056891110223 | |
| 2023-01-15 00:00:00 | North | 345.4599737656805 | 26.1716545337051 | 106.9900402195306 | |
| 2023-01-16 00:00:00 | North | 346.72360788274705 | 9.506250474003547 | 73.339483844606 | |
| 2023-01-17 00:00:00 | North | 443.39379436763016 | 34.87320592563277 | 81.191129933838 | |
| 2023-01-18 00:00:00 | North | 619.8051453057903 | 15.15766782597098 | 26.389290744873016 | |
| 2023-01-19 00:00:00 | North | 545.5560149136926 | 4.452296668617907 | 44.8307811502656 | |
| 2023-01-20 00:00:00 | North | 598.1797359052366 | -0.0949903428610915 | 83.48523176160363 | |
| 2023-01-21 00:00:00 | North | 689.482315779035 | 5.773424042877281 | 83.48523176160363 | |
| 2023-01-22 00:00:00 | North | 311.59508852163344 | 6.763997319179786 | 83.48523176160363 | |
| 2023-01-23 00:00:00 | North | 433.71571882817454 | -6.907884502421756 | 69.92247961544456 | |
| 2023-01-24 00:00:00 | North | 493.0894746349534 | 6.630838974204792 | 131.07193673233843 | |
| 2023-01-25 00:00:00 | North | 564.8559873736287 | 10.890733388265122 | 117.36987226264439 | |
| 2023-01-26 00:00:00 | North | 828.1407691144109 | 10.890733388265122 | 55.40100163534076 | |
| 2023-01-27 00:00:00 | North | 359.7390257266878 | 19.552626812472357 | 35.27523927260853 | |
| 2023-01-28 00:00:00 | North | 611.3875507308893 | 21.89445448531194 | 52.75976881372932 | |
| 2023-01-29 00:00:00 | North | 673.931655089634 | -9.62363431554143 | 139.6983386371154 | |
| 2023-01-30 00:00:00 | North | 237.16033017599818 | -0.20496701887587854 | 95.6065489204682 | |
| 2023-01-31 00:00:00 | North | 686.0358815211507 | 19.775792977993543 | 89.35242713682177 | |
| 2023-02-01 00:00:00 | North | 336.4192899498332 | 11.77952156808622 | 97.53116796002 | |
| 2023-02-02 00:00:00 | North | 252.0412743882236 | -9.760893594924433 | 67.58367454704698 | |

| Electricity Data Analysis | | | | | | | | | | |
|---------------------------|-------------------|--------------------|--------------------|--------------------|---------------------|---------------------|---------------------|---------------------|---------------------|--|
| Original DataFrame | Cleaned DataFrame | Summary Statistics | Graph Distribution | Time-Series Graphs | | | | | | |
| Statistic | count | unique | top | freq | mean | min | 25% | 50% | 75% | |
| Date | 3285 | nan | nan | nan | 2023-07-02 00:00:00 | 2023-01-01 00:00:00 | 2023-04-02 00:00:00 | 2023-07-02 00:00:00 | 2023-10-01 00:00:00 | |
| Region | 3285 | 9 | North | 365 | nan | nan | nan | nan | nan | |
| Electricity_Consumpt | 3285.0 | nan | nan | nan | 597.9506343886529 | 200.00930780429292 | 416.6658010096594 | 597.7988047139083 | 774.4984609811955 | |
| Temperature | 3285.0 | nan | nan | nan | 12.34077824566436 | -9.998617651957792 | 1.11325214635164 | 12.10986943604589 | 23.63842865359272 | |
| Price_per_MWh | 3285.0 | nan | nan | nan | 83.48523176160363 | 20.006867501198585 | 53.88662973307323 | 83.48523176160363 | 112.83131294905226 | |



EXAMPLE 6.5

Advanced Handling Missing Data by Filling It Using Synthetic Wind Power Generation Dataset

This project involves generating and processing a synthetic dataset to simulate wind power generation across various regions over the course of a year. Using pandas and NumPy, the dataset includes daily records of wind power generation, temperature, and wind speed, with missing values intentionally introduced to mimic real-world data imperfections. The data is grouped by regions, and various methods are employed to handle the missing values. Specifically, wind power generation data is imputed with the median value of each region, while temperature data is filled using forward and backward fill methods to handle gaps. Wind speed data is interpolated linearly to estimate missing values.

The processed dataset is then saved to an Excel file for further analysis or reporting. This approach ensures that the dataset remains comprehensive and usable despite the presence of missing values. By employing advanced imputation techniques, such as median substitution, forward/backward filling, and linear interpolation, the project demonstrates robust handling of incomplete data, making the synthetic dataset a valuable resource for testing and validating analytical models and visualizations related to wind power generation.

```
import pandas as pd
import numpy as np

# Set a random seed for reproducibility
np.random.seed(42)

# Generate date range for one year
date_range = pd.date_range(start="2023-01-01",
                           end="2023-12-31", freq='D')
```

```
# Define regions
regions = ['North', 'South', 'East', 'West',
'Central']

# Create synthetic data
data = {
    'Date': np.tile(date_range, len(regions)),
    'Region': np.repeat(regions, len(date_range)),
    'Wind_Power_Generation': np.random.uniform(50,
300, len(date_range) * len(regions)),
    'Temperature': np.random.uniform(-10, 35,
len(date_range) * len(regions)),
    'Wind_Speed': np.random.uniform(0, 100,
len(date_range) * len(regions))
}

df = pd.DataFrame(data)

# Introduce missing values
df.loc[df.sample(frac=0.1).index,
'Wind_Power_Generation'] = np.nan
df.loc[df.sample(frac=0.1).index, 'Temperature'] =
np.nan
df.loc[df.sample(frac=0.1).index, 'Wind_Speed'] =
np.nan

# Handle missing data by filling it
df['Wind_Power_Generation'] = df.groupby('Region')
['Wind_Power_Generation'].transform(lambda x:
x.fillna(x.median()))
df['Temperature'] =
df['Temperature'].ffill().bfill() # Updated to
use ffill() and bfill() directly
df['Wind_Speed'] =
df['Wind_Speed'].interpolate(method='linear')
```

```
# Save the DataFrame to Excel  
df.to_excel('synthetic_wind_power_data.xlsx',  
index=False)
```

Here's the breakdown of each part of the code:

1. Import Libraries

```
import pandas as pd  
import numpy as np
```

- import pandas as pd: Imports the pandas library, which is used for data manipulation and analysis. It is commonly used to work with DataFrames, which are 2-dimensional labeled data structures.
- import numpy as np: Imports the numpy library, which provides support for numerical operations and working with arrays. It's widely used for generating random numbers and performing mathematical operations.

2. Set Random Seed

```
np.random.seed(42)
```

- np.random.seed(42): Sets the seed for NumPy's random number generator. This ensures that the random numbers generated are reproducible. Using the same seed will generate the same random numbers every time the code is run, which is useful for debugging and consistent results.

3. Generate Date Range

```
date_range = pd.date_range(start="2023-01-01",
                           end="2023-12-31", freq='D')
```

- `pd.date_range(start="2023-01-01", end="2023-12-31", freq='D')`: Creates a DatetimeIndex object with daily frequency for the entire year of 2023. start and end specify the start and end dates, while freq='D' denotes a daily frequency.

4. Define Regions

```
regions = ['North', 'South', 'East', 'West',
           'Central']
```

- `regions`: A list of region names that will be used to simulate data for different geographic areas.

5. Create Synthetic Data

```
data = {
    'Date': np.tile(date_range, len(regions)),
    'Region': np.repeat(regions, len(date_range)),
    'Wind_Power_Generation': np.random.uniform(50,
300, len(date_range) * len(regions)),
    'Temperature': np.random.uniform(-10, 35,
len(date_range) * len(regions)),
    'Wind_Speed': np.random.uniform(0, 100,
len(date_range) * len(regions))
}
```

- `'Date': np.tile(date_range, len(regions))`: Repeats the date_range for each region. np.tile duplicates the date

range to match the number of regions.

- 'Region': np.repeat(regions, len(date_range)): Repeats the list of regions for each date. np.repeat creates a list where each region is repeated for each day in the date_range.
- 'Wind_Power_Generation': np.random.uniform(50, 300, len(date_range) * len(regions)): Generates random values for wind power generation between 50 and 300, with a total number of values equal to the product of the number of dates and regions.
- 'Temperature': np.random.uniform(-10, 35, len(date_range) * len(regions)): Generates random temperature values between -10 and 35 degrees Celsius.
- 'Wind_Speed': np.random.uniform(0, 100, len(date_range) * len(regions)): Generates random wind speed values between 0 and 100 km/h.

6. Create DataFrame

```
df = pd.DataFrame(data)
```

- pd.DataFrame(data): Converts the data dictionary into a pandas DataFrame. Each key in the dictionary becomes a column in the DataFrame.

7. Introduce Missing Values

```
df.loc[df.sample(frac=0.1).index, 'Wind_Power_Generation'] = np.nan
df.loc[df.sample(frac=0.1).index, 'Temperature'] = np.nan
df.loc[df.sample(frac=0.1).index, 'Wind_Speed'] = np.nan
```

- `df.sample(frac=0.1).index`: Randomly selects 10% of the rows in the DataFrame.
- `df.loc[<indices>, 'column_name'] = np.nan`: Assigns NaN values to the specified columns for the selected rows. This introduces missing values into the DataFrame to simulate real-world data issues.

8. Handle Missing Data by Filling It

```
df['Wind_Power_Generation'] = df.groupby('Region')
['Wind_Power_Generation'].transform(lambda x:
x.fillna(x.median()))
df['Temperature'] =
df['Temperature'].ffill().bfill() # Updated to
use ffill() and bfill() directly
df['Wind_Speed'] =
df['Wind_Speed'].interpolate(method='linear')
```

- `df.groupby('Region')['Wind_Power_Generation'].transform(lambda x: x.fillna(x.median()))`: Groups the data by 'Region' and fills missing values in the 'Wind_Power_Generation' column with the median of each group. `transform` ensures that the operation is applied to each group individually.
- `df['Temperature'].ffill().bfill()`: Fills missing values in the 'Temperature' column using forward fill (ffill) to propagate the last known value forward and backward fill (bfill) to fill any remaining missing values. This method is useful for time series data where values can be interpolated from adjacent values.

- df['Wind_Speed'].interpolate(method='linear'): Uses linear interpolation to estimate missing values in the 'Wind_Speed' column based on the surrounding data points.

9. Save DataFrame to Excel

```
df.to_excel('synthetic_wind_power_data.xlsx',
index=False)
```

- df.to_excel('synthetic_wind_power_data.xlsx', index=False): Saves the DataFrame to an Excel file named 'synthetic_wind_power_data.xlsx'. The index=False parameter ensures that the DataFrame index is not included in the Excel file. This is useful for sharing and analyzing the data in spreadsheet software.

EXAMPLE 6.6

GUI Tkinter for Advanced Handling Missing Data by Filling It Using Synthetic Wind Power Generation Dataset

The purpose of this project is to create a comprehensive tool for analyzing synthetic wind power generation data through a user-friendly graphical interface. The project begins by generating a synthetic dataset that simulates daily wind power generation, temperature, and wind speed across various regions. This dataset includes realistic data points, as well as intentionally introduced missing values to test the robustness of data handling techniques. The missing data is then addressed through imputation methods to ensure the dataset is complete and reliable for subsequent analysis.

The second goal of the project is to provide detailed insights into the data through various forms of statistical summaries and visualizations. By leveraging the tkinter library, a tabbed graphical user interface (GUI) is created. This GUI features tabs for displaying the original and cleaned data, as well as for generating summary statistics. Additionally, the project includes visualizations such as distribution graphs and time-series plots. These visualizations are designed to help users better understand the distribution of wind power generation, temperature, and wind speed, as well as to track trends over time across different regions.

Finally, the project aims to deliver an interactive and informative experience by integrating these functionalities into a single application. Using matplotlib and seaborn for plotting, combined with tkinter for the GUI, users can interactively explore the data. This setup not only facilitates a deep dive into the dataset through summary statistics and graphical representations but also provides an accessible platform for users to engage with and analyze the wind power data effectively.

```
import warnings
warnings.filterwarnings("ignore",
category=FutureWarning, module="seaborn")

import pandas as pd
import numpy as np
import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Step 1: Generate the Synthetic Wind Power
Dataset
np.random.seed(42)
date_range = pd.date_range(start="2023-01-01",
                           end="2023-12-31", freq='D')
regions = ['North', 'South', 'East', 'West',
           'Central']

data = {
    'Date': np.tile(date_range, len(regions)),
    'Region': np.repeat(regions, len(date_range)),
    'Wind_Power_Generation': np.random.uniform(50,
                                              300, len(date_range) * len(regions)),
    'Temperature': np.random.uniform(-10, 35,
                                      len(date_range) * len(regions)),
    'Wind_Speed': np.random.uniform(0, 100,
                                    len(date_range) * len(regions))
}

df = pd.DataFrame(data)

# Introduce missing values
df.loc[df.sample(frac=0.1).index,
       'Wind_Power_Generation'] = np.nan
df.loc[df.sample(frac=0.1).index, 'Temperature'] = np.nan
df.loc[df.sample(frac=0.1).index, 'Wind_Speed'] = np.nan

# Step 2: Handle Missing Data by Filling It
df['Wind_Power_Generation'] = df.groupby('Region')[['Wind_Power_Generation']].transform(lambda x: x.fillna(x.median()))
df['Temperature'] =
df['Temperature'].ffill().bfill()
```

```
df['Wind_Speed'] = df['Wind_Speed'].interpolate(method='linear')

# Step 3: Save the DataFrame to Excel
df.to_excel('synthetic_wind_power_data.xlsx',
index=False)

# Function to display a DataFrame in a Treeview
# with alternating row colors
def display_dataframe(tab, dataframe):
    tree = ttk.Treeview(tab, columns=[str(i) for i
in dataframe.columns], show='headings')
    tree.pack(expand=True, fill='both')

    # Define headings
    for col in dataframe.columns:
        tree.heading(str(col), text=str(col))
        tree.column(str(col), width=120,
anchor='center')

    # Insert data with alternating row colors
    for i, row in dataframe.iterrows():
        tags = 'evenrow' if i % 2 == 0 else
'oddrow'
        tree.insert("", "end", values=[str(x) for
x in row], tags=(tags,))

        tree.tag_configure('evenrow',
background="#E8E8E8")
        tree.tag_configure('oddrow',
background="#DFDFDF")

# Function to create summary statistics
def summary_statistics(tab, dataframe):
    # Generate summary statistics and transpose to
align with columns
```

```

stats =
dataframe.describe(include='all').transpose()

# Insert a 'Statistic' column for the index
stats.insert(0, 'Statistic', stats.index)

# Ensure all column names are strings
columns = [str(col) for col in stats.columns]

# Create Treeview with the correct columns
tree = ttk.Treeview(tab, columns=columns,
show='headings')
tree.pack(expand=True, fill='both')

# Define headings
for col in columns:
    tree.heading(col, text=col)
    tree.column(col, width=120,
anchor='center')

# Insert summary statistics into the Treeview
without alternating row colors
for i, row in stats.iterrows():
    values = [str(x) for x in row] # Convert
each value to string
    tree.insert("", "end", values=values)

# Function to create distribution graphs
def distribution_graphs(tab, dataframe):
    fig, axs = plt.subplots(2, 2, figsize=(12,
10))

    # Wind Power Generation Distribution

    sns.histplot(dataframe['Wind_Power_Generation'],
bins=30, ax=axs[0, 0], color='blue', kde=True)

```

```

        axs[0, 0].set_title('Wind Power Generation
Distribution')
    for p in axs[0, 0].patches:
        height = p.get_height()
        axs[0, 0].text(p.get_x() + p.get_width() /
2., height, f'{height:.0f}', ha='center',
va='bottom')

    # Temperature Distribution
    sns.histplot(dataframe['Temperature'],
bins=30, ax=axs[0, 1], color='orange', kde=True)
    axs[0, 1].set_title('Temperature
Distribution')
    for p in axs[0, 1].patches:
        height = p.get_height()
        axs[0, 1].text(p.get_x() + p.get_width() /
2., height, f'{height:.0f}', ha='center',
va='bottom')

    # Wind Speed Distribution
    sns.histplot(dataframe['Wind_Speed'], bins=30,
ax=axs[1, 0], color='green', kde=True)
    axs[1, 0].set_title('Wind Speed Distribution')
    for p in axs[1, 0].patches:
        height = p.get_height()
        axs[1, 0].text(p.get_x() + p.get_width() /
2., height, f'{height:.0f}', ha='center',
va='bottom')

    # Region Count
    sns.countplot(y=dataframe['Region'],
order=dataframe['Region'].value_counts().index,
ax=axs[1, 1], palette='viridis')
    axs[1, 1].set_title('Region Count')
    for p in axs[1, 1].patches:
        width = p.get_width()

```

```
        axs[1, 1].text(width + 0.5, p.get_y() +
p.get_height() / 2., f'{width:.0f}', ha='left',
va='center')

plt.tight_layout()

canvas = FigureCanvasTkAgg(fig, master=tab)
canvas.draw()
canvas.get_tk_widget().pack(expand=True,
fill='both')

# Function to create time-series graphs
def time_series_graphs(tab, dataframe):
    fig, axs = plt.subplots(3, 1, figsize=(12,
15), sharex=True)

    sns.lineplot(data=dataframe, x='Date',
y='Wind_Power_Generation', hue='Region',
ax=axs[0], palette='tab10')
    axs[0].set_title('Wind Power Generation Over
Time')

    sns.lineplot(data=dataframe, x='Date',
y='Temperature', hue='Region', ax=axs[1],
palette='coolwarm')
    axs[1].set_title('Temperature Over Time')

    sns.lineplot(data=dataframe, x='Date',
y='Wind_Speed', hue='Region', ax=axs[2],
palette='magma')
    axs[2].set_title('Wind Speed Over Time')

plt.tight_layout()

canvas = FigureCanvasTkAgg(fig, master=tab)
canvas.draw()
```

```
    canvas.get_tk_widget().pack(expand=True,
fill='both')

# Create the main application window
root = tk.Tk()
root.title("Wind Power Data Analysis")
root.geometry("1200x800")

# Create a Notebook (tabbed interface)
notebook = ttk.Notebook(root)
notebook.pack(expand=True, fill='both')

# Add tabs
tab_original = ttk.Frame(notebook)
tab_cleaned = ttk.Frame(notebook)
tab_summary = ttk.Frame(notebook)
tab_graphs = ttk.Frame(notebook)
tab_time_series = ttk.Frame(notebook)

notebook.add(tab_original, text="Original
DataFrame")
notebook.add(tab_cleaned, text="Cleaned
DataFrame")
notebook.add(tab_summary, text="Summary
Statistics")
notebook.add(tab_graphs, text="Distribution
Graphs")
notebook.add(tab_time_series, text="Time-Series
Graphs")

# Display dataframes in the respective tabs
display_dataframe(tab_original, df)
display_dataframe(tab_cleaned, df)

# Display summary statistics in the summary tab
summary_statistics(tab_summary, df)
```

```
# Display distribution graphs in the graphs tab
distribution_graphs(tab_graphs, df)

# Display time-series graphs in the time-series
tab
time_series_graphs(tab_time_series, df)

# Run the application
root.mainloop()
```

Here's the breakdown of each part of the provided code:

Imports and Warning Filter

```
import warnings
warnings.filterwarnings("ignore",
category=FutureWarning, module="seaborn")

import pandas as pd
import numpy as np
import tkinter as tk
from tkinter import ttk
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
import matplotlib.pyplot as plt
import seaborn as sns
```

- Warnings: The `warnings` module is used to filter out `FutureWarning` messages from `seaborn`. This can help in avoiding unnecessary warnings about future deprecations that may not affect the current functionality.

- Pandas (pd): A powerful data manipulation library used for handling data in DataFrame format.
 - NumPy (np): A library for numerical operations, including generating random numbers and handling arrays.
 - Tkinter (tk and ttk): Tkinter is used for creating the graphical user interface (GUI). The ttk module provides themed widgets like Treeview for displaying tables.
 - Matplotlib (plt): A plotting library used to create static, animated, and interactive visualizations.
 - Seaborn (sns): A statistical data visualization library that builds on Matplotlib and provides more attractive and informative plots.

Step 1: Generate the Synthetic Wind Power Dataset

```
np.random.seed(42)
date_range      = pd.date_range(start="2023-01-01",
end="2023-12-31", freq='D')
regions        = ['North', 'South', 'East', 'West',
'Central']

data = {
    'Date': np.tile(date_range, len(regions)),
    'Region': np.repeat(regions, len(date_range)),
    'Wind_Power_Generation': np.random.uniform(50,
300, len(date_range) * len(regions)),
    'Temperature': np.random.uniform(-10, 35,
len(date_range) * len(regions)),
    'Wind_Speed': np.random.uniform(0, 100,
len(date_range) * len(regions)))
}
```

```
df = pd.DataFrame(data)
```

- Random Seed: np.random.seed(42) ensures reproducibility of the random numbers.
- Date Range: pd.date_range() generates a range of dates from January 1, 2023, to December 31, 2023, with daily frequency.
- Regions: A list of five regions to simulate wind power data.
- Data Generation: np.tile and np.repeat create the synthetic dataset with columns for Date, Region, Wind Power Generation, Temperature, and Wind Speed. The data is randomly generated within specified ranges.
- DataFrame: pd.DataFrame(data) converts the dictionary into a DataFrame.

Introduce Missing Values

```
df.loc[df.sample(frac=0.1).index,
'Wind_Power_Generation'] = np.nan
df.loc[df.sample(frac=0.1).index, 'Temperature'] =
np.nan
df.loc[df.sample(frac=0.1).index, 'Wind_Speed'] =
np.nan
```

- Missing Values: Introduces missing values (NaNs) into 10% of the rows for each of the columns: 'Wind_Power_Generation', 'Temperature', and 'Wind_Speed' to simulate real-world scenarios where data might be incomplete.

Step 2: Handle Missing Data by Filling It

```
df['Wind_Power_Generation'] = df.groupby('Region')[['Wind_Power_Generation']].transform(lambda x: x.fillna(x.median()))
df['Temperature'] =
df['Temperature'].ffill().bfill()
df['Wind_Speed'] =
df['Wind_Speed'].interpolate(method='linear')
```

- Wind Power Generation: Missing values are filled with the median of each region's 'Wind_Power_Generation' to maintain regional consistency.
- Temperature: Forward fill (ffill) and backward fill (bfill) are used to fill missing temperature values by propagating the next or previous value.
- Wind Speed: Linear interpolation is used to estimate missing 'Wind_Speed' values based on the surrounding data points.

Step 3: Save the DataFrame to Excel

```
df.to_excel('synthetic_wind_power_data.xlsx',
index=False)
```

- Save to Excel: The cleaned DataFrame is saved to an Excel file named 'synthetic_wind_power_data.xlsx' without including the DataFrame index.

Function Definitions

Display DataFrame

```
def display_dataframe(tab, dataframe):
```

```

        tree = ttk.Treeview(tab, columns=[str(i) for i
in dataframe.columns], show='headings')
        tree.pack(expand=True, fill='both')

        for col in dataframe.columns:
            tree.heading(str(col), text=str(col))
            tree.column(str(col), width=120,
anchor='center')

        for i, row in dataframe.iterrows():
            tags = 'evenrow' if i % 2 == 0 else
'oddrow'
            tree.insert("", "end", values=[str(x) for
x in row], tags=(tags,))

            tree.tag_configure('evenrow',
background='#E8E8E8')
            tree.tag_configure('oddrow',
background='#DFDFDF')

```

- Treeview: Creates a table view to display the DataFrame. Each row is colored alternately for better readability.

Summary Statistics

```

def summary_statistics(tab, dataframe):
    stats = dataframe.describe(include='all').transpose()
    stats.insert(0, 'Statistic', stats.index)
    columns = [str(col) for col in stats.columns]
    tree = ttk.Treeview(tab, columns=columns,
show='headings')
    tree.pack(expand=True, fill='both')

    for col in columns:

```

```

        tree.heading(col, text=col)
                    tree.column(col,    width=120,
anchor='center')

    for i, row in stats.iterrows():
        values = [str(x) for x in row]
        tree.insert("", "end", values=values)

```

- Summary Statistics: Calculates and displays summary statistics of the DataFrame. The statistics are displayed in a Treeview without alternating row colors.

Distribution Graphs

```

def distribution_graphs(tab, dataframe):
    fig, axs = plt.subplots(2, 2, figsize=(12, 10))

    sns.histplot(dataframe['Wind_Power_Generation'],
                 bins=30, ax=axs[0, 0], color='blue', kde=True)
    axs[0, 0].set_title('Wind Power Generation Distribution')
    for p in axs[0, 0].patches:
        height = p.get_height()
        axs[0, 0].text(p.get_x() + p.get_width() / 2., height, f'{height:.0f}', ha='center', va='bottom')

    sns.histplot(dataframe['Temperature'],
                 bins=30, ax=axs[0, 1], color='orange', kde=True)
    axs[0, 1].set_title('Temperature Distribution')
    for p in axs[0, 1].patches:
        height = p.get_height()

```

```

        axs[0, 1].text(p.get_x() + p.get_width() / 2.,      height,      f'{height:.0f}',      ha='center', va='bottom')

    sns.histplot(dataframe['Wind_Speed'], bins=30, ax=axs[1, 0], color='green', kde=True)
    axs[1, 0].set_title('Wind Speed Distribution')
    for p in axs[1, 0].patches:
        height = p.get_height()
        axs[1, 0].text(p.get_x() + p.get_width() / 2.,      height,      f'{height:.0f}',      ha='center', va='bottom')

    sns.countplot(y=dataframe['Region'], order=dataframe['Region'].value_counts().index, ax=axs[1, 1], palette='viridis')
    axs[1, 1].set_title('Region Count')
    for p in axs[1, 1].patches:
        width = p.get_width()
        axs[1, 1].text(width + 0.5, p.get_y() + p.get_height() / 2., f'{width:.0f}', ha='left', va='center')

plt.tight_layout()

canvas = FigureCanvasTkAgg(fig, master=tab)
canvas.draw()
    canvas.get_tk_widget().pack(expand=True, fill='both')

```

- Distribution Graphs: Generates and displays histograms and a count plot for different data attributes (Wind Power Generation, Temperature, Wind Speed, Region Count) in a 2x2 grid layout. Adds text annotations on top of each bar to show exact values.

Time-Series Graphs

```
def time_series_graphs(tab, dataframe):
    fig, axs = plt.subplots(3, 1, figsize=(12, 15), sharex=True)

        sns.lineplot(data=dataframe, x='Date',
y='Wind_Power_Generation', hue='Region',
ax=axs[0], palette='tab10')
    axs[0].set_title('Wind Power Generation Over Time')

        sns.lineplot(data=dataframe, x='Date',
y='Temperature', hue='Region', ax=axs[1],
palette='coolwarm')
    axs[1].set_title('Temperature Over Time')

        sns.lineplot(data=dataframe, x='Date',
y='Wind_Speed', hue='Region', ax=axs[2],
palette='magma')
    axs[2].set_title('Wind Speed Over Time')

    plt.tight_layout()

    canvas = FigureCanvasTkAgg(fig, master=tab)
    canvas.draw()
    canvas.get_tk_widget().pack(expand=True,
fill='both')
```

- Time-Series Graphs: Creates line plots to show trends over time for Wind Power Generation, Temperature, and Wind Speed, split by Region. Uses a shared x-axis for time and different color palettes for each plot.

Main Application Window

```
root = tk.Tk()
root.title("Wind Power Data Analysis")
root.geometry("1200x800")

notebook = ttk.Notebook(root)
notebook.pack(expand=True, fill='both')

tab_original = ttk.Frame(notebook)
tab_cleaned = ttk.Frame(notebook)
tab_summary = ttk.Frame(notebook)
tab_graphs = ttk.Frame(notebook)
tab_time_series = ttk.Frame(notebook)

notebook.add(tab_original, text="Original DataFrame")
notebook.add(tab_cleaned, text="Cleaned DataFrame")
notebook.add(tab_summary, text="Summary Statistics")
notebook.add(tab_graphs, text="Distribution Graphs")
notebook.add(tab_time_series, text="Time-Series Graphs")

display_dataframe(tab_original, df)
display_dataframe(tab_cleaned, df)
summary_statistics(tab_summary, df)
distribution_graphs(tab_graphs, df)
time_series_graphs(tab_time_series, df)

root.mainloop()
```

- Main Window: Creates the main Tkinter window with a title and size.
- Notebook: Adds a tabbed interface to the main window using ttk.Notebook.
- Tabs: Adds five tabs for displaying different views (Original DataFrame, Cleaned DataFrame, Summary Statistics, Distribution Graphs, Time-Series Graphs).
- Display Functions: Calls functions to populate the tabs with the DataFrame, summary statistics, and graphs.
- Mainloop: Starts the Tkinter event loop to display the GUI and handle user interactions.

This comprehensive code structure provides a robust and interactive interface for analyzing and visualizing synthetic wind power generation data.

| Wind Power Data Analysis | | | | |
|--------------------------|-------------------|-----------------------|---------------------|--------------------|
| Original DataFrame | Cleaned DataFrame | Summary Statistics | Distribution Graphs | Time-Series Graphs |
| Date | Region | Wind_Power_Generation | Temperature | Wind_Speed |
| 2023-01-01 00:00:00 | North | 179.8573303651471 | 13.86359959621213 | 76.9448894082047 |
| 2023-01-02 00:00:00 | North | 287.67857660247904 | -8.112194945576368 | 39.78657030174272 |
| 2023-01-03 00:00:00 | North | 179.8573303651471 | 33.58119949917427 | 82.75530522740469 |
| 2023-01-04 00:00:00 | North | 179.8573303651471 | 25.942138624455858 | 17.070836331248017 |
| 2023-01-05 00:00:00 | North | 179.8573303651471 | 3.176991651543414 | 3.0463686308960125 |
| 2023-01-06 00:00:00 | North | 88.99863008405066 | 34.098664821148276 | 20.444837246427305 |
| 2023-01-07 00:00:00 | North | 64.52090304204987 | 17.084671533634495 | 34.041155611425984 |
| 2023-01-08 00:00:00 | North | 266.54403644373383 | 16.209019619326092 | 51.05807574595599 |
| 2023-01-09 00:00:00 | North | 200.2787529358022 | 23.663293074328948 | 71.05932373369907 |
| 2023-01-10 00:00:00 | North | 227.01814444901137 | 26.529640464056193 | 91.0605717214217 |
| 2023-01-11 00:00:00 | North | 55.14612357395061 | 19.54153732991762 | 50.98540095504577 |
| 2023-01-12 00:00:00 | North | 292.4774630404986 | -4.235691413520999 | 50.127564462572934 |
| 2023-01-13 00:00:00 | North | 258.11066020010543 | 5.222037769752427 | 5.024300982298213 |
| 2023-01-14 00:00:00 | North | 103.08477766956904 | 31.763763814672238 | 3.49115526562381 |
| 2023-01-15 00:00:00 | North | 95.45624180177515 | 0.10770253297333632 | 55.11599619163721 |
| 2023-01-16 00:00:00 | North | 95.85112746335845 | 0.10770253297333632 | 43.81786132483445 |
| 2023-01-17 00:00:00 | North | 126.06056073988444 | 9.44349732488819 | 83.91803949802986 |
| 2023-01-18 00:00:00 | North | 181.18910790805947 | 9.77322453625035 | 16.067958471184518 |
| 2023-01-19 00:00:00 | North | 157.98625466052894 | 17.582281212253648 | 2.49716621170496 |
| 2023-01-20 00:00:00 | North | 122.80728504951048 | 32.43841269851184 | 44.903876793012586 |
| 2023-01-21 00:00:00 | North | 202.96322368059487 | 0.8311720324627228 | 23.74785567522575 |
| 2023-01-22 00:00:00 | North | 84.87346516301045 | -4.532438066458687 | 4.96143733883745 |
| 2023-01-23 00:00:00 | North | 123.03616213380454 | -1.1138279563028206 | 8.060368664804246 |
| 2023-01-24 00:00:00 | North | 141.59046082342292 | 29.91162050417126 | 11.159299995724748 |
| 2023-01-25 00:00:00 | North | 164.01749605425897 | 19.06148656948427 | 60.83195000077717 |
| 2023-01-26 00:00:00 | North | 246.2939903482534 | 2.86580515762206 | 28.104975016688915 |
| 2023-01-27 00:00:00 | North | 99.91844553958992 | 26.717612253587774 | 17.3496050355022 |
| 2023-01-28 00:00:00 | North | 179.8573303651471 | 28.76164983635462 | 37.97515229193175 |
| 2023-01-29 00:00:00 | North | 198.1036422155106 | 28.09314524106251 | 31.61852163892758 |
| 2023-01-30 00:00:00 | North | 179.8573303651471 | 31.351694013779948 | 25.261890985923408 |
| 2023-01-31 00:00:00 | North | 201.8862129753596 | 1.350845907801272 | 18.905260332919237 |

Wind Power Data Analysis

| Original DataFrame | Cleaned DataFrame | Summary Statistics | Distribution Graphs | Time-Series Graphs |
|---------------------|-------------------|-----------------------|---------------------|--------------------|
| Date | Region | Wind_Power_Generation | Temperature | Wind_Speed |
| 2023-01-01 00:00:00 | North | 179.8573303651471 | 13.863595996212123 | 76.9448894082047 |
| 2023-01-02 00:00:00 | North | 287.67857660247904 | -8.12194945576368 | 39.78657030174272 |
| 2023-01-03 00:00:00 | North | 179.8573303651471 | 33.58199499174727 | 82.75530522740469 |
| 2023-01-04 00:00:00 | North | 179.8573303651471 | 25.942138624455858 | 17.070836331248017 |
| 2023-01-05 00:00:00 | North | 179.8573303651471 | 3.17691651543414 | 3.0463686308960125 |
| 2023-01-06 00:00:00 | North | 88.99863008405066 | 34.098664821148276 | 20.444837246427305 |
| 2023-01-07 00:00:00 | North | 64.5209304204987 | 17.084671533634495 | 34.041155611425964 |
| 2023-01-08 00:00:00 | North | 266.54403644373383 | 16.209019619326092 | 51.05807574595599 |
| 2023-01-09 00:00:00 | North | 200.2787529358022 | 23.663293074328948 | 71.05932373369907 |
| 2023-01-10 00:00:00 | North | 227.01814444901137 | 26.529640464056193 | 91.06057172144217 |
| 2023-01-11 00:00:00 | North | 55.14612357395061 | 19.54153732991762 | 50.98540095504577 |
| 2023-01-12 00:00:00 | North | 292.4774630404986 | -4.235691413520999 | 50.127564462572934 |
| 2023-01-13 00:00:00 | North | 258.11066020010543 | 5.222037769752427 | 5.024300982298213 |
| 2023-01-14 00:00:00 | North | 103.08477766956904 | 31.763763814672238 | 3.491155265622381 |
| 2023-01-15 00:00:00 | North | 95.45624180177515 | 0.10770253297333632 | 55.11599619163721 |
| 2023-01-16 00:00:00 | North | 95.8511746335845 | 0.10770253297333632 | 43.81786132483445 |
| 2023-01-17 00:00:00 | North | 126.06056073984844 | 9.44345973248819 | 83.91803949802986 |
| 2023-01-18 00:00:00 | North | 181.18910790805947 | 9.7732245625035 | 16.067958471184518 |
| 2023-01-19 00:00:00 | North | 157.98625466052894 | 17.58228121253648 | 2.4971662170496 |
| 2023-01-20 00:00:00 | North | 122.80728504951048 | 32.43841269851184 | 44.903876793012586 |
| 2023-01-21 00:00:00 | North | 202.96322368059487 | 0.8311720324627228 | 23.74785567522575 |
| 2023-01-22 00:00:00 | North | 84.87346516301045 | -4.53243806458687 | 4.96143733883745 |
| 2023-01-23 00:00:00 | North | 123.03616213380454 | -1.1138279563028206 | 8.060368664804246 |
| 2023-01-24 00:00:00 | North | 141.59046082342292 | 29.911620509417126 | 11.159299995724748 |
| 2023-01-25 00:00:00 | North | 164.01749605425897 | 19.06148656948427 | 60.83195000077717 |
| 2023-01-26 00:00:00 | North | 246.2939903482534 | 2.865805515762206 | 28.104975016688915 |
| 2023-01-27 00:00:00 | North | 99.91844553958992 | 26.71761225358774 | 17.3496050355022 |
| 2023-01-28 00:00:00 | North | 179.8573303651471 | 28.76164983635462 | 37.97515229193175 |
| 2023-01-29 00:00:00 | North | 198.1036422155106 | 28.09314524106251 | 31.61852163892758 |
| 2023-01-30 00:00:00 | North | 179.8573303651471 | 31.351694013779948 | 25.261890985923408 |
| 2023-01-31 00:00:00 | North | 201.8862129753596 | 1.350845907801272 | 18.905260332919237 |

Wind Power Data Analysis

| Original DataFrame | Cleaned DataFrame | Summary Statistics | Distribution Graphs | Time-Series Graphs | | | | | |
|--------------------|-------------------|--------------------|---------------------|--------------------|-----------------------|---------------------|---------------------|---------------------|---------------------|
| Statistic | count | unique | top | freq | mean | min | 25% | 50% | 75% |
| Date | 1825 | nan | nan | nan | 2023-07-02 00:00:00.0 | 2023-01-01 00:00:00 | 2023-04-02 00:00:00 | 2023-07-02 00:00:00 | 2023-10-01 00:00:00 |
| Region | 1825 | 5 | North | 365 | nan | nan | nan | nan | nan |
| Wind_Power_Generat | 1825.0 | nan | nan | nan | 174.370134845119 | 51.15800575115072 | 114.95330218625855 | 174.09156180030908 | 230.61302881537634 |
| Temperature | 1825.0 | nan | nan | nan | 12.44622928935024 | -9.999476436008523 | 1.6420238969948429 | 12.51617507970023 | 23.429667215057258 |
| Wind_Speed | 1825.0 | nan | nan | nan | 49.26466603845087 | 0.00307188453824158 | 25.44813161826025 | 48.329428496632524 | 73.02491895295712 |

