Pandas Data Analytics Cheat Sheet

Anis Koubaa

October 2024

Companion Notebook

For more detailed examples, visit the Google Colab notebook at: Jupyter Notebook for Pandas Data Analytics.

1. Data Loading

- **Reading Files**:
  - `pd.read_csv('filename.csv')` - Load data from a CSV file.
  - `pd.read_excel('filename.xlsx')` - Load data from an Excel file.
  - `pd.read_sql(query, connection_object)` - Load data from a SQL table/database.
  - `pd.read_json('filename.json')` - Load data from a JSON formatted string, URL or file.
  - `pd.read_html('filename.html')` - Parses an html URL, string or file and extracts tables to a list of dataframes.

2. Data Cleaning

- **Handling Missing Data**:
  - `df.dropna()` - Drop rows with any column having NA/null data.
  - `df.fillna(value)` - Replace all NA/null data with value.
- **Drop Duplicates**:
  - `df.drop_duplicates()` - Drop duplicate rows.
- **Data Type Conversion**:
  - `df.astype(dtype)` - Convert the data type of a series.
  - `pd.to_numeric(df['column'], errors='coerce')` - Convert data to numeric type, coerce errors.

3. Data Manipulation

- **Column Operations**:
  - `df['new_column'] = df['column1'] + df['column2']` - Create a new column based on existing columns.
  - `df.rename(columns={'old_name': 'new_name'})` - Rename columns.
- **Row Operations**:
  - `df.sort_values('column')` - Sort the dataframe by column.
  - `df.query('expression')` - Filter rows under condition.
- **Aggregation**:
  - `df.groupby('column').mean()` - Group by column and return the mean of each group.
  - `df.pivot_table(values='column_to_aggregate', index='row_index', columns='column_index', aggfunc=np.mean)` - Create a pivot table.

4. Data Visualization

- **Basic Plotting**:
  - `df.plot(kind='line')` - Line plot of all columns.
  - `df.plot(kind='bar')` - Bar plot of all columns.
  - `df.plot(kind='scatter', x = 'feature1' , y = 'feature2')` - Scatter plot of two features.
  - `df['column'].hist()` - Histogram of a column.
  - `df['column'].plot(kind='pie')` - Pie chart of a column's categories.
  - `df.plot(kind='area')` - Area plot for all columns.
- **Advanced Plotting**:
  - `df.boxplot(column=['col1', 'col2'])` - Box plot of one or more columns.
  - `pd.plotting.scatter_matrix(df)` - Scatter matrix of DataFrame columns.
  - `sns.heatmap(df.corr(), annot=True, cmap='coolwarm')` - Heatmap of the correlation matrix.
  - `sns.pairplot(df)` - Pairwise plots of DataFrame's variables.
  - `df['column1'].plot(kind='kde')` - Kernel Density Estimate plot for a column.

5. Complex Operations

- **Merging and Joining**:
  - `pd.merge(df1, df2, on='key_column')` - SQL-like join operation.
  - `df1.join(df2, on='key_column', how='left')` - Join matching rows from df2 to df1.
- **Handling Time Series**:
  - `pd.to_datetime(df['date_column'])` - Convert the column to datetime.
  - `df.resample('M').mean()` - Resample time-series data.
- **Window Functions**:
  - `df.rolling(window=5).mean()` - Moving average over a window.
  - `df.expanding(min_periods=1).sum()` - Cumulative sum.

Additional Tips

- **Efficient Practices**:
  - Use `inplace=True` in functions to modify the DataFrame in place.
  - Utilize vectorized operations with NumPy for performance-efficient computations.
- **Miscellaneous**:
  - `df.describe()` - Generate descriptive statistics.
  - `df.info()` - Summary of DataFrame including the index dtype and columns, non-null values, and memory usage.

1 Introduction

This document demonstrates the use of the pandas library in Python to load, clean, manipulate, and visualize data from the heart disease dataset obtained from the UCI Machine Learning Repository.

Sample Data

Below is a sample of patient data collected in a study on heart disease. We will use variations of this dataset to demonstrate merging and joining operations.

Age	Sex	CP	Trestbps	Chol	FBS	Restecg	Thalach	Exang	Oldpeak	Slope	CA	Thal	Target
63	1	1	145	233	1	2	150	0	2.3	3	0	6	0
67	1	4	160	286	0	2	108	1	1.5	2	3	3	2
67	1	4	120	229	0	2	129	1	2.6	2	2	7	1
37	1	3	130	250	0	0	187	0	3.5	3	0	3	0
41	0	2	130	204	0	2	172	0	1.4	1	0	3	0

2 Data Loading

```
import pandas as pd

# URL of the heart disease dataset (assuming it's named 'data.csv' locally)
url = 'data.csv'

# Define the column names since the dataset does not contain headers
column_names = ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg',
                'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'target']

# Load the data into a DataFrame
heart_disease_data = pd.read_csv(url, header=None, names=column_names)

# Preview the data
print(heart_disease_data.head())
```

3 Data Exploration and Summary Statistics

This section demonstrates how to use pandas to explore and summarize the characteristics of a dataset, providing insights into the data's structure, content, and missing values.

3.1 Exploring Data Structure with `info()`

The `info()` method provides a concise summary of a DataFrame, including the number of non-null entries for each column, the data type of each column, and the memory usage.

```
# Get information about the DataFrame structure
heart_disease_data.info()
```

3.2 Summarizing Data Characteristics with `describe()`

The `describe()` method generates descriptive statistics that summarize the central tendency, dispersion, and shape of a dataset's distribution, excluding NaN values.

```
# Generate descriptive statistics
heart_disease_data.describe()
```

3.3 Checking for Missing Values

3.3.1 Using `isnull()` and `sum()`

To identify missing values in the dataset, `isnull()` can be used to create a Boolean series where each cell is True if data is missing. Combining `isnull()` with `sum()` provides a count of missing values per column.

```
# Count missing values in each column
heart_disease_data.isnull().sum()
```

3.4 Additional Summary Functions

- **Counting Values:** The `value_counts()` method counts the number of unique entries in a column, with the ability to normalize the results to show relative frequencies.

```
    # Count occurrences of unique values in the 'cp' column
    heart_disease_data['cp'].value_counts()

```

- **Checking for Duplicates:** The `duplicated()` function can be used to check for duplicate rows, returning a Boolean series.

```
    # Check for duplicate rows
    heart_disease_data.duplicated().sum()

```

- **Checking Unique Values:** The `nunique()` function returns the number of unique values for each column.

```
    # Get the number of unique values in each column
    heart_disease_data.nunique()

```

Each of these functions plays a crucial role in the initial stages of data analysis, helping to understand the amount of information, check for data integrity, and prepare the dataset for further analysis.

4 Data Cleaning

This section covers different data cleaning techniques including filling and removing missing values from the dataset using the pandas library.

4.1 Handling Missing Values

- **Filling Missing Values:**

```
# Fill all missing values with zero
heart_disease_data.fillna(0, inplace=True)

# Fill missing values with the mean of the respective column
heart_disease_data.fillna(heart_disease_data.mean(), inplace=True)

# Calculate the median of the 'trestbps' column
median_trestbps = heart_disease_data['trestbps'].median()

# Fill missing values in 'trestbps' column with its median
heart_disease_data['trestbps'].fillna(median_trestbps, inplace=True)

# Fill missing values forward (propagate last valid observation forward)
heart_disease_data.fillna(method='ffill', inplace=True)

# Fill missing values backward (use next valid observation to fill gap)
heart_disease_data.fillna(method='bfill', inplace=True)

```

- **Dropping Missing Values:**

```
# Drop rows where any cell in that row is NA
heart_disease_data.dropna(inplace=True)

# Drop rows where all cells in that row are NA
heart_disease_data.dropna(how='all', inplace=True)

# Drop columns where any cell in that column is NA
heart_disease_data.dropna(axis=1, inplace=True)

# Drop rows where NA values appear in specific columns (e.g., 'cp' or 'thal
')
heart_disease_data.dropna(subset=['cp', 'thal'], inplace=True)

```

4.2 Handling Duplicate Entries

- **Dropping Duplicate Rows:**

```
# Drop duplicate rows
heart_disease_data.drop_duplicates(inplace=True)

# Drop duplicates in a subset of columns and keep the last occurrence
heart_disease_data.drop_duplicates(subset=['age', 'cp'], keep='last',
inplace=True)

```

4.3 Data Type Conversion

To ensure data integrity, it is sometimes necessary to explicitly convert the data types of certain columns. This might involve changing data types to accommodate mathematical operations, align with modeling requirements, or standardize input formats.

```
# Convert data type of a column to float
heart_disease_data['age'] = heart_disease_data['age'].astype(float)

# Convert data type of a column to integer
heart_disease_data['target'] = heart_disease_data['target'].astype(int)

# Use pd.to_numeric to convert the 'chol' column to numeric, coercing errors
heart_disease_data['chol'] = pd.to_numeric(heart_disease_data['chol'], errors='
    coerce')
```

3. Data Manipulation

This section details practical examples of manipulating data using pandas, including operations on columns, rows, and groups.

Column Operations

- **Creating a New Column:**

```
    # Create a new column as the sum of two existing columns
    heart_disease_data['new_column'] = heart_disease_data['age'] +
    heart_disease_data['trestbps']

```

- **Renaming Columns:**

```
    # Rename columns to more descriptive names
    heart_disease_data.rename(columns={'cp': 'chest_pain_type', 'trestbps': '
    resting_bp'}, inplace=True)

```

Row Operations

- **Sorting Data:**

```
    # Sort the DataFrame by the 'age' column in descending order
    heart_disease_data.sort_values('age', ascending=False, inplace=True)

```

- **Filtering Rows:**

```
    # Filter rows where 'age' is greater than 50 and 'sex' is 1
    filtered_data = heart_disease_data.query('age > 50 & sex == 1')

```

Aggregation

- **Grouping and Aggregating:**

```
    # Group by 'sex' and calculate the mean of each group for 'cholesterol'
    average_cholesterol_per_sex = heart_disease_data.groupby('sex')['chol'].
    mean()

```

- **Creating a Pivot Table:**

```
# Create a pivot table that calculates the average 'thalach' for each 'sex' and 'cp' combination
pivot_table = heart_disease_data.pivot_table(values='thalach', index='sex', columns='cp', aggfunc='mean')

```

Merging and Joining

- **SQL-like Join Operation:**

```
import pandas as pd

# Assume df1 and df2 are derived from the main dataset with modifications
df1 = data[['age', 'sex', 'cp', 'target']].copy()
df2 = data[['cp', 'thalach', 'exang']].drop_duplicates().copy()

# Merge two dataframes on a 'cp' (chest pain type) column
merged_data = pd.merge(df1, df2, on='cp')

```

Description: This function merges two DataFrames based on the 'cp' column, akin to a SQL join operation. This can merge detailed patient data with exercise-related outcomes, assuming 'cp' acts as a unique key for these exercises.

- **Join Matching Rows from Another DataFrame:**

```
# Join df2 to df1 based on the 'cp' column which is set as an index in df2
df1.set_index('cp', inplace=True)
df2.set_index('cp', inplace=True)
joined_data = df1.join(df2, how='left')

```

Description: This left join appends columns from df2 to df1 based on matching 'cp' values. It's useful for enriching patient records with corresponding exercise data without duplicating records.

4. Data Visualization

This section demonstrates various plotting techniques using pandas and seaborn to explore and represent data visually.

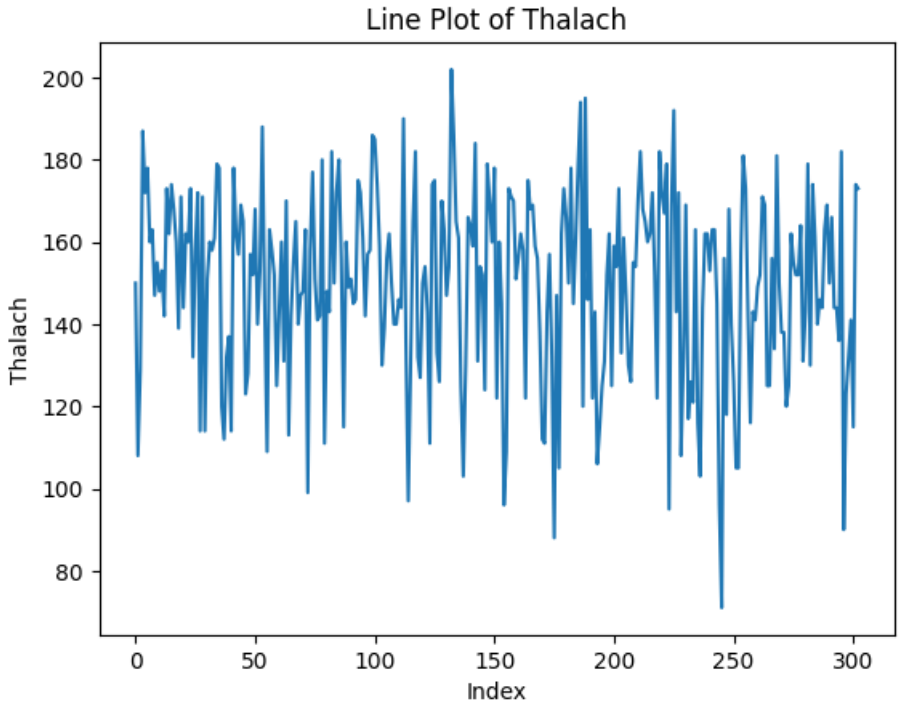
Basic Plotting

- **Line Plot:**

```
# Plot 'thalach' (maximum heart rate achieved) over the index
import matplotlib.pyplot as plt
heart_disease_data['thalach'].plot(kind='line')
plt.xlabel('Index')
plt.ylabel('Thalach')
plt.show()

```

Description: A line plot is useful for showing trends over a period of time or ordinal categories. Expect to see how 'thalach' varies across the index, potentially indicating trends or patterns in heart rate data across observations.

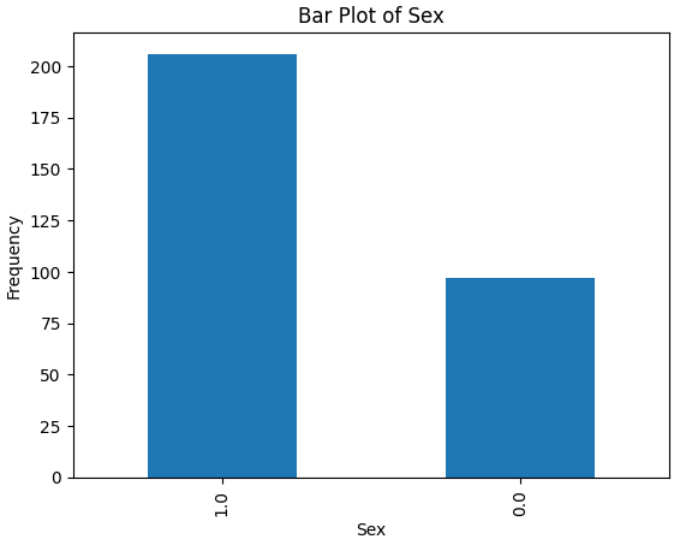


- **Bar Plot:**

```
# Bar plot of 'sex' column to count occurrences of each category
data['sex'].value_counts().plot(kind='bar', title='Bar Plot of Gender')
plt.xlabel('Gender')
plt.ylabel('Frequency')
plt.show()

```

Description: Bar plots help in comparing quantities corresponding to different groups. Expect to see the distribution of data between different genders, highlighting potential imbalances or differences in sample sizes.

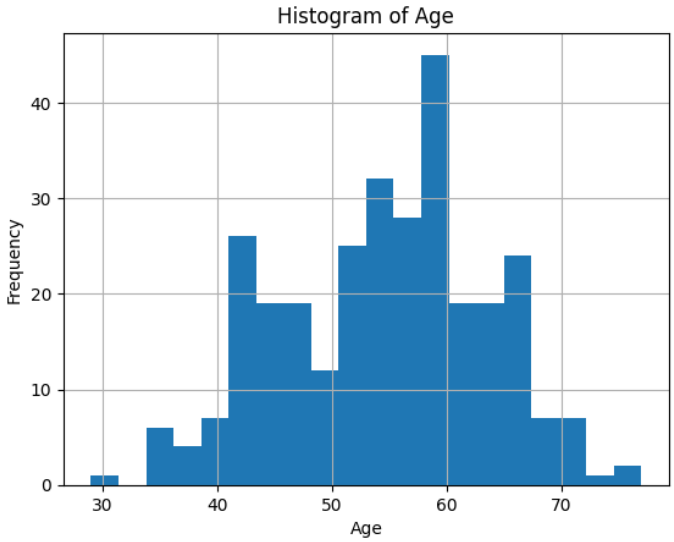


- **Histogram:**

```
# Histogram of 'age' to see the distribution of ages
heart_disease_data['age'].hist(bins=20)
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()

```

Description: Histograms are great for visualizing the distribution of numerical data. Expect to see the frequency distribution of ages, which helps in understanding the age composition of the subjects.

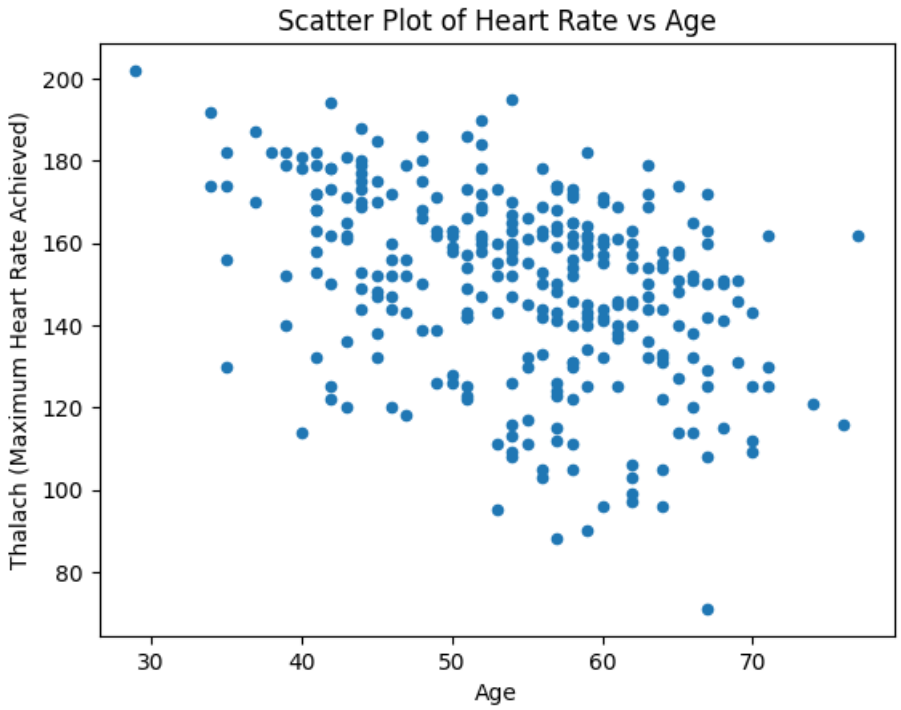


– **Scatter Plot:**

```
# Plot 'thalach' vs 'age' (maximum heart rate achieved vs age)
import matplotlib.pyplot as plt
heart_disease_data.plot(kind='scatter', x='age', y='thalach')
plt.xlabel('Age')
plt.ylabel('Thalach (Maximum Heart Rate Achieved)')
plt.title('Scatter Plot of Heart Rate vs Age')
plt.show()

```

Description: A scatter plot is ideal for showing the relationship between two numerical variables. This plot of 'thalach' versus 'age' helps to visualize how maximum heart rate achieved varies with age, which can be useful for identifying patterns or correlations between these variables.

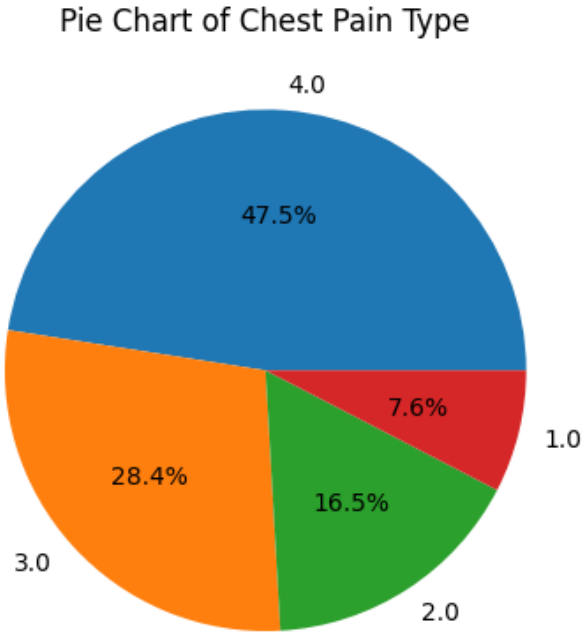


- **Pie Chart:**

```
# Pie chart of 'cp' (chest pain type)
heart_disease_data['cp'].value_counts().plot(kind='pie')
plt.ylabel('')  # Remove the y-label as it's unnecessary for pie charts
plt.show()

```

Description: Pie charts are effective for showing relative proportions or percentages of categories. Expect to see how different types of chest pain contribute to the overall dataset.

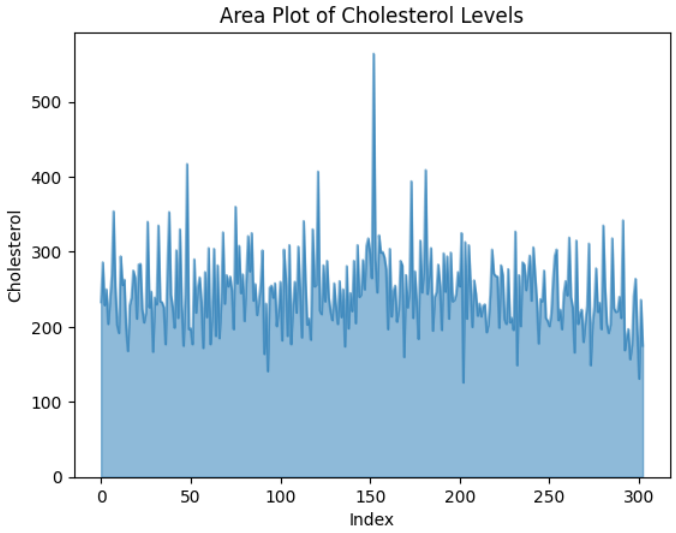


- **Area Plot:**

```
    # Area plot for 'chol' (cholesterol levels) over the index
    heart_disease_data['chol'].plot(kind='area', alpha=0.5)
    plt.xlabel('Index')
    plt.ylabel('Cholesterol')
    plt.show()

```

Description: Area plots are useful for tracking changes over time among related attributes. Expect to see the area under the curve filled, indicating the magnitude of cholesterol levels through the dataset's range.



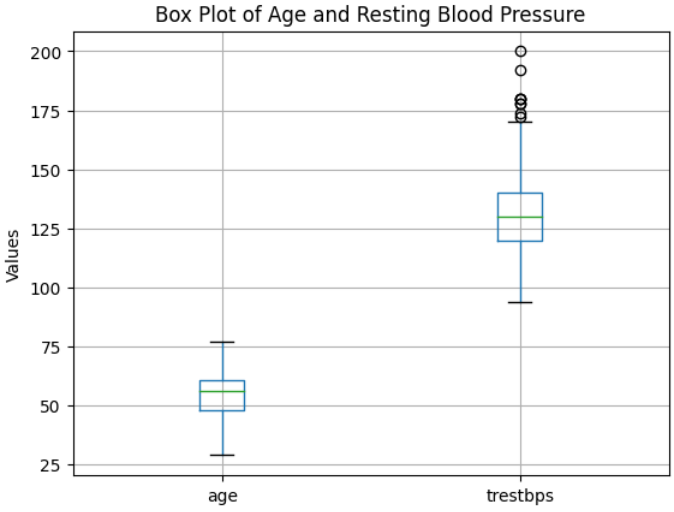
Advanced Plotting

- **Box Plot:**

```
# Box plot for 'age' and 'trestbps' (resting blood pressure)
heart_disease_data.boxplot(column=['age', 'trestbps'])
plt.title('Box Plot of Age and Resting Blood Pressure')
plt.ylabel('Values')
plt.show()

```

Description: Box plots are useful for identifying outliers and understanding the spread of the data. Expect to see the median, quartiles, and outliers for age and resting blood pressure.

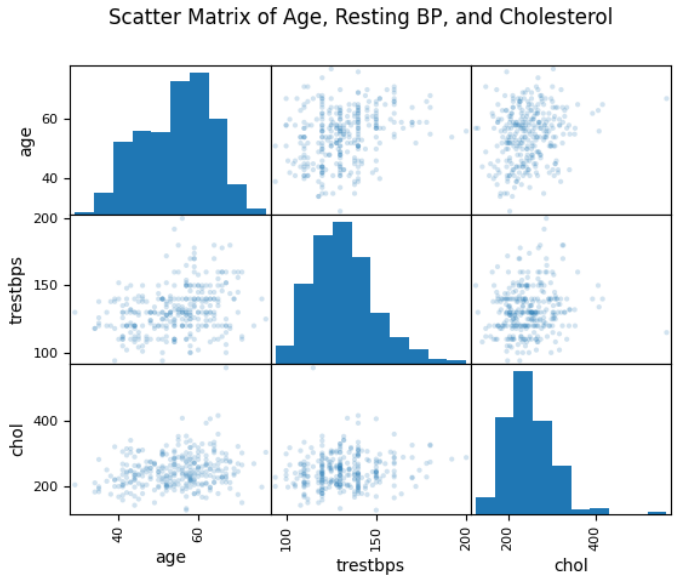


- **Scatter Matrix:**

```
# Scatter matrix of multiple variables
pd.plotting.scatter_matrix(heart_disease_data[['age', 'trestbps', 'chol']],
 alpha=0.2, figsize=(10, 10))
plt.suptitle('Scatter Matrix of Age, Resting BP, and Cholesterol')
plt.show()

```

Description: Scatter matrices allow for the visualization of potential relationships between different numerical variables. Expect to see scatter plots for each pair of variables and histograms on the diagonal.

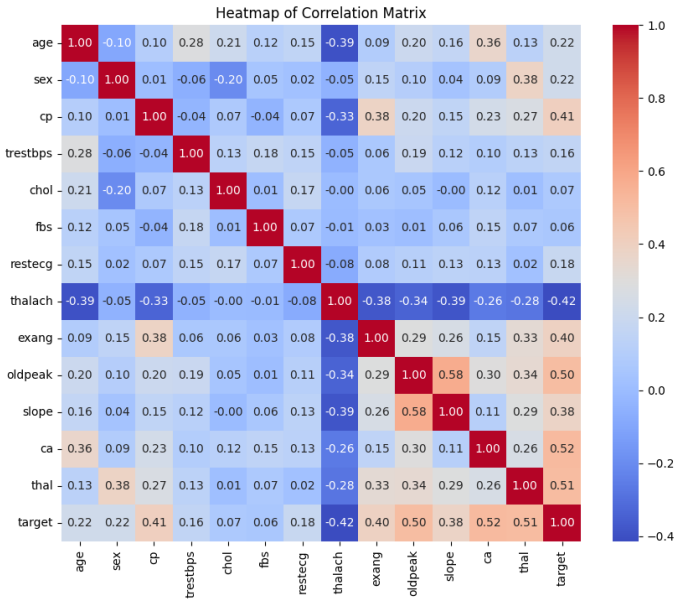


- **Heatmap of Correlation Matrix:**

```
# Heatmap showing correlations between variables
sns.heatmap(heart_disease_data.corr(), annot=True, cmap='coolwarm')
plt.title('Heatmap of Correlation Matrix')
plt.show()

```

Description: Heatmaps help in visualizing the strength of relationships between variables. Look for color-coded correlations, with warmer colors indicating stronger positive relationships and cooler colors indicating negative relationships.

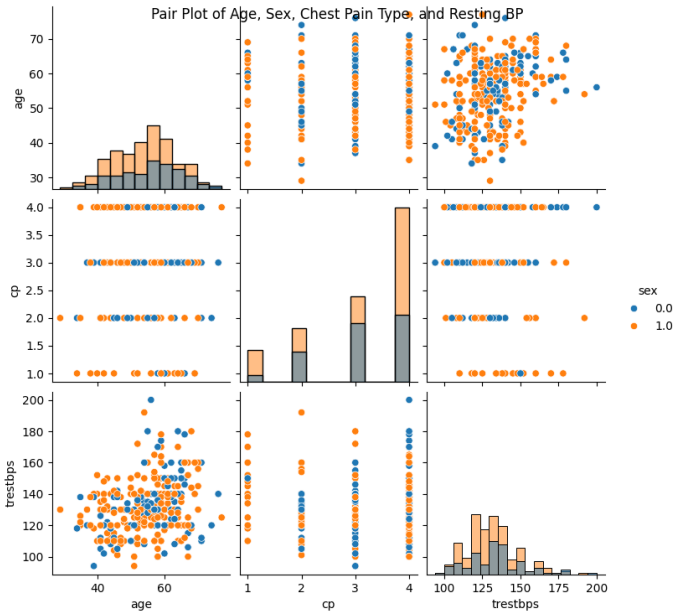


- **Pair Plot:**

```
    # Pairwise plots of DataFrame's variables
    sns.pairplot(heart_disease_data[['age', 'sex', 'cp', 'trestbps']], hue='sex
    ')
    plt.suptitle('Pair Plot of Age, Sex, Chest Pain Type, and Resting BP')
    plt.show()

```

Description: Pair plots are useful for exploring correlations between multiple variables broken down by categories (colored by 'sex'). Expect to see a grid of plots showing relationships between pairs of variables.

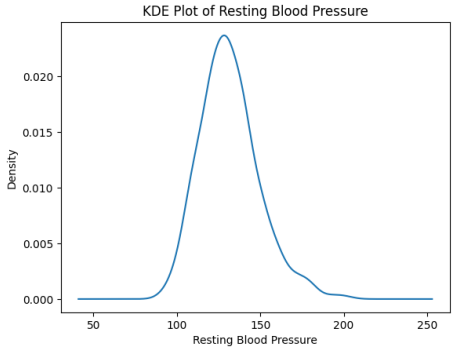


- **Kernel Density Estimate Plot:**

```
# KDE plot for 'trestbps'
heart_disease_data['trestbps'].plot(kind='kde')
plt.xlabel('Resting Blood Pressure')
plt.show()

```

Description: KDE plots are used to estimate the probability density function of a continuous random variable. Expect to see a smoothed version of the histogram, providing insights into the distribution of resting blood pressures.



5 Time Series Operations

Sample Data

Below is a sample of the weather dataset that will be used for the following examples. This data includes hourly recordings of various meteorological parameters.

Year	Month	Day	Hour	Temperature (°C)	Humidity (%)	MSL Pressure (hPa)	Wind Speed (km/h)
2018	11	6	0	21.65	38	1011.7	17.15
2018	11	6	1	20.94	40	1011.4	16.42
2018	11	6	2	20.29	41	1011.2	16.08
2018	11	6	3	19.67	43	1011	14.01
2018	11	6	4	18.76	45	1011.8	10.54
2018	11	6	5	18.25	47	1012.5	9.26
2018	11	6	6	17.68	48	1012.3	8.09
2018	11	6	7	19.63	42	1013.1	6.79

Handling Time Series

- **Converting Columns to Datetime:**

```
# Create a datetime index from the 'Year', 'Month', 'Day', 'Hour' columns
daily_weather['Datetime'] = pd.to_datetime(daily_weather[['Year', 'Month',
'Day', 'Hour']])
daily_weather.set_index('Datetime', inplace=True)

```

Description: Converting these columns into a single datetime column enables effective indexing and manipulation of time series data, essential for time-based analysis.

- **Resampling Time-Series Data:**

```
# Resample the data by month and calculate mean temperature
monthly_avg_temperature = daily_weather['Temperature'].resample('M').mean()

```

Description: Resampling helps summarize daily temperature data into monthly averages, simplifying the data and highlighting longer-term trends.

Window Functions

- **Moving Average:**

```
# Compute a moving average of the temperature over a 5-day window
daily_weather['5_day_temp_avg'] = daily_weather['Temperature'].rolling(
window=5).mean()

```

Description: A moving average smooths out short-term fluctuations and helps to identify longer-term trends in temperature data.

- **Cumulative Sum:**

```
    # Calculate a cumulative sum of wind speed
    daily_weather['cumulative_wind_speed'] = daily_weather['Wind Speed'].
    expanding(min_periods=1).sum()

```

Description: The cumulative sum of wind speed can reveal the total amount of wind experienced over a period, useful for climatological studies and analysis.

6 Matplotlib Visualization

Matplotlib is one of the most popular Python libraries for producing static, animated, and interactive visualizations in Python. This section will cover the basics of creating visualizations using Matplotlib, including line plots, bar charts, histograms, scatter plots, and pie charts.

Line Plots

Line plots are one of the simplest types of plots, useful for displaying trends over time.

```
import matplotlib.pyplot as plt

# Sample data
x = range(10)
y = [xi**2 for xi in x]

# Creating a line plot
plt.figure()
plt.plot(x, y)
plt.title('Simple Line Plot')
plt.xlabel('X Axis Label')
plt.ylabel('Y Axis Label')
plt.show()
```

Bar Charts

Bar charts are great for comparing numerical values across different categories.

```
# Sample data
categories = ['Category 1', 'Category 2', 'Category 3']
values = [10, 20, 15]

# Creating a bar chart
plt.figure()
plt.bar(categories, values)
plt.title('Simple Bar Chart')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.show()
```

Histograms

Histograms are used to summarize the distribution of a dataset.

```
import numpy as np

# Generating random data
data = np.random.randn(1000)

# Creating a histogram
plt.figure()
plt.hist(data, bins=30)
plt.title('Simple Histogram')
plt.xlabel('Data Points')
plt.ylabel('Frequency')
plt.show()
```

Scatter Plots

Scatter plots are used to determine the relationship between two variables.

```
# Generating sample data
x = np.random.rand(50)
y = np.random.rand(50)

# Creating a scatter plot
plt.figure()
plt.scatter(x, y)
plt.title('Simple Scatter Plot')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.show()
```

Pie Charts

Pie charts show the proportions of categories as parts of a whole.

```
# Sample data
labels = ['Part 1', 'Part 2', 'Part 3', 'Part 4']
sizes = [15, 30, 45, 10]

# Creating a pie chart
plt.figure()
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title('Simple Pie Chart')
plt.show()
```

7 Seaborn Visualization

Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. This section covers the basics of creating visualizations using Seaborn, including histograms, bar plots, count plots, box plots, and violin plots.

Histograms (Distplots)

Seaborn's distplot lets you show a histogram with a line on it. This can be used to display the distribution of the data.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Generating random data
data = np.random.randn(1000)

# Creating a distplot
plt.figure()
sns.distplot(data, bins=30, kde=True)
plt.title('Seaborn Histogram with Density Plot')
plt.show()
```

Bar Plots

Bar plots help you visualize the distributions of categorical data.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
categories = ['Category 1', 'Category 2', 'Category 3']
values = [10, 20, 15]

# Creating a bar plot
plt.figure()
sns.barplot(x=categories, y=values)
plt.title('Seaborn Bar Plot')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.show()
```

Count Plots

Count plots can be thought of as histograms across a categorical, instead of quantitative, variable.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample categorical data
data = ['Category 1', 'Category 2', 'Category 1', 'Category 3', 'Category 2', '
    Category 3', 'Category 1']

# Creating a count plot
plt.figure()
sns.countplot(data)
plt.title('Seaborn Count Plot')
plt.xlabel('Categories')
plt.ylabel('Frequency')
plt.show()
```

Box Plots

Box plots show distributions with quartiles and whiskers. They can be used to compare distributions across the categories.

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Sample data
data = np.random.rand(10, 4)
df = pd.DataFrame(data, columns=['A', 'B', 'C', 'D'])

# Creating a box plot
plt.figure()
sns.boxplot(data=df)
plt.title('Seaborn Box Plot')
plt.show()
```

Violin Plots

Violin plots are used to visualize the distribution of the data and its probability density.

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Sample data
data = np.random.rand(10, 4)
df = pd.DataFrame(data, columns=['A', 'B', 'C', 'D'])

# Creating a violin plot
plt.figure()
sns.violinplot(data=df)
plt.title('Seaborn Violin Plot')
plt.show()
```