Summary

Exploratory Data Analysis (EDA) in Python

Exploratory Data Analysis (EDA) is the process of analyzing datasets to summarize their main characteristics, often using visual methods. The goal is to understand the data's structure, identify patterns, spot anomalies, test hypotheses, and check assumptions with the help of summary statistics and graphical representations.

Utility:

EDA is a critical step in the data analysis process as it informs feature selection, hypothesis generation, and the overall modeling process. Python offers several libraries, such as pandas, NumPy, matplotlib, and Seaborn, that facilitate effective EDA.

Keywords:

EDA, data visualization, summary statistics, missing values, outliers, correlation, distribution.

Key Steps in EDA

Loading the Data:

Use pandas to load datasets from various sources (CSV, Excel, SQL, etc.).

Example:

```
import pandas as pd

# Load data from a CSV file
df = pd.read_csv('data.csv')
```

Understanding the Data Structure:

Get a quick overview of the dataset's structure and contents.

Example:

```
# Display the first few rows
print(df.head())

# Get summary of the DataFrame
print(df.info())

# Describe numerical features
print(df.describe())
```

Checking for Missing Values:

Identify and handle missing values in the dataset.

Example:

```
# Check for missing values
missing_values = df.isnull().sum()
print(missing_values)

# Fill or drop missing values
df.fillna(df.mean(), inplace=True)  # Filling missing values with mean
```

Data Visualization:

Use visualizations to understand distributions, relationships, and patterns.

Example:

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

# Histogram for a numeric feature
sns.histplot(df['feature'], bins=30, kde=True)
plt.title('Distribution of Feature')
plt.show()

# Scatter plot for two variables
sns.scatterplot(data=df, x='feature1', y='feature2')
plt.title('Feature1 vs Feature2')
plt.show()
```

Identifying Outliers:

Use box plots or z-scores to detect outliers.

Example:

```
# Box plot to visualize outliers
sns.boxplot(x=df['feature'])
plt.title('Box Plot of Feature')
plt.show()

# Identifying outliers using z-score
from scipy import stats

z_scores = stats.zscore(df['feature'])
outliers = df[(z_scores < -3) | (z_scores > 3)]
```

Exploring Relationships Between Variables:

Analyze correlations and relationships between features.

Example:

```
# Correlation matrix
correlation_matrix = df.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()

# Pair plot for multiple features
sns.pairplot(df)
plt.title('Pair Plot of Features')
plt.show()
```

Feature Engineering:

Create new features based on existing data to improve analysis and modeling.

Example:

```
# Creating a new feature
df['new_feature'] = df['feature1'] / df['feature2']
```

Perform Standard Data Import, Joining, and Aggregation Tasks

Import Data from Flat Files (CSV) into Python

Description: Load data from flat files like CSV into a DataFrame using pandas. This is crucial for working with structured datasets.

Utility: Read CSV files for data analysis or preprocessing tasks.

Keywords: import CSV, read data, pandas DataFrame, flat file, read_csv.

Example:

```
import pandas as pd
data = pd.read_csv('file.csv')
```

Import Data from Databases into Python

Description: Fetch data directly from databases using SQLAlchemy or sqlite3 in Python and convert it to a DataFrame.

Utility: Useful for pulling data from SQL databases for analysis.

Keywords: database import, SQL, fetch data, SQLAlchemy, sqlite3, connection.

Example:

```
import sqlite3
import pandas as pd

conn = sqlite3.connect('database.db')
data = pd.read_sql_query("SELECT * FROM table_name", conn)
```

Aggregate Numeric, Categorical Variables, and Dates by Groups

Description: Use groupby in pandas to aggregate data based on numeric, categorical, or date columns.

Utility: Summarize or aggregate data, often used for statistical reports.

Keywords: groupby, aggregate, sum, mean, aggregate dates, categorical.

Example:

```
# Aggregating numerical columns by a categorical column
data_grouped = data.groupby('category_column')['numeric_column'].sum()
```

Combine Multiple Tables by Rows or Columns

Description: Use concat or merge in pandas to combine data from different sources, either by rows or columns.

Utility: Combine datasets for broader analysis, handling data spread across multiple tables.

Keywords: combine tables, merge, concat, join tables, append.

Example:

```
# Combining by columns
combined = pd.concat([df1, df2], axis=1)

# Combining by rows
combined = pd.concat([df1, df2], axis=0)
```

Filter Data Based on Different Criteria

Description: Use boolean indexing in pandas to filter data based on conditions.

Utility: Select specific subsets of data for targeted analysis.

Keywords: filter, conditional, subset data, query.

Example:

```
# Filtering rows where the value in 'column' is greater than 10
filtered_data = data[data['column'] > 10]
```

Assess Data Quality and Perform Validation Tasks

Identify and Replace Missing Values

Description: Use functions like isnull() or fillna() in pandas to locate and handle missing values in datasets. You can either fill them with a default value or drop them based on the context.

Utility: Ensures data quality by handling missing data appropriately, which is essential for maintaining consistency in analysis.

Keywords: missing values, NaN, null values, fillna, dropna.

Example:

```
import pandas as pd

# Identify missing values

missing_data = data.isnull().sum()


# Replace missing values with the mean of the column

data['column'] = data['column'].fillna(data['column'].mean())
```

Perform Different Types of Data Validation

Description: Validate data for consistency, enforce constraints, ensure values fall within an expected range, and check for uniqueness using conditional statements and built-in pandas methods.

Utility: Guarantees the reliability and validity of the dataset by ensuring that data adheres to specific rules.

Keywords: data validation, range validation, consistency check, constraints, unique values.

Example:

```
# Range validation: ensure values in a column are between 0 and 100
valid_data = data[(data['column'] >= 0) & (data['column'] <= 100)]


# Check for uniqueness in a column
unique_values = data['column'].is_unique
```

Identify and Validate Data Types

Description: Use the dtypes attribute to identify the data types of columns, and astype() or infer_objects() to validate or convert them.

Utility: Ensures that each column has the correct data type, which is essential for applying further analysis or calculations.

Keywords: data type validation, dtypes, astype, infer_objects, validate types.

Example:

```
# Identify data types in the dataset
data_types = data.dtypes


# Convert a column to float type if necessary
data['column'] = data['column'].astype(float)
```

Collect Data from Non-Standard Formats by Modifying Existing Code

Adapt Provided Code to Import Data from an API

Description: Use the requests library in Python to retrieve data from an API and convert it into a DataFrame using pandas.

Utility: Allows you to collect live or real-time data from web APIs, such as financial data, weather reports, or social media metrics.

Keywords: API data, requests, GET request, retrieve API, import JSON.

Example:

```
import requests
import pandas as pd

# Send a GET request to the API
response = requests.get('https://api.example.com/data')

# Convert the JSON response to a DataFrame
data = pd.DataFrame(response.json())
```

Identify the Structure of HTML and JSON Data and Parse Them

Description: Use BeautifulSoup from bs4 to parse HTML and json library to parse JSON data into usable formats for analysis in Python.

Utility: Extract relevant data from HTML pages (web scraping) or process JSON data (common in APIs) for structured data analysis.

Keywords: HTML parsing, JSON parsing, web scraping, BeautifulSoup, json, from_dict.

Example:

```
# Parsing JSON data
import json

# Load JSON data
json_data = '{"name": "John", "age": 30, "city": "New York"}'
parsed_data = json.loads(json_data)
```

```
# Converting JSON data to a DataFrame
data = pd.DataFrame([parsed_data])

# Parsing HTML using BeautifulSoup
from bs4 import BeautifulSoup

html = '<html><body><h1>Hello World</h1></body></html>'
soup = BeautifulSoup(html, 'html.parser')

# Extracting data from HTML
heading = soup.find('h1').text
```

Parsing JSON

Description: The json.loads() method helps convert a JSON object into a Python dictionary for further manipulation.

Utility: Useful for decoding JSON responses from APIs and working with structured data in Python.

Keywords: json, loads, parse JSON, dictionary.

Example:

```
import json

# Parse JSON string
data_dict = json.loads('{"key": "value"}')
```

Calculate Metrics to Report Characteristics of Data and Relationships Between Features

Calculate Measures of Center (Mean, Median, Mode)

Description: Use mean(), median(), and mode() from pandas or scipy to calculate central tendencies in datasets.

Utility: These measures summarize data by indicating the central or typical value for a dataset.

Keywords: mean, median, mode, central tendency, average.

Example:

```
import pandas as pd

from scipy import stats


# Calculate mean, median, and mode

mean_value = data['column'].mean()

median_value = data['column'].median()

mode_value = stats.mode(data['column'])[0]
```

Calculate Measures of Spread (Range, Standard Deviation, Variance)

Description: Use functions like std() and var() from pandas to measure how data is dispersed. Range can be calculated by subtracting the minimum value from the maximum.

Utility: Helps in understanding the variability and distribution of data.

Keywords: range, standard deviation, variance, spread, dispersion.

Example:

```
# Calculate standard deviation, variance, and range
std_dev = data['column'].std()
variance = data['column'].var()
data_range = data['column'].max() - data['column'].min()
```

Calculate Skewness

Description: Use skew() from scipy.stats or pandas to determine the asymmetry of data distributions.

Utility: Skewness helps identify whether the data distribution leans more towards higher or lower values.

Keywords: skewness, distribution, asymmetry, left-skew, right-skew.

Example:

```
from scipy.stats import skew

# Calculate skewness
skewness_value = skew(data['column'])
```

Calculate Correlation Between Variables

Description: Use the corr() function in pandas to calculate Pearson correlation coefficients between two or more variables.

Utility: Correlation indicates the strength and direction of linear relationships between variables.

Keywords: correlation, correlation coefficient, relationship between variables, Pearson.

Example:

```python
# Calculate correlation between two variables
correlation_value = data['column1'].corr(data['column2'])

# Calculate correlation matrix for the entire dataset
correlation_matrix = data.corr()
```

Create Data Visualizations in Python to Demonstrate Data Characteristics

Create and Customize Bar Charts

Description: Use matplotlib or seaborn to create bar charts, representing categorical data with rectangular bars.

Utility: Bar charts are great for comparing quantities across categories.

Keywords: bar chart, categorical data, bar width, bar height, matplotlib, seaborn.

Example:

```
import matplotlib.pyplot as plt

# Create a bar chart
plt.bar(data['category_column'], data['value_column'])

# Customize the bar chart
plt.title('Bar Chart Example')
plt.xlabel('Category')
plt.ylabel('Value')
plt.show()
```

Create and Customize Box Plots

Description: Use seaborn or matplotlib to generate box plots that visualize the distribution, including medians, quartiles, and outliers.

Utility: Box plots are ideal for displaying the spread and identifying outliers in continuous data.

Keywords: box plot, outliers, quartiles, distribution, seaborn.

Example:

```
import seaborn as sns

# Create a box plot
sns.boxplot(x='category_column', y='value_column', data=data)

# Customize the box plot
plt.title('Box Plot Example')
plt.show()
```

Create and Customize Line Graphs

Description: Use matplotlib or seaborn to create line graphs, ideal for showing trends over time or continuous data.

Utility: Line graphs help in analyzing trends, particularly for time-series data.

Keywords: line graph, time series, trend, continuous data, matplotlib.

Example:

```
# Create a line graph
plt.plot(data['date_column'], data['value_column'])

# Customize the line graph
plt.title('Line Graph Example')
plt.xlabel('Date')
plt.ylabel('Value')
plt.show()
```

Create and Customize Histograms

Description: Use matplotlib or seaborn to create histograms that represent the distribution of continuous data.

Utility: Histograms display the frequency of data within specified ranges.

Keywords: histogram, distribution, frequency, bins, seaborn.

Example:

```
# Create a histogram
plt.hist(data['value_column'], bins=10)

# Customize the histogram
plt.title('Histogram Example')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

Create Data Visualizations in Python to Represent Relationships Between Features

Create and Customize Scatterplots

Description: Use matplotlib or seaborn to create scatterplots that show relationships between two continuous variables.

Utility: Scatterplots reveal correlations and patterns between variables.

Keywords: scatterplot, relationship, correlation, continuous data, seaborn.

Example:

```
# Create a scatterplot
plt.scatter(data['column1'], data['column2'])
```

```
# Customize the scatterplot
plt.title('Scatterplot Example')
plt.xlabel('Variable 1')
plt.ylabel('Variable 2')
plt.show()
```

Create and Customize Heatmaps

Description: Use seaborn to create heatmaps that visualize the correlation between variables or intensity of values.

Utility: Heatmaps are useful for analyzing the correlation matrix or frequency distributions.

Keywords: heatmap, correlation, intensity, seaborn.

Example:

```
# Create a heatmap for a correlation matrix
sns.heatmap(data.corr(), annot=True, cmap='coolwarm')

# Customize the heatmap
plt.title('Heatmap Example')
plt.show()
```

Create and Customize Pivot Tables

Description: Use pandas to create pivot tables for summarizing and aggregating data. Visualize the output using seaborn or matplotlib.

Utility: Pivot tables are ideal for summarizing categorical data.

Keywords: pivot table, summarize data, categorical data, aggregation, pandas.

Example:

```
# Create a pivot table

pivot_table = data.pivot_table(index='category_column', values='value_column', aggfunc='sum')


# Plot the pivot table data

pivot_table.plot(kind='bar')

plt.title('Pivot Table Example')

plt.show()
```

Data Visualization with Seaborn in Python

Description:

Seaborn is a powerful Python visualization library based on Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics. It simplifies the creation of complex visualizations and offers built-in themes and color palettes.

Utility:

Seaborn is particularly useful for visualizing data distributions, relationships between variables, and categorical data. Its integration with pandas allows for easy plotting directly from DataFrames.

Keywords:

data visualization, Seaborn, scatter plot, bar plot, box plot, heatmap, pair plot, catplot.

Common Visualization Types in Seaborn

Scatter Plot:

Description: Displays values for typically two variables for a set of data.

Example:

```
import seaborn as sns

import matplotlib.pyplot as plt


# Sample dataset

tips = sns.load_dataset('tips')
```

```
# Create a scatter plot
sns.scatterplot(data=tips, x='total_bill', y='tip', hue='time', style='time')
plt.title('Scatter Plot of Total Bill vs Tip')
plt.show()
```

Bar Plot:

Description: Shows the relationship between a categorical variable and a continuous variable.

Example:

```
# Create a bar plot
sns.barplot(data=tips, x='day', y='total_bill', estimator=sum)
plt.title('Total Bill by Day')
plt.show()
```

Box Plot:

Description: Summarizes the distribution of a dataset through its quartiles.

Example:

```
# Create a box plot
sns.boxplot(data=tips, x='day', y='total_bill', palette='Set2')
plt.title('Box Plot of Total Bill by Day')
plt.show()
```

Histogram:

Description: Shows the distribution of a single continuous variable.

Example:

```
# Create a histogram
sns.histplot(data=tips, x='total_bill', bins=20, kde=True)
plt.title('Histogram of Total Bill')
plt.show()
```

Heatmap:

Description: Visualizes data through variations in color, typically used for correlation matrices.

Example:

```
# Calculate the correlation matrix
corr = tips.corr()

# Create a heatmap
sns.heatmap(corr, annot=True, cmap='coolwarm', square=True)
plt.title('Heatmap of Correlation Matrix')
plt.show()
```

Pair Plot:

Description: Displays pairwise relationships in a dataset.

Example:

Create a pair plot

sns.pairplot(tips, hue='time')

plt.title('Pair Plot of Tips Dataset')

plt.show()

Facet Grid:

Description: A grid of subplots based on a categorical variable, useful for visualizing the distribution of a variable across different categories.

Example:

```
# Create a facet grid
g = sns.FacetGrid(tips, col='time')
g.map(sns.histplot, 'total_bill')
plt.subplots_adjust(top=0.8)
g.fig.suptitle('Total Bill Distribution by Time')
plt.show()
```

Customization Options

Color Palettes: Seaborn provides various color palettes that can be easily applied to your plots.

```
sns.set_palette('husl')  # Set a color palette
```

Themes: You can change the overall style of your plots using:

```
sns.set_style('whitegrid')  # Options: darkgrid, whitegrid, dark, white, ticks
```

Joining Data with pandas in Python

Description:

Use pandas functions like merge(), join(), and concat() to combine two or more dataframes. The method of joining depends on the structure and requirements of the data (e.g., inner, outer, left, right joins). This allows you to combine data from multiple sources or tables for analysis.

Utility:

Joining data allows you to combine datasets with common columns or indices, making it easier to work with larger and more complex datasets.

Common use cases include merging data from different sources (e.g., combining sales data with customer data).

Keywords:

merge, join, concat, inner join, outer join, left join, right join, combining dataframes.

Common Joining Methods

Inner Join:

Merges only the rows with keys present in both dataframes.

Example:

```
merged_df = pd.merge(df1, df2, on='key_column', how='inner')
```

Left Join:

Merges all rows from the left dataframe, and only matching rows from the right.

Example:

```
merged_df = pd.merge(df1, df2, on='key_column', how='left')
```

Right Join:

Merges all rows from the right dataframe, and only matching rows from the left.

Example:

```
merged_df = pd.merge(df1, df2, on='key_column', how='right')
```

Outer Join:

Merges all rows from both dataframes. Missing values are filled with NaN.

Example:

```
merged_df = pd.merge(df1, df2, on='key_column', how='outer')
```

Concatenating DataFrames:

Used to stack dataframes either vertically (row-wise) or horizontally (column-wise).

Example (row-wise):

```
concatenated_df = pd.concat([df1, df2], axis=0)
```

Join on Index:

Merges two dataframes based on their indices.

Example:

```
joined_df = df1.join(df2, how='inner')
```

Example of a Left Join:

```
import pandas as pd

# Sample data
df1 = pd.DataFrame({
    'ID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'David']
})

df2 = pd.DataFrame({
    'ID': [1, 2, 5],
    'Score': [85, 90, 95]
})

# Perform a left join on the 'ID' column
merged_df = pd.merge(df1, df2, on='ID', how='left')
```

merge_ordered()

Description:

Merges DataFrames, maintaining the order of observations. Useful for time series data.

Use Case:

Used when merging sorted or time series data to preserve the order.

Syntax:

```
pd.merge_ordered(df1, df2, on='key_column')
```

merge_asof()

Description:

Performs an asof merge, matching on the closest previous key (usually for time series).

Use Case:

Merges DataFrames where rows are aligned by the nearest match before a specified date or key.

Syntax:

```
pd.merge_asof(df1, df2, on='key_column', direction='backward')
```

concat()

Description:

Concatenates multiple DataFrames either along rows (axis=0) or columns (axis=1).

Use Case:

Used for stacking DataFrames either vertically or horizontally.

Syntax:

```
pd.concat([df1, df2], axis=0)  # Vertical concatenation
pd.concat([df1, df2], axis=1)  # Horizontal concatenation
```

Example:

```
df1 = pd.DataFrame({'ID': [1, 2], 'Name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'ID': [3, 4], 'Name': ['Charlie', 'David']})

concatenated = pd.concat([df1, df2])
```

Output:

```
   ID    Name
0   1   Alice
1   2     Bob
0   3 Charlie
1   4   David
```

Data Manipulation Functions in Pandas

Pandas offers a wide array of functions for reshaping, merging, and manipulating data. Here's a breakdown of important functions like melt(), pivot(), pivot_table(), crosstab(), cut(), qcut(), merge(), merge_ordered(), merge_asof(), and concat().

1. melt()

Description:

Transforms a wide format DataFrame into a long format by unpivoting columns into rows.

Use Case:

When you need to reshape data for better visualization or analysis, especially in tidy data format.

Syntax:

```
pd.melt(df, id_vars=['key_column'], value_vars=['col1', 'col2'])
```

Example:

```
df = pd.DataFrame({
  'ID': [1, 2],
  'Height': [150, 160],
  'Weight': [50, 60]
})

melted = pd.melt(df, id_vars=['ID'], value_vars=['Height', 'Weight'])
```

Output:

```
  ID  variable  value
0  1   Height   150
1  2   Height   160
2  1   Weight    50
3  2   Weight    60
```

2. pivot()

Description:

Transforms long-format data into wide format by reshaping rows into columns.

Use Case:

Used when you need to reorganize or summarize data by turning unique values into separate columns.

Syntax:

```
df.pivot(index='key_column', columns='column_to_pivot', values='value_column')
```

Example:

```
df = pd.DataFrame({
    'ID': [1, 1, 2, 2],
    'Variable': ['Height', 'Weight', 'Height', 'Weight'],
    'Value': [150, 50, 160, 60]
})
```

```
pivoted = df.pivot(index='ID', columns='Variable', values='Value')
```

Output:

```
Variable  Height  Weight
ID
1         150     50
2         160     60
```

3. pivot_table()

Description:

Creates a pivot table for summarizing data with aggregation functions like mean, sum, count.

Use Case:

When you want to summarize or aggregate data, similar to Excel’s pivot table.

Syntax:

```
df.pivot_table(index='key_column', values='value_column', aggfunc='mean')
```

Example:

```
df = pd.DataFrame({
    'ID': [1, 1, 2, 2],
    'Score': [90, 85, 88, 92],
```

```
    'Subject': ['Math', 'Science', 'Math', 'Science']
})

pivot_tbl = df.pivot_table(index='ID', columns='Subject', values='Score', aggfunc='mean')
```

Output:

```
Subject  Math  Science
ID
1        90    85
2        88    92
```

4. crosstab()

Description:

Computes a cross-tabulation of two or more factors, creating a contingency table.

Use Case:

For comparing frequencies of categorical data, or for simple aggregation of data across categories.

Syntax:

```
pd.crosstab(df['col1'], df['col2'])
```

Example:

```
df = pd.DataFrame({
```

    'Gender': ['Male', 'Female', 'Male', 'Female'],

    'Preference': ['Football', 'Basketball', 'Basketball', 'Football']

})

crosstab = pd.crosstab(df['Gender'], df['Preference'])

Output:

```
Preference  Basketball  Football
Gender
Female           1        1
Male             1        1
```

5. cut()

Description:

Bins continuous data into discrete intervals.

Use Case:

When you want to segment data into intervals (bins), for example, to create categories for age groups or income ranges.

Syntax:

pd.cut(df['numeric_column'], bins=[0, 10, 20, 30])

Example:

```
ages = [15, 22, 35, 45, 52]

categories = pd.cut(ages, bins=[0, 18, 35, 60], labels=['Teen', 'Young Adult', 'Adult'])
```

Output:

```
['Teen', 'Young Adult', 'Adult', 'Adult', 'Adult']

Categories (3, object): ['Teen' < 'Young Adult' < 'Adult']
```

6. qcut()

Description:

Divides data into quantiles (equal-sized intervals) based on the rank or distribution.

Use Case:

For splitting data into quantiles (e.g., quartiles, deciles), where each bin contains approximately the same number of observations.

Syntax:

```
pd.qcut(df['numeric_column'], q=4)
```

Example:

```
values = [1, 2, 3, 4, 5]

quantiles = pd.qcut(values, q=3)
```

Output:

```
[(0.999, 2.333], (0.999, 2.333], (2.333, 3.667], (3.667, 5.0], (3.667, 5.0]]

Categories (3, interval[float64]): [(0.999, 2.333] < (2.333, 3.667] < (3.667, 5.0]]
```

Data Manipulation with pandas in Python

Description:

pandas is a powerful Python library used for data manipulation and analysis. It provides data structures like DataFrames and Series that make it easy to work with structured data. Common tasks include filtering, sorting, grouping, transforming, and reshaping data.

Utility:

pandas helps in cleaning, modifying, and transforming datasets to make them ready for analysis.

You can handle missing data, calculate new columns, aggregate data, and reshape it to fit your requirements.

Keywords:

filter, sort, groupby, transform, reshape, manipulate data, aggregate, pandas.

Common Data Manipulation Tasks in pandas

Filtering Data

Description: Extract specific rows based on conditions.

Example:

```
# Filter rows where 'age' is greater than 30
filtered_df = df[df['age'] > 30]
```

Sorting Data

Description: Sort the DataFrame by one or more columns.

Example:

```
# Sort by 'age' in descending order
sorted_df = df.sort_values(by='age', ascending=False)
```

Creating New Columns

Description: Create new columns based on existing data or calculations.

Example:

```
# Create a new column 'total' as the sum of 'col1' and 'col2'
df['total'] = df['col1'] + df['col2']
```

Handling Missing Data

Description: Fill, drop, or manipulate missing values.

Example:

```
# Count the number of missing values in each column
print(planes.isna().sum())

# Find the five percent threshold
threshold = len(planes) * 0.05

# Create a filter
cols_to_drop = planes.columns[planes.isna().sum() <= threshold]

# Drop missing values for columns below the threshold
planes.dropna(subset=cols_to_drop, inplace=True)
```

```
print(planes.isna().sum())

# Calculate median plane ticket prices by Airline
airline_prices = planes.groupby("Airline")["Price"].median()

print(airline_prices)

# Convert to a dictionary
prices_dict = airline_prices.to_dict()

# Map the dictionary to missing values of Price by Airline
planes["Price"] = planes["Price"].fillna(planes["Airline"].map(prices_dict))
```

Grouping and Aggregating Data

Description: Group data by one or more columns and calculate aggregates like sum, mean, etc.

Example:

```
# Group by 'category' and calculate the mean of 'value' column
grouped_df = df.groupby('category')['value'].mean()
```

Renaming Columns

Description: Rename one or more columns in the DataFrame.

Example:

```
# Rename a single column
df.rename(columns={'old_name': 'new_name'}, inplace=True)
```

Dropping Columns or Rows

- Description: Remove unnecessary columns or rows.
- Example:

```
# Drop a column
df.drop(columns=['column_to_drop'], inplace=True)
```

Reshaping Data

- Description: Use functions like pivot() or melt() to reshape the DataFrame.
- Example (Pivot):

```
# Pivot the DataFrame to reorganize data
pivot_df = df.pivot(index='category', columns='type', values='value')
```

Merging or Joining Data

- Description: Combine multiple DataFrames using merge or join operations.
- Example:

  ```
  # Merge two DataFrames on a common column
  merged_df = pd.merge(df1, df2, on='key_column', how='inner')
  ```

Example of Data Manipulation:

```
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 45],
    'Salary': [50000, 60000, 70000, 80000, 90000],
    'Department': ['HR', 'Finance', 'IT', 'HR', 'Finance']
}
df = pd.DataFrame(data)

# Filtering data for employees older than 30
filtered_df = df[df['Age'] > 30]

# Adding a new column 'Bonus' as 10% of 'Salary'
df['Bonus'] = df['Salary'] * 0.10

# Grouping by 'Department' and calculating average 'Salary'
grouped_df = df.groupby('Department')['Salary'].mean()

# Sorting by 'Age'
sorted_df = df.sort_values(by='Age')
```

Handling Missing Data Example:

```
import numpy as np
```

```
# Introduce missing data
df.loc[2, 'Salary'] = np.nan

# Fill missing 'Salary' with the mean of the column
df['Salary'].fillna(df['Salary'].mean(), inplace=True)
```

Data Cleaning in Python (using Pandas)

Data cleaning is a crucial part of the data preprocessing pipeline. It involves handling missing data, correcting inconsistent entries, and transforming data into a suitable format for analysis. Pandas is the go-to library for data cleaning tasks in Python.

Duplicates Handling

Description:

Duplicates can skew analysis and need to be removed or handled properly.

2.1. Identifying Duplicates

Function:

duplicated()

Syntax:

```
df.duplicated()  # Returns True for duplicates
```

Example:

```
df[df.duplicated()]  # Show duplicate rows
```

Dropping Duplicates

Function:

drop_duplicates()

Syntax:

```
df.drop_duplicates(inplace=True)
```

Example:

```
df_cleaned = df.drop_duplicates()
```

Handling Outliers

Description:

Outliers are extreme values that may distort the analysis. They can be handled by capping or removing them.

3.1. Removing Outliers Using Quantiles

Function:

Quantile-based filtering.

Syntax:

```
Q1 = df['column_name'].quantile(0.25)
Q3 = df['column_name'].quantile(0.75)
IQR = Q3 - Q1
df_cleaned = df[~((df['column_name'] < (Q1 - 1.5 * IQR)) | (df['column_name'] > (Q3 + 1.5 * IQR)))]
```

Example:

```
Q1 = df['price'].quantile(0.25)
```

```
Q3 = df['price'].quantile(0.75)
IQR = Q3 - Q1
df_cleaned = df[~((df['price'] < (Q1 - 1.5 * IQR)) | (df['price'] > (Q3 + 1.5 * IQR)))]
```

Data Type Conversion

Description:

Converting data to appropriate types (e.g., integers, floats, categories) ensures correct analysis.

Converting Data Types

Function:

astype()

Syntax:

```
df['column_name'] = df['column_name'].astype('int')  # Convert to integer
df['column_name'] = df['column_name'].astype('category')  # Convert to categorical
```

Example:

```
df['age'] = df['age'].astype('int')
```

Converting to Datetime

Function:

pd.to_datetime()

Syntax:

```
df['date_column'] = pd.to_datetime(df['date_column'])
```

Example:

```
df['date'] = pd.to_datetime(df['date'])
```

Handling Categorical Data

Description:

Categorical data often requires special handling, including encoding for machine learning models.

Converting Categorical Data

Function:

pd.Categorical()

Syntax:

```
df['category_column'] = pd.Categorical(df['category_column'])
```

Example:

```
df['gender'] = pd.Categorical(df['gender'])
```

Encoding Categorical Variables

Function:

pd.get_dummies()

Syntax:

```
df_encoded = pd.get_dummies(df, columns=['category_column'])
```

Example:

```
df_encoded = pd.get_dummies(df, columns=['gender'])
```

Renaming Columns

Description:

Renaming columns can improve clarity and consistency in the dataset.

Function:

rename()

Syntax:

```
df.rename(columns={'old_column': 'new_column'}, inplace=True)
```

Example:

```
df.rename(columns={'age': 'Age in Years'}, inplace=True)
```

Filtering Data

Description:

Filtering allows you to subset your dataset based on conditions.

Function:

Boolean indexing

Syntax:

```
df_filtered = df[df['column_name'] == value]
```

Example:

```
df_filtered = df[df['age'] > 30]
```

Removing Unnecessary Columns

Description:

Dropping irrelevant columns reduces noise in the dataset.

Function:

drop()

Syntax:

```
df.drop(columns=['unwanted_column'], inplace=True)
```

Example:

```
df.drop(columns=['ID'], inplace=True)
```

String Manipulation in Python (using Pandas)

String manipulation is an essential part of cleaning and transforming textual data. Pandas provides a set of string functions that allow for efficient and flexible manipulation of string data in DataFrames.

1. Removing Whitespace

Description:

Remove leading, trailing, or both types of whitespace from string data.

Function:

str.strip(), str.lstrip(), str.rstrip()

Syntax:

```
df['column_name'] = df['column_name'].str.strip()  # Remove leading and trailing whitespace
df['column_name'] = df['column_name'].str.lstrip()  # Remove leading whitespace
df['column_name'] = df['column_name'].str.rstrip()  # Remove trailing whitespace
```

Example:

```
df['name'] = df['name'].str.strip()
```

2. Changing Case

Description:

Convert string data to lower, upper, or title case for uniformity.

Function:

str.lower(), str.upper(), str.title()

Syntax:

```
df['column_name'] = df['column_name'].str.lower()  # Convert to lowercase
df['column_name'] = df['column_name'].str.upper()  # Convert to uppercase
df['column_name'] = df['column_name'].str.title()  # Convert to title case
```

Example:

```
df['city'] = df['city'].str.lower()
```

3. Replacing Substrings

Description:

Replace specific substrings with another value in the string.

Function:

str.replace()

Syntax:

```
df['column_name'] = df['column_name'].str.replace('old_value', 'new_value')
```

Example:

```
df['address'] = df['address'].str.replace('Street', 'St.')
```

4. Extracting Substrings

Description:

Extract specific parts of strings using regular expressions or slicing.

Function:

str.extract(), str.slice()

Syntax:

```
df['new_column'] = df['column_name'].str.extract(r'(regex_pattern)')
df['new_column'] = df['column_name'].str.slice(start, stop)
```

Example:

```
df['zipcode'] = df['address'].str.extract(r'(\d{5})')  # Extract ZIP codes (5 digits)
df['first_three_chars'] = df['column'].str.slice(0, 3)  # Extract first 3 characters
```

5. Splitting Strings

Description:

Split strings into multiple parts using a delimiter.

Function:

str.split()

Syntax:

```
df['new_column'] = df['column_name'].str.split('delimiter')
```

Example:

```
df['first_name'] = df['name'].str.split(' ').str[0]  # Extract the first name from full name
```

6. Concatenating (Joining) Strings

Description:

Concatenate two or more string columns together.

Function:

+ operator, str.cat()

Syntax:

```
df['full_name'] = df['first_name'] + ' ' + df['last_name']
df['full_name'] = df['first_name'].str.cat(df['last_name'], sep=' ')
```

Example:

```
df['full_address'] = df['street'] + ', ' + df['city'] + ', ' + df['state']
```

7. Checking for Substring Presence

Description:

Check if a substring is present in a string.

Function:

str.contains()

Syntax:

```
df['contains_substring'] = df['column_name'].str.contains('substring')
```

Example:

```
df['has_new_york'] = df['city'].str.contains('New York')
```

8. String Length

Description:

Find the length of each string in a column.

Function:

str.len()

Syntax:

```
df['string_length'] = df['column_name'].str.len()
```

Example:

```
df['name_length'] = df['name'].str.len()
```

9. Finding Substring Index

Description:

Find the index of the first occurrence of a substring.

Function:

str.find()

Syntax:

```
df['substring_index'] = df['column_name'].str.find('substring')
```

Example:

```
df['index_of_comma'] = df['address'].str.find(',')
```

10. Removing Characters

Description:

Remove specific characters or patterns from strings.

Function:

str.replace()

Syntax:

```
df['cleaned_column'] = df['column_name'].str.replace('[^A-Za-z0-9]', '', regex=True)
```

Example:

```
df['phone_number_cleaned'] = df['phone_number'].str.replace('[^0-9]', '',
regex=True)
```

Dealing with Datetimelike Data in Python

Working with datetimelike data (e.g., dates, times, and time spans) is an essential part of data analysis. Pandas provides extensive capabilities to manipulate and process such data efficiently.

1. Converting to Datetime Format

Description:

Convert strings or other formats into proper datetime objects.

Function:

pd.to_datetime()

Syntax:

```
df['date_column'] = pd.to_datetime(df['date_column'])
```

Example:

```
import pandas as pd
df = pd.DataFrame({'date': ['2022-10-01', '2022-11-01']})
df['date'] = pd.to_datetime(df['date'])
```

2. Extracting Components (Year, Month, Day, etc.)

Description:

Extract specific components (year, month, day, hour, etc.) from a datetime column.

Attributes:

dt.year

dt.month

dt.day

dt.hour, dt.minute, dt.second

Syntax:

```
df['year'] = df['date_column'].dt.year
df['month'] = df['date_column'].dt.month
df['day'] = df['date_column'].dt.day
```

Example:

```
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
```

3. Creating Date Ranges

Description:

Generate a sequence of dates over a specified period.

Function:

pd.date_range()

Syntax:

```
date_range = pd.date_range(start='2022-01-01', end='2022-12-31', freq='M')
```

Example:

```
date_range = pd.date_range(start='2022-01-01', periods=10, freq='D')
```

4. Handling Time Differences (Timedeltas)

Description:

Perform operations on datetime columns to calculate differences.

Function:

pd.Timedelta()

Syntax:

```
df['time_diff'] = df['end_date'] - df['start_date']
```

Example:

```
df['time_diff'] = df['end_date'] - df['start_date']
```

You can also create custom timedeltas:

```
pd.Timedelta(days=5, hours=3)
```

5. Resampling Time Series Data

Description:

Resample time series data to a different frequency (e.g., daily, monthly).

Function:

resample()

Syntax:

```
df.resample('M', on='date_column').mean()  # Resample by month
```

Example:

python

```python
df = df.set_index('date')
df_monthly = df['value'].resample('M').mean()  # Monthly averages
```

6. Shifting and Lagging Data

Description:

Shift the data forward or backward by a certain number of periods (useful for creating lagged features in time series analysis).

Function:

shift()

Syntax:

```
df['lagged_value'] = df['value'].shift(1)  # Shift by 1 period
```

Example:

```
df['previous_day_value'] = df['value'].shift(1)
```

7. Filtering by Date/Time

Description:

Filter rows based on specific datetime ranges.

Boolean Indexing:

```
df[(df['date_column'] >= '2022-01-01') & (df['date_column'] < '2023-01-01')]
```

Example:

```
df_filtered = df[(df['date'] >= '2022-01-01') & (df['date'] < '2023-01-01')]
```

8. Time Zones and Localization

Description:

Handle time zones and convert between them.

Function:

tz_localize() and tz_convert()

Syntax:

```
df['date_column'] = df['date_column'].dt.tz_localize('UTC')
df['date_column'] = df['date_column'].dt.tz_convert('America/New_York')
```

Example:

```
df['date'] = df['date'].dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
```

9. Date Offsets

Description:

Date offsets allow you to add or subtract dates with different intervals (e.g., adding a month or a week).

Function:

pd.DateOffset()

Syntax:

```
df['next_month'] = df['date_column'] + pd.DateOffset(months=1)
```

Example:

```
df['next_week'] = df['date'] + pd.DateOffset(weeks=1)
```

10. Custom Date Formatting

Description:

Format datetime objects as strings using custom formats.

Function:

strftime()

Syntax:

```
df['formatted_date'] = df['date_column'].dt.strftime('%Y-%m-%d')
```

Example:

```
df['formatted_date'] = df['date'].dt.strftime('%B %d, %Y')  # Output: 'October 01, 2022'
```

Aggregation and Transformation in Python (using Pandas)

Pandas provides powerful methods for performing operations on data groups and applying functions across rows or columns. The .agg(), .apply(), and .transform() functions are used to manipulate data, offering different levels of flexibility and behavior.

1. .agg() (Aggregation)

Description:

The .agg() function is used to apply one or more aggregation functions to a DataFrame or Series. It is often used for grouped data to summarize multiple columns in a single operation.

Function:

.agg()

Utilization:

Aggregate data by applying one or more functions (e.g., mean(), sum(), min(), max()) to columns.

Useful for summarizing data after grouping.

Syntax:

df.groupby('column_name').agg({'column1': 'mean', 'column2': 'sum'})

df.agg({'column1': ['min', 'max'], 'column2': 'mean'})

Example:

```python
# Calculate mean for one column and sum for another, grouped by a category
df.groupby('category').agg({'sales': 'sum', 'profit': 'mean'})
```

Key Points:

- You can pass a dictionary to apply different functions to different columns.
- Supports multiple functions per column (in a list).

2. .transform() (Element-wise Transformation)

Description:

The .transform() function is used to perform operations that return a result with the same shape as the input, meaning it operates on each element or group independently.

- Function:
- .transform()

Utilization:

- Applies a function to each element in a column or group and returns the same shape as the input.
- Commonly used with grouped data to apply transformations within groups.

Syntax:

```python
df['transformed_column'] =
df.groupby('group_column')['column_name'].transform(function)
```

Example:

Calculate the z-score for each value within its group

df['zscore'] = df.groupby('category')['value'].transform(lambda x: (x - x.mean()) / x.std())

Key Points:

Unlike .agg(), .transform() returns an object that has the same shape as the original.

Frequently used for operations like normalization, ranking, or calculating running totals within groups.

3. .apply() (Apply Custom Functions)

Description:

The .apply() function allows you to apply custom functions row-wise or column-wise. It's extremely flexible and can be used for almost any transformation or calculation.

Function:

.apply()

Utilization:

Apply a function across rows or columns.

Used when you need to apply a custom function that may not be directly supported by Pandas.

Syntax:

```
df['new_column'] = df['column_name'].apply(function)

df.apply(function, axis=0)  # Apply the function to each column

df.apply(function, axis=1)  # Apply the function to each row
```

Example:

```
# Apply a function to each row to calculate a new value

df['new_column'] = df.apply(lambda row: row['col1'] + row['col2'], axis=1)
```

Key Points:

Supports both row-wise and column-wise operations (controlled by the axis parameter).

Allows for more complex and customized operations than .agg() or .transform().

Comparison

.agg(): Used for aggregation operations like mean, sum, min, and max. It usually reduces the number of rows.

.transform(): Used for element-wise transformations that return an object of the same shape as the original. Common for operations within groups.

.apply(): Most flexible and can be used for custom row/column operations. It does not necessarily change the shape but allows you to apply more complex logic.

Description:

Statistics is a branch of mathematics that deals with collecting, analyzing, interpreting, presenting, and organizing data. Python, with libraries like NumPy, pandas, and SciPy, provides powerful tools for performing statistical analysis.

Utility:

Understanding statistics is essential for data analysis, as it helps in making informed decisions based on data. Python's libraries make it easy to perform statistical tests, summarize data, and visualize results.

Keywords:

descriptive statistics, inferential statistics, mean, median, mode, standard deviation, probability distributions, hypothesis testing.

Basic Statistical Concepts

- Descriptive Statistics:
  - Summary measures that describe the main features of a dataset.
  - Common Measures:
    - Mean: The average of a dataset.
    - Median: The middle value when data is sorted.
    - Mode: The most frequently occurring value.
    - Standard Deviation: A measure of the dispersion or spread of the data.
  - Example:

```
import pandas as pd

# Sample data
data = [1, 2, 2, 3, 4, 5, 5, 5, 6]
```

```
# Convert to DataFrame
df = pd.DataFrame(data, columns=['Values'])

# Descriptive statistics
mean = df['Values'].mean()
median = df['Values'].median()
mode = df['Values'].mode()[0]  # Mode can return multiple values
std_dev = df['Values'].std()

print(f"Mean: {mean}, Median: {median}, Mode: {mode}, Std Dev: {std_dev}")
```

Inferential Statistics:

Techniques that allow us to use a sample to make inferences about a population.

Common Techniques:

Confidence Intervals: A range of values that is likely to contain the population parameter.

Hypothesis Testing: A method to determine if there is enough evidence to reject a null hypothesis.

Probability Distributions:

Functions that describe the likelihood of obtaining the possible values that a random variable can take.

Common Distributions:

Normal Distribution: Bell-shaped curve, characterized by its mean and standard deviation.

Binomial Distribution: Represents the number of successes in a fixed number of trials.

Example:

```python
import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns


# Generate a normal distribution

data = np.random.normal(loc=0, scale=1, size=1000)


# Plot the distribution

sns.histplot(data, bins=30, kde=True)

plt.title('Normal Distribution')

plt.xlabel('Value')

plt.ylabel('Frequency')

plt.show()
```

Hypothesis Testing:

A process to test assumptions or claims about a population based on sample data.

Common Tests:

- t-test: Compares the means of two groups.
- Chi-square test: Tests the relationship between categorical variables.

Example:

```python
from scipy import stats
```

```
# Sample data
group1 = [20, 21, 23, 24, 30]
group2 = [22, 25, 28, 29, 31]

# Perform a t-test
t_stat, p_value = stats.ttest_ind(group1, group2)

print(f"T-statistic: {t_stat}, P-value: {p_value}")
if p_value < 0.05:
   print("Reject the null hypothesis")
else:
   print("Fail to reject the null hypothesis")
```

Probability

Description:

Probability is a measure of the likelihood that a certain event will occur. It ranges from 0 (impossible event) to 1 (certain event). Probability is used to quantify uncertainty in various scenarios and is the foundation for many areas in statistics, data analysis, and machine learning.

Formula:

For a simple event AA, P(A)=Number of favorable outcomesTotal number of outcomesP(A)=Total number of outcomesNumber of favorable outcomes

Key Concepts:

Experiment: A process that produces random outcomes (e.g., rolling a die, flipping a coin).

Sample Space (SS): The set of all possible outcomes of an experiment.

Event: A subset of the sample space (e.g., rolling a 5 on a die).

Outcome: A single result of an experiment (e.g., heads in a coin toss).

Types of Probability:

Classical Probability:

Used when all outcomes in the sample space are equally likely.

Formula:

P(A)=|A||S|

P(A)=|S||A| where |A||A| is the number of favorable outcomes and |S||S| is the total number of outcomes.

Empirical (Experimental) Probability:

Based on observations or experiments.

Formula:

$P(A)=$Number of times A occursTotal number of trials

$P(A)=$Total number of trialsNumber of times A occurs Example: Estimating the probability of rain based on historical weather data.

Subjective Probability:

Based on personal judgment or experience, without formal calculations.

Example: Estimating the likelihood of a stock price increase based on expert knowledge.

Conditional Probability:

The probability of an event AA occurring given that event BB has already occurred.

Formula:

$P(A|B)=P(A\cap B)P(B)$

$P(A|B)=P(B)P(A\cap B)$ Example: Probability of drawing a king from a deck given that a red card was drawn.

Keywords:

likelihood, chance, randomness, favorable outcome, conditional probability.

Basic Rules of Probability:

Sum Rule (Addition Rule):

For two mutually exclusive events AA and BB, the probability of either AA or BB occurring is:

$P(A\cup B)=P(A)+P(B)$

$P(A\cup B)=P(A)+P(B)$

Product Rule (Multiplication Rule):

For two independent events AA and BB, the probability of both AA and BB occurring is:

P(A∩B)=P(A)×P(B)

P(A∩B)=P(A)×P(B)

Complement Rule:

The probability of an event AA not occurring is:

P(not A)=1−P(A)

P(not A)=1−P(A)

Example Problems in Python:

Classical Probability: Probability of rolling a 3 on a fair die.

```
# Classical probability
favorable_outcomes = 1  # Only 1 favorable outcome (rolling a 3)
total_outcomes = 6  # 6 sides on the die
probability = favorable_outcomes / total_outcomes
print(probability)  # Output: 0.1667 (or 1/6)
```

Conditional Probability: Probability of drawing an ace given that a face card has been drawn from a standard deck.

```
# Conditional probability
# There are 3 aces in the remaining 51 cards after one card (not an ace) has been drawn
probability_given_face_card = 3 / 51
print(probability_given_face_card)  # Output: 0.0588
```

Independent and Dependent Events in Probability

Independent Events:

Description:

Two events are independent if the outcome of one event does not affect the outcome of the other event. In other words, knowing that one event occurred gives no information about whether the other event occurred.

Utility:

Independent events often simplify probability calculations since the joint probability of two independent events is simply the product of their individual probabilities.

Keywords:

independent events, no influence, joint probability, $P(A \text{ and } B) = P(A) * P(B)$.

Dependent Events:

Description:

Two events are dependent if the outcome of one event affects the outcome of the other event. In this case, the probability of one event occurring depends on whether the other event has occurred.

Utility:

Dependent events require conditional probability, where the probability of one event is adjusted based on the occurrence of another event.

Keywords:

dependent events, conditional probability, influence, $P(A \text{ and } B) = P(A) * P(B|A)$.

Independent vs Dependent Event Examples in Python

Independent Events Example:

Tossing two coins (the result of one coin flip does not influence the other).

Example:

```
P_A = 0.5  # Probability of heads on the first coin
P_B = 0.5  # Probability of heads on the second coin

# Since the events are independent, the joint probability is:
P_A_and_B = P_A * P_B
print(f"Joint Probability of two heads: {P_A_and_B}")
```

Dependent Events Example:

Drawing two cards from a deck without replacement (the outcome of the first draw influences the second).

Example:

```
total_cards = 52
spades = 13

# Probability of drawing a spade on the first draw
P_A = spades / total_cards

# Probability of drawing another spade if the first card was a spade (without replacement)
P_B_given_A = (spades - 1) / (total_cards - 1)
```

```python
# Joint probability of drawing two spades in a row (dependent events)
P_A_and_B = P_A * P_B_given_A
print(f"Probability of drawing two spades: {P_A_and_B}")
```

Probability with Replacement:

Description:

In probability with replacement, after an event (like drawing a card), the item is returned to the original set before the next event occurs. This ensures that the total number of possible outcomes remains constant.

Utility:

When sampling with replacement, the events are independent, as the probability of each event does not change between draws.

Keywords:

with replacement, constant probability, independent events.

Probability without Replacement:

Description:

In probability without replacement, after an event (like drawing a card), the item is not returned to the original set. This changes the total number of possible outcomes and makes the events dependent.

Utility:

When sampling without replacement, the events are dependent, as the outcome of one event affects the probabilities of subsequent events.

Keywords:

without replacement, changing probability, dependent events.

Probability with and without Replacement Examples in Python

With Replacement:

Drawing a ball from a bag, replacing it, and then drawing again.

Example:

```
total_balls = 10
red_balls = 4

# Probability of drawing a red ball twice (with replacement)
P_A = red_balls / total_balls
P_B = red_balls / total_balls

# Joint probability (independent events with replacement)
P_A_and_B = P_A * P_B
print(f"Probability of drawing two red balls (with replacement): {P_A_and_B}")
```

Without Replacement:

Drawing a ball from a bag, not replacing it, and then drawing again.

Example:

```
total_balls = 10
red_balls = 4

# Probability of drawing a red ball first
```

```
P_A = red_balls / total_balls

# Probability of drawing a red ball second (without replacement)
P_B_given_A = (red_balls - 1) / (total_balls - 1)

# Joint probability (dependent events without replacement)
P_A_and_B = P_A * P_B_given_A
print(f"Probability of drawing two red balls (without replacement): {P_A_and_B}")
```

CDF, PMF, PDF, and PPF

These are key functions used in probability theory and statistics to describe different aspects of probability distributions. Here's a breakdown of each concept, its role, and how they are typically used:

1. CDF (Cumulative Distribution Function)

Description:

The Cumulative Distribution Function (CDF) gives the probability that a random variable XX will take a value less than or equal to a given value xx. It works for both discrete and continuous distributions.

Formula:

Discrete Distributions:

F(x)=P(X≤x)

F(x)=P(X≤x)

Continuous Distributions:

F(x)=∫−∞xf(t)dt

F(x)=∫−∞xf(t)dt Where f(t)f(t) is the Probability Density Function (PDF).

Keywords:

cumulative probability, range, less than or equal to.

Example in Python:

```
from scipy.stats import norm

# CDF of standard normal distribution at x = 1.96
```

```
cdf_value = norm.cdf(1.96)
print(cdf_value)  # Output: 0.975 (probability that X ≤ 1.96)
```

2. PMF (Probability Mass Function)

Description:

The Probability Mass Function (PMF) applies to discrete distributions. It provides the probability that a discrete random variable XX takes exactly the value xx.

Formula:

For a discrete random variable XX,

P(X=x)

P(X=x)

Keywords:

discrete probability, exact value, specific outcome.

Example in Python:

```
from scipy.stats import binom

# PMF of binomial distribution with n=10, p=0.5, at k=3
pmf_value = binom.pmf(3, 10, 0.5)
print(pmf_value)  # Output: probability of exactly 3 successes in 10 trials
```

3. PDF (Probability Density Function)

Description:

The Probability Density Function (PDF) is used for continuous distributions. Unlike PMF, it doesn't give the probability of a specific value (since the probability of a specific value in a continuous distribution is zero), but rather the density at a given value, which is used to calculate the probability over a range.

Formula:

For continuous random variables, the probability that XX falls between two values aa and bb is given by the area under the PDF between aa and bb:

P(a≤X≤b)=∫abf(x)dx

P(a≤X≤b)=∫abf(x)dx

Keywords:

continuous probability, density, range probability.

Example in Python:

```
from scipy.stats import norm

# PDF of standard normal distribution at x = 0
pdf_value = norm.pdf(0)
print(pdf_value)  # Output: 0.3989 (density at X = 0)
```

4. PPF (Percent Point Function / Quantile Function)

Description:

The Percent Point Function (PPF) is the inverse of the CDF. Given a probability pp, the PPF returns the value xx such that P(X≤x)=pP(X≤x)=p. This function is used to find quantiles, which are values corresponding to given cumulative probabilities.

Formula:

If F(x)F(x) is the CDF, then:

PPF(p)=xsuch thatF(x)=p

PPF(p)=xsuch thatF(x)=p

Keywords:

inverse CDF, quantile, percentile.

Example in Python:

```
from scipy.stats import norm

# PPF (quantile function) of standard normal distribution for p = 0.975
ppf_value = norm.ppf(0.975)
print(ppf_value)  # Output: 1.96 (value corresponding to the 97.5th percentile)
```

Use Cases:

CDF: To find cumulative probabilities or evaluate the probability that a random variable falls within a range.

PMF: For finding the exact probability of discrete outcomes (e.g., number of successes in binomial experiments).

PDF: To evaluate probability densities in continuous distributions.

PPF: To find quantiles, such as the median or percentiles, based on a given probability.

Sampling in Python

Description:

Sampling refers to the process of selecting a subset of data from a larger dataset to estimate characteristics of the whole population. In statistics, this can involve random sampling, stratified sampling, and other techniques to ensure that the sample accurately reflects the population.

Utility:

Sampling is essential in data analysis and statistics, especially when dealing with large datasets, as it can save time and computational resources while still providing insights about the population.

Keywords:

sampling, random sampling, stratified sampling, sample size, pandas, numpy, scipy.

Common Sampling Techniques in Python

Random Sampling with Pandas:

Use the sample() method to randomly select rows from a DataFrame.

Example:

```
import pandas as pd

# Create a sample DataFrame
data = {'A': range(1, 101), 'B': range(101, 201)}
df = pd.DataFrame(data)

# Randomly sample 10 rows
```

random_sample = df.sample(n=10, random_state=1) # random_state for reproducibility

print(random_sample)

Sampling with NumPy:

Use numpy.random.choice() for sampling from a 1D array.

Example:

import numpy as np

Create a numpy array

arr = np.arange(1, 101)

Randomly sample 10 values

np_sample = np.random.choice(arr, size=10, replace=False) # replace=False ensures no duplicates

print(np_sample)

Stratified Sampling:

This method involves dividing the population into strata and then taking a sample from each stratum.

Example:

from sklearn.model_selection import train_test_split

Sample DataFrame with a categorical variable

```
df = pd.DataFrame({
    'Feature': range(100),
    'Stratum': ['A'] * 50 + ['B'] * 50
})

# Stratified sampling based on the 'Stratum' column
stratified_sample, _ = train_test_split(df, test_size=0.5, stratify=df['Stratum'], random_state=1)
print(stratified_sample)
```

Bootstrapping:

A resampling technique that involves repeatedly drawing samples from the data with replacement to estimate statistics.

Example:

```
# Bootstrapping example
bootstrapped_samples = []
for _ in range(1000):  # Create 1000 bootstrap samples
    sample = df.sample(n=len(df), replace=True)
    bootstrapped_samples.append(sample['A'].mean())

# Calculate the mean and confidence intervals
mean_estimate = np.mean(bootstrapped_samples)
ci_lower = np.percentile(bootstrapped_samples, 2.5)
ci_upper = np.percentile(bootstrapped_samples, 97.5)

print(f"Bootstrapped Mean Estimate: {mean_estimate}")
print(f"95% Confidence Interval: ({ci_lower}, {ci_upper})")
```

Systematic Sampling:

A method where you select every k-th element from a population list.

Example:

```
k = 5  # Sampling every 5th element
systematic_sample = df.iloc[::k]  # Select every 5th row
print(systematic_sample)
```

Sample Data from a Statistical Distribution

Description: Use numpy and scipy to generate random samples from common statistical distributions like normal, binomial, Poisson, and exponential.

Utility: Sampling allows you to simulate and analyze data that follows specific theoretical distributions.

Keywords: normal distribution, binomial distribution, Poisson distribution, exponential distribution, random samples.

Example:

```
import numpy as np

# Generate samples from a normal distribution
normal_samples = np.random.normal(loc=0, scale=1, size=1000)

# Generate samples from a binomial distribution
binomial_samples = np.random.binomial(n=10, p=0.5, size=1000)

# Generate samples from a Poisson distribution
```

```
poisson_samples = np.random.poisson(lam=3, size=1000)

# Generate samples from an exponential distribution
exponential_samples = np.random.exponential(scale=1, size=1000)
```

Confidence Interval

```
# Generate a 95% confidence interval using the quantile method
lower_quant = np.quantile(bootstrap_distribution, 0.025)
upper_quant = np.quantile(bootstrap_distribution, 0.975)

# Print quantile method confidence interval
print((lower_quant, upper_quant))

# Find the mean and std dev of the bootstrap distribution
point_estimate = np.mean(bootstrap_distribution)
standard_error = np.std(bootstrap_distribution, ddof=1)

# Find the lower limit of the confidence interval
lower_se = norm.ppf(0.025, loc=point_estimate, scale=standard_error)

# Find the upper limit of the confidence interval
upper_se = norm.ppf(0.975, loc=point_estimate, scale=standard_error)

# Print standard error method confidence interval
print((lower_se, upper_se))
```

Identifying Statistical Distributions

Description:

Identifying distributions involves determining the underlying probability distribution that best describes the behavior of a dataset. Different statistical distributions model different types of data and their characteristics, such as spread, central tendency, and probability of events. Understanding the correct distribution is critical for proper data analysis, hypothesis testing, and predictive modeling.

Common Distributions and Their Characteristics:

Normal Distribution (Gaussian):

Description:

Symmetrical, bell-shaped distribution where the mean, median, and mode are the same. Many natural phenomena follow this distribution.

Key Features:

Symmetrical around the mean.

68% of the data falls within 1 standard deviation from the mean.

95% of the data falls within 2 standard deviations from the mean.

Keywords:

bell-shaped, mean = median = mode, symmetric.

Use Cases:

Heights, test scores, measurement errors.

Binomial Distribution:

Description:

Describes the number of successes in a fixed number of independent Bernoulli trials (e.g., coin flips).

Key Features:

Discrete distribution.

Fixed number of trials with two possible outcomes (success or failure).

Probability of success (p) is constant.

Keywords:

success/failure, fixed trials, binary outcomes.

Use Cases:

Coin flips, pass/fail exams, quality control tests.

Poisson Distribution:

Description:

Models the probability of a given number of events occurring in a fixed interval of time or space, given a constant mean rate.

Key Features:

Discrete distribution.

Used for rare events.

The mean equals the variance (λ).

Keywords:

rare events, time/space intervals, mean = variance.

Use Cases:

Number of emails received per hour, number of car accidents per day.

Exponential Distribution:

Description:

Models the time between events in a Poisson process (i.e., events occurring continuously and independently at a constant average rate).

Key Features:

Continuous distribution.

Memoryless property (the future probability is independent of the past).

Mean = 1/λ.

Keywords:

time between events, Poisson process, memoryless.

Use Cases:

Time between phone calls, lifespan of electronic components.

Uniform Distribution:

Description:

All outcomes are equally likely within a certain range. The distribution is flat, indicating no preference for any interval.

Key Features:

Continuous or discrete.

All values within the interval are equally probable.

Keywords:

equally likely outcomes, flat, constant probability.

Use Cases:

Rolling a fair die, selecting a random number between two bounds.

Chi-Square Distribution:

Description:

The distribution of the sum of the squares of independent standard normal variables. It is used primarily in hypothesis testing.

Key Features:

Asymmetrical and skewed to the right.

Non-negative values only.

Defined by degrees of freedom.

Keywords:

hypothesis testing, skewed right, degrees of freedom.

Use Cases:

Chi-square tests for independence and goodness of fit.

T-Distribution (Student's t-distribution):

Description:

Similar to the normal distribution but with heavier tails, meaning more probability in the tails. Used when the sample size is small and the population standard deviation is unknown.

Key Features:

Symmetrical but with fatter tails than normal distribution.

As sample size increases, the t-distribution approaches the normal distribution.

Keywords:

small sample, fatter tails, approaches normal.

Use Cases:

Small sample hypothesis tests, confidence intervals.

Identifying Distributions from Data

To identify the correct distribution for your data, you can use several methods:

Visual Inspection:

Histogram:

Plot a histogram of your data to check the shape (e.g., bell-shaped, skewed, uniform).

Q-Q Plot (Quantile-Quantile Plot):

Compare your data against a theoretical distribution. If the data points lie approximately along a straight line, it suggests that the data follows the tested distribution.

Summary Statistics:

Calculate descriptive statistics like mean, median, skewness, and kurtosis.

Skewness: Helps identify whether the data is symmetrically distributed or skewed.

Kurtosis: Measures the tail behavior of the distribution.

Statistical Tests:

Kolmogorov-Smirnov Test:

A goodness-of-fit test to compare a sample with a reference probability distribution.

Shapiro-Wilk Test:

Tests whether a dataset follows a normal distribution.

Chi-Square Goodness-of-Fit Test:

Tests whether a sample matches a specific distribution (discrete).

Keywords to Identify Distributions in Questions:

Normal Distribution: Symmetrical, bell-shaped, mean = median = mode.

Binomial Distribution: Fixed trials, success/failure, binary outcomes.

Poisson Distribution: Events in time or space, rare events, mean = variance.

Exponential Distribution: Time between events, memoryless, constant rate.

Uniform Distribution: Equally likely, constant probability, flat.

Chi-Square Distribution: Hypothesis testing, degrees of freedom, skewed right.

T-Distribution: Small samples, fatter tails, unknown standard deviation.

Hypothesis Testing in Python

Description:

Hypothesis testing is a statistical method used to make inferences or draw conclusions about a population based on sample data. It involves formulating a null hypothesis (H0) and an alternative hypothesis (H1), followed by the selection of a significance level and conducting a test to determine whether to reject or fail to reject the null hypothesis.

Utility:

Hypothesis testing is widely used in various fields, including psychology, medicine, and economics, to validate assumptions or theories based on empirical data. Python provides libraries like scipy and statsmodels to perform these tests effectively.

Keywords:

hypothesis testing, null hypothesis, alternative hypothesis, p-value, t-test, ANOVA, chi-square test.

Common Hypothesis Tests in Python

One-Sample t-Test:

Used to determine if the sample mean is significantly different from a known population mean.

Example:

```
from scipy import stats

# Sample data
sample_data = [22, 21, 23, 24, 20, 21, 19, 22, 25]
```

```
# Hypothesized population mean
population_mean = 21

# Perform one-sample t-test
t_statistic, p_value = stats.ttest_1samp(sample_data, population_mean)

print(f"T-statistic: {t_statistic}, P-value: {p_value}")

# Interpretation
alpha = 0.05
if p_value < alpha:
    print("Reject the null hypothesis")
else:
    print("Fail to reject the null hypothesis")
```

Two-Sample t-Test:

- Used to compare the means of two independent groups.
- Example:

```
# Sample data for two groups
group1 = [22, 21, 23, 24, 20]
group2 = [19, 18, 20, 21, 17]

# Perform two-sample t-test
t_statistic, p_value = stats.ttest_ind(group1, group2)

print(f"T-statistic: {t_statistic}, P-value: {p_value}")
```

```
# Interpretation
if p_value < alpha:
    print("Reject the null hypothesis")
else:
    print("Fail to reject the null hypothesis")
```

ANOVA (Analysis of Variance):

Used to compare means across three or more groups.

Example:

```
# Sample data for three groups
group1 = [22, 21, 23, 24]
group2 = [19, 18, 20, 21]
group3 = [25, 24, 26, 27]

# Perform ANOVA
f_statistic, p_value = stats.f_oneway(group1, group2, group3)

print(f"F-statistic: {f_statistic}, P-value: {p_value}")

# Interpretation
if p_value < alpha:
    print("Reject the null hypothesis")
else:
    print("Fail to reject the null hypothesis")
```

Chi-Square Test:

Used to determine if there is a significant association between two categorical variables.

Example:

```
# Contingency table data
observed = [[10, 20], [20, 30]]

# Perform chi-square test
chi2_statistic, p_value, dof, expected = stats.chi2_contingency(observed)

print(f"Chi-square statistic: {chi2_statistic}, P-value: {p_value}")

# Interpretation
if p_value < alpha:
    print("Reject the null hypothesis")
else:
    print("Fail to reject the null hypothesis")
```

Proportion Test:

Used to compare proportions between two groups.

Example:

```
from statsmodels.stats.proportion import proportions_ztest
```

```
# Count of successes and sample sizes
successes = [30, 45]  # successes in group 1 and group 2
samples = [100, 120]  # total samples in group 1 and group 2

# Perform proportion test
z_statistic, p_value = proportions_ztest(successes, samples)

print(f"Z-statistic: {z_statistic}, P-value: {p_value}")

# Interpretation
if p_value < alpha:
    print("Reject the null hypothesis")
else:
    print("Fail to reject the null hypothesis")
```

Z-Score and P-Value

Example:

The null hypothesis is that the proportion of late shipments is six percent.

The alternative hypothesis is that the proportion of late shipments is **greater than** six percent.

The observed sample statistic, late_prop_samp, the hypothesized value, late_prop_hyp (6%), and the bootstrap standard error, std_error are available. norm from scipy.stats has also been loaded without an alias.

```
# Calculate the z-score of late_prop_samp
z_score = (late_prop_samp - late_prop_hyp) / std_error
```

```
# Calculate the p-value
p_value = 1 - norm.cdf(z_score)


# Print the p-value
print(p_value)
```

Statistical Experimentation Theory

Description:

Statistical experimentation theory, also known as design of experiments (DOE), is the process of designing and analyzing controlled experiments to make data-driven conclusions. This approach helps understand relationships between variables and determine causal effects. It is widely used in fields like medicine, manufacturing, and social sciences to test hypotheses, validate models, and make predictions based on sample data.

Key Components:

1. **Hypothesis Testing**:
   Formulating null and alternative hypotheses to compare results.
   - **Null Hypothesis (H_0)**: Assumes no effect or no difference.
   - **Alternative Hypothesis (H_1)**: Assumes some effect or difference.
2. **Randomization**:
   Randomly assigning subjects or samples to different groups to avoid biases and ensure that treatment effects are not influenced by confounding variables.
3. **Control Group**:
   A group that does not receive the treatment or intervention, providing a baseline to compare with the treatment group.
4. **Replication**:
   Repeating the experiment multiple times or with different subjects to reduce the effect of variability and ensure the results are consistent.
5. **Blocking**:
   Grouping experimental units into blocks that are similar, to control for variables that are not of primary interest but may affect the response.
6. **Factorial Design**:
   A design that involves more than one factor (independent variable), where each factor has different levels. This allows for testing the effect of multiple factors simultaneously and identifying interactions between them.

Key Terminologies in Statistical Experimentation

1. **Independent Variable (Factor)**:
   The variable that is manipulated or controlled in the experiment to observe its effect on the dependent variable.
2. **Dependent Variable (Response)**:
   The outcome variable that is measured to assess the effect of the independent variable.
3. **Confounding Variable**:
   An extraneous variable that affects the dependent variable, leading to incorrect conclusions if not controlled for.
4. **P-Value**:
   The probability that the observed results occurred by chance, assuming the null hypothesis is true. A low p-value (< 0.05) suggests strong evidence against the null hypothesis.
5. **Effect Size**:
   A quantitative measure of the strength of the phenomenon or the size of the treatment effect.
6. **Power of a Test**:
   The probability that the test correctly rejects a false null hypothesis ($1 - \beta$). Higher power reduces the risk of Type II errors.

Steps in Statistical Experimentation

1. **Define the Objective**:
   Clearly state the problem and the goal of the experiment. Identify the variables to be studied and the outcomes to be measured.
2. **Formulate Hypotheses**:
   Develop the null and alternative hypotheses based on the research question.
3. **Design the Experiment**:
   Decide on the experiment type, sampling method, and factors to be tested. Use randomization, blocking, or other strategies to minimize bias.
4. **Collect Data**:
   Conduct the experiment by applying treatments and collecting data from both the treatment and control groups.

5. **Analyze the Data**:
   Use statistical techniques like t-tests, ANOVA, or regression to analyze the data and determine whether to reject the null hypothesis.
6. **Draw Conclusions**:
   Interpret the results based on the analysis, assess the validity of the experiment, and consider the practical significance of the findings.

Types of Experimental Designs

1. **Completely Randomized Design (CRD)**:
   Subjects or units are randomly assigned to different treatments. This is the simplest experimental design and is appropriate when there are no known confounding variables.
2. **Randomized Block Design (RBD)**:
   Units are grouped into blocks based on a characteristic that may influence the response. Randomization is applied within each block. This design helps control for block-related variability.
3. **Factorial Design**:
   A design that involves multiple factors with different levels. Each combination of factor levels is tested, allowing for the study of interaction effects. Factorial designs are efficient for testing multiple variables simultaneously.
4. **Latin Square Design**:
   A design used when there are two potential blocking factors. Each treatment appears exactly once in each row and column, controlling for both factors.
5. **Crossover Design**:
   Subjects receive multiple treatments in different periods, allowing each subject to serve as their own control. This design is often used in medical trials.

Types of Statistical Tests Used in Experiments

1. **T-Test**:
   Compares the means of two groups to determine if they are significantly different.

- **Independent t-test**: Used when comparing two independent groups.
- **Paired t-test**: Used when comparing two related groups (e.g., before and after treatment).

2. **ANOVA (Analysis of Variance)**:
   Used to compare the means of three or more groups to determine if at least one group is significantly different.
   - **One-way ANOVA**: Tests the effect of one factor.
   - **Two-way ANOVA**: Tests the effect of two factors and their interaction.
3. **Chi-Square Test**:
   Tests the association between categorical variables.
   - **Chi-Square Test of Independence**: Determines if there is a relationship between two categorical variables.
   - **Goodness-of-Fit Test**: Compares the observed data to an expected distribution.
4. **Regression Analysis**:
   Models the relationship between a dependent variable and one or more independent variables. It is used to predict outcomes and understand the strength of relationships.

Types of Errors in Statistical Experimentation

1. **Type I Error (False Positive)**:
   Rejecting the null hypothesis when it is actually true (finding a difference when none exists). This error is controlled by the significance level (α).
2. **Type II Error (False Negative)**:
   Failing to reject the null hypothesis when it is false (not finding a difference when one exists). This error is controlled by the power of the test ($1 - \beta$).

Design of Experiments (DOE)

Description:

Design of Experiments (DOE) is a systematic method used to determine the relationship between factors affecting a process and the output of that process. It helps in understanding the cause-and-effect relationships and is widely applied in experimental research and quality improvement projects.

Key Components of DOE:

Factors:

These are the input variables or conditions that are changed in the experiment to observe their effect on the output. Each factor can have multiple levels or values.

Example: Temperature, pressure, or concentration in a chemical experiment.

Response:

The measurable outcome or the result that is affected by changes in factors. It could be a quantity such as yield, time, or defect rate.

Example: Product yield or process efficiency.

Treatments:

Combinations of factor levels that are applied during the experiment. Each treatment represents a different set of conditions.

Example: High temperature and low pressure as one treatment, low temperature and high pressure as another.

Randomization:

Ensures that the experimental conditions are applied in a random order to avoid bias or the influence of uncontrolled variables.

Replication:

The repetition of the experiment under the same conditions to estimate the variability in the results and increase the reliability of conclusions.

Blocking:

A technique used to account for known sources of variability that are not of primary interest. The experiment is divided into blocks where conditions are relatively homogeneous.

Example: Performing the experiment at different times of the day to block the effect of time on the outcome.

Types of Experimental Designs:

Full Factorial Design:

All possible combinations of factor levels are tested. This design is thorough but can be resource-intensive with many factors.

Use Case: When it's critical to understand every interaction between factors.

Example: For two factors (A and B), each with two levels (low, high), there are 4 combinations:

(A low, B low), (A low, B high), (A high, B low), (A high, B high)

(A low, B low), (A low, B high), (A high, B low), (A high, B high)

Fractional Factorial Design:

A reduced version of a full factorial design where only a subset of the factor combinations is tested. It is useful when resources are limited, but it may not capture all interactions.

Use Case: When you have a large number of factors and want to test the most important combinations.

Randomized Block Design:

A design that accounts for the variability between different blocks or groups of experimental units by randomizing within blocks.

Use Case: To control for known sources of variability, such as different production shifts.

Response Surface Methodology (RSM):

An advanced technique used to model the relationship between the factors and response when the relationship is complex and non-linear.

Use Case: When seeking the optimal levels of factors to maximize or minimize a response.

Steps in Designing an Experiment:

Define the objective:

Clearly state the problem and what you aim to achieve (e.g., improving product quality, optimizing a process).

Identify factors and levels:

Select the key variables (factors) that will be tested, and determine the levels (e.g., high/low) of each factor.

Choose a design type:

Decide between full factorial, fractional factorial, randomized block design, etc., based on available resources and the complexity of the system.

Conduct the experiment:

Perform the experiment according to the chosen design, randomizing and replicating where necessary.

Analyze the data:

Use statistical tools (e.g., Analysis of Variance - ANOVA) to identify the effects of factors and interactions on the response.

Interpret results and optimize:

Draw conclusions about the factors that significantly influence the response and use this knowledge to optimize the process.

Use Cases of DOE:

- **Manufacturing**: To improve product quality by optimizing factors such as temperature, time, and materials.
- **Pharmaceuticals**: To determine the optimal combination of ingredients in a drug formulation.
- **Marketing**: To test the impact of different pricing strategies and advertising campaigns on sales.