#UpSkillWithKalpesh

Day 10

Data Science Unlocked

From Zero to Data Hero

Pandas for Data Science – Part 1

Kalpesh Pathade
@DataSimplified

Pandas for Data Science - Part 1

Type	Data science masterclass

I. Introduction to Pandas

1.1 What is Pandas?

Overview and History

- **Pandas** is a powerful and flexible open-source data manipulation and analysis library for Python.
- **Origins:**
  - Created by **Wes McKinney** in 2008 while he was working at AQR Capital.
  - Initially developed to address the need for high-performance, easy-to-use data structures for financial data analysis.
- **Evolution:**
  - Over time, Pandas has grown into one of the cornerstone libraries in the data science ecosystem.
  - It has become widely adopted in academia, industry, and research due to its efficiency and ease of use.

Importance in Data Science

- **Data Structures:**
  - **Series:** A one-dimensional labeled array capable of holding any data type.
  - **DataFrame:** A two-dimensional, size-mutable, heterogeneous tabular data structure with labeled axes (rows and columns).
- **Core Functionality:**

    - Simplifies tasks such as data cleaning, transformation, aggregation, and visualization.
    - Provides intuitive and powerful tools for data indexing, slicing, and reshaping.
- **Productivity and Performance:**
    - Optimized for performance with many operations vectorized (leveraging NumPy under the hood).
    - Facilitates rapid prototyping and analysis with concise and readable syntax.
- **Real-World Application:**
    - Essential for exploratory data analysis (EDA), feature engineering, and preparation of data for machine learning models.
    - Widely used in sectors like finance, healthcare, marketing, and research.

1.2 Installation and Setup

Installing via pip/conda

- **Using pip:**
    - Open your terminal or command prompt and run:

      ```
      pip install pandas
      ```

    - This command downloads and installs Pandas along with its dependencies (like NumPy).
- **Using conda:**
    - If you are using the Anaconda distribution, run:

      ```
      conda install pandas
      ```

    - Conda automatically manages the environment and dependencies.

- **Verification:**
    - After installation, open a Python interpreter or script and type:

      ```
      import pandas as pd
      print(pd.__version__)
      ```

    - This should display the installed version of Pandas, confirming a successful installation.

Setting up Your Development Environment (Jupyter Notebooks, IDEs)

- **Jupyter Notebooks:**
    - **Why Jupyter?**
        - Interactive, browser-based environment ideal for data analysis and visualization.
        - Allows for inline code execution, rich text annotations, and visual output display.
    - **Installation:**
        - Install via pip:

          ```
          pip install notebook
          ```

        - Launch by typing `jupyter notebook` in your terminal.
    - **Usage:**
        - Create notebooks that mix code, visualizations, and markdown text to document your analysis.
- **Integrated Development Environments (IDEs):**
    - **Popular IDEs:**
        - Visual Studio Code (with Python extension)
        - PyCharm

        - Spyder
    - **Setup Tips:**
        - Ensure the IDE is configured to use the correct Python interpreter where Pandas is installed.
        - Many IDEs offer features like interactive consoles, debugging, and integrated terminal support.
        - Some IDEs (e.g., VS Code) now support notebook-like interfaces or interactive Python sessions.
- **Best Practices:**
    - **Virtual Environments:**
        - Create a virtual environment for your projects using `venv`, `virtualenv`, or conda environments.
        - This practice isolates your project dependencies and avoids version conflicts.
    - **Dependency Management:**
        - Maintain a `requirements.txt` (for pip) or `environment.yml` (for conda) file to ensure reproducibility.
        - Example for pip:

```
pip freeze > requirements.txt
```

1.3 Pandas in the Data Science Ecosystem

Integration with NumPy, Matplotlib, Seaborn, etc.

- **NumPy:**
    - Pandas is built on top of **NumPy** and leverages its powerful array computations.
    - Many Pandas operations are vectorized, enabling high-performance calculations on large datasets.

    - NumPy arrays are used internally to store data in Series and DataFrames, providing speed and efficiency.
- **Matplotlib:**
    - **Visualization:**
        - Pandas integrates seamlessly with **Matplotlib**, enabling quick plotting directly from DataFrames.
        - Example: `df.plot()` produces a variety of plot types (line, bar, histograms) with minimal code.
    - **Customization:**
        - For more advanced visualizations, you can combine Pandas' built-in plotting with Matplotlib's customization options.
- **Seaborn:**
    - **Enhanced Statistical Plots:**
        - **Seaborn** is built on top of Matplotlib and works well with Pandas DataFrames.
        - It provides a high-level interface for creating informative and attractive statistical graphics.
    - **Use Cases:**
        - Ideal for visualizing distributions, relationships, and trends in data with less boilerplate code.
- **Other Libraries:**
    - **Scikit-learn:**
        - Used for machine learning, Pandas is often the go-to tool for preprocessing and cleaning data before feeding it into ML models.
    - **Statsmodels:**
        - Useful for statistical modeling and hypothesis testing, often using data processed with Pandas.
    - **Dask:**

        - For handling larger-than-memory datasets, Dask provides a Pandas-like interface that scales computations.
    - **Plotly:**
        - For interactive visualizations, Plotly works well with Pandas data structures, offering dynamic plots for web applications.

Use Cases in Data Analysis and Machine Learning

- **Data Cleaning and Preparation:**
    - **Missing Data:**
        - Pandas provides functions to detect, handle, and impute missing data.
        - Techniques include using methods like `dropna()`, `fillna()`, and interpolation.
    - **Data Transformation:**
        - Functions for converting data types, normalizing, and transforming data are readily available.
        - String manipulation and categorical data management are simplified with Pandas.
- **Exploratory Data Analysis (EDA):**
    - **Descriptive Statistics:**
        - Quickly generate summary statistics with methods like `describe()`, `info()`, and `value_counts()`.
    - **Data Visualization:**
        - Use plotting functions to explore data distributions, detect outliers, and identify trends.
        - Integrated plotting with Matplotlib and Seaborn enhances the exploratory process.
- **Feature Engineering:**
    - **Creation of New Features:**

        - Use Pandas to derive new columns based on existing data (e.g., extracting date parts, computing ratios).
    - **Transformation Techniques:**
        - Apply transformations such as scaling, encoding categorical variables, and creating interaction features.
    - **Time Series Handling:**
        - Pandas offers robust tools for dealing with date/time data, making it a natural choice for time series analysis.
- **Machine Learning Workflows:**
    - **Data Preprocessing:**
        - Clean and prepare datasets for training machine learning models.
        - Merge, join, and reshape data to fit model input requirements.
    - **Pipeline Integration:**
        - Combine with libraries like Scikit-learn to create seamless workflows from data cleaning to model evaluation.
    - **Visualization of Model Results:**
        - Use Pandas alongside visualization libraries to interpret and communicate model performance and insights.

II. Core Data Structures

Understanding Pandas' core data structures is fundamental to effectively manipulating and analyzing data. The primary structures are **Series**, **DataFrame**, and **Index Objects**.

2.1 Series

A **Series** is a one-dimensional labeled array capable of holding any data type (integers, floats, strings, Python objects, etc.). It is similar to a one-dimensional NumPy array but comes with additional functionalities like custom indexing.

Creation and Basic Operations

- **Creating a Series:**
  - **From a List:**

    ```
    import pandas as pd
    s = pd.Series([10, 20, 30, 40])
    print(s)
    ```

  - **From a Dictionary:**

    ```
    data = {'a': 1, 'b': 2, 'c': 3}
    s = pd.Series(data)
    print(s)
    ```

- **Accessing Data:**
  - **Values:**

    ```
    print(s.values)  # Returns the underlying numpy array
    ```

  - **Index:**

    ```
    print(s.index)  # Returns the index labels
    ```

- **Basic Operations:**
  - **Vectorized Operations:** Series operations are applied element-wise.

    ```
    s2 = s * 2  # Each element is multiplied by 2
    print(s2)
    ```

  - **Common Methods:**

        - `s.head()` – returns the first few elements.
        - `s.tail()` – returns the last few elements.
        - `s.describe()` – provides summary statistics.

Indexing, Slicing, and Arithmetic Operations

- **Indexing:**
    - **Label-based Indexing:** Access elements using their index labels.

    ```
    s = pd.Series([100, 200, 300], index=['a', 'b', 'c'])
    print(s['a'])  # Accesses the element with label 'a'
    ```

    - **Position-based Indexing:** Use `.iloc` for positional indexing.

    ```
    print(s.iloc[0])  # Accesses the first element
    ```

- **Slicing:**
    - **Label Slicing:** The slice is inclusive of the end label.

    ```
    print(s['a':'b'])  # Returns elements from 'a' to 'b' (inclusive)
    ```

    - **Position Slicing:** Standard Python slicing rules (end-exclusive).

    ```
    print(s.iloc[0:2])  # Returns the first two elements
    ```

- **Arithmetic Operations:**
    - **Element-wise Arithmetic:** Operations align by index labels.

    ```
    s1 = pd.Series([1, 2, 3])
    s2 = pd.Series([4, 5, 6])
    result = s1 + s2  # [5, 7, 9]
    print(result)
    ```

  - **Alignment:** If indices do not match, the result contains `NaN` for non-matching labels.

    ```
    s3 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
    s4 = pd.Series([4, 5, 6], index=['b', 'c', 'd'])
    print(s3 + s4)  # 'a' and 'd' will have NaN
    ```

2.2 DataFrame

A **DataFrame** is a two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes (rows and columns). It is one of the most commonly used structures in Pandas for data manipulation and analysis.

Creating DataFrames from Various Sources

- **From a List of Lists:**

  ```
  data = [
      [1, 2, 3],
      [4, 5, 6]
  ]
  df = pd.DataFrame(data, columns=['A', 'B', 'C'])
  print(df)
  ```

- **From a Dictionary:**

  ```
  data = {
      'Name': ['Alice', 'Bob', 'Charlie'],
      'Age': [25, 30, 35],
      'City': ['New York', 'Paris', 'London']
  }
  df = pd.DataFrame(data)
  print(df)
  ```

- **From a CSV File:**

```
df = pd.read_csv('path/to/your/file.csv')
print(df.head())
```

- **Other Sources:**
  - Excel files via `pd.read_excel()`
  - SQL databases via `pd.read_sql()`

Understanding Rows, Columns, and Indexes

- **Rows and Columns:**
  - **Rows:** Represent individual records or observations.
  - **Columns:** Represent variables or features.
- **Index:**
  - **Default Index:** By default, DataFrame rows are labeled with a RangeIndex starting at 0.
  - **Custom Index:** You can set a custom index from one of the columns.

    ```
    df.set_index('Name', inplace=True)
    print(df)
    ```

  - **Accessing Columns:**
    - Using dictionary-like notation:

      ```
      print(df['Age'])
      ```

    - Using attribute notation (if column names are valid identifiers):

      ```
      print(df.Age)
      ```

- **Additional Metadata:**
  - **Shape of DataFrame:** `df.shape` returns a tuple (rows, columns).

  - **Column Labels:** `df.columns` returns an index object containing the column names.
  - **Row Labels:** `df.index` returns the index object for rows.

2.3 Index Objects

Indexes in Pandas play a crucial role in aligning data, selecting subsets, and enhancing performance through optimized lookups.

Role of Indexes in Data Selection and Alignment

- **Indexing and Alignment:**
  - Index objects hold the labels for the rows (and columns) and are immutable.
  - They enable efficient lookups, reindexing, and alignment during arithmetic operations.
  - When combining or performing operations on Series/DataFrames, Pandas aligns data based on index labels.
- **Data Selection:**
  - Indexes simplify data selection, allowing you to use labels to retrieve data.

    ```
    # Assuming a DataFrame with a custom index:
    df.loc['Alice']
    ```

  - They facilitate operations such as merging, joining, and reshaping by ensuring proper alignment of data.

Setting and Resetting Indexes

- **Setting an Index:**
  - Use the `set_index` method to designate one (or more) columns as the index.

    ```
    df = pd.DataFrame({
        'id': [1, 2, 3],
    ```

```
    'name': ['Alice', 'Bob', 'Charlie']
})
df.set_index('id', inplace=True)
print(df)
```

- **Benefits:**
    - Improves performance for lookup operations.
    - Provides a unique identifier for each row.

- **Resetting an Index:**
  - To revert the index back to a default integer index, use `reset_index`.

    ```
    df.reset_index(inplace=True)
    print(df)
    ```

  - **When to Reset:**
    - When you need the index values as a regular column again.
    - When preparing data for operations that require a default index.

- **Hierarchical Indexes (MultiIndex):**
  - Pandas supports MultiIndex for handling higher-dimensional data in a 2D DataFrame.

    ```
    df = pd.DataFrame({
        'state': ['CA', 'CA', 'NY', 'NY'],
        'city': ['San Francisco', 'Los Angeles', 'New York', 'Buffalo'],
        'population': [883305, 3990456, 8398748, 261310]
    })
    df = df.set_index(['state', 'city'])
    print(df)
    ```

  - **Usage:**
    - Useful when working with data that has a natural hierarchical structure.

        - Provides more advanced slicing and grouping capabilities.

III. Data Input/Output

Pandas provides a rich set of functions for reading from and writing to various data formats. This section covers the essentials for loading data from common file types and exporting your DataFrames, along with best practices for handling different file encodings, delimiters, and file management.

3.1 Reading Data

Pandas makes it straightforward to load data from various sources. Below are the common methods and best practices for reading data.

Loading CSV, Excel, JSON, and SQL Data

- **CSV Files:**
    - **Basic Usage:**

      ```
      import pandas as pd

      # Read a CSV file into a DataFrame
      df_csv = pd.read_csv('path/to/your/file.csv')
      print(df_csv.head())
      ```

    - **Parameters:**
        - `sep`: Specify a delimiter other than a comma.

          ```
          df_csv = pd.read_csv('path/to/your/file.csv', sep=';')
          ```

        - `header`: Define the row number to use as column names (default is 0).

          ```
          df_csv = pd.read_csv('path/to/your/file.csv', header=0)
          ```

- **Excel Files:**
  - **Basic Usage:**

    ```
    # Read an Excel file into a DataFrame
    df_excel = pd.read_excel('path/to/your/file.xlsx')
    print(df_excel.head())
    ```

  - **Specifying a Sheet:**
    - Use the `sheet_name` parameter to read a particular sheet.

      ```
      df_excel = pd.read_excel('path/to/your/file.xlsx', sheet_name='Sheet1')
      ```

- **JSON Files:**
  - **Basic Usage:**

    ```
    # Read a JSON file into a DataFrame
    df_json = pd.read_json('path/to/your/file.json')
    print(df_json.head())
    ```

  - **Handling Complex JSON:**
    - For nested JSON, consider using the `json_normalize` function from Pandas.

      ```
      from pandas import json_normalize
      import json

      # Load JSON data
      with open('path/to/your/file.json') as f:
          data = json.load(f)

      df_json_nested = json_normalize(data)
      print(df_json_nested.head())
      ```

- **SQL Databases:**
    - **Basic Usage:**

      ```
      import sqlite3

      # Establish a connection to the SQLite database
      conn = sqlite3.connect('path/to/your/database.db')

      # Read data using a SQL query
      query = "SELECT * FROM your_table"
      df_sql = pd.read_sql(query, conn)
      print(df_sql.head())

      # Always close the connection when done
      conn.close()
      ```

    - **Other Databases:**
        - For databases like PostgreSQL or MySQL, you may use libraries such as `SQLAlchemy` for connection management.

Handling Different File Encodings and Delimiters

- **File Encodings:**
    - Some files might use encodings other than the default UTF-8.

      ```
      df_csv = pd.read_csv('path/to/your/file.csv', encoding='ISO-8859-1')
      ```

    - Always verify the file encoding if you encounter errors related to character decoding.
- **Custom Delimiters:**
    - When dealing with files that use non-standard delimiters (e.g., semicolons, tabs):

```
# For a semicolon-delimited file
df_csv = pd.read_csv('path/to/your/file.csv', sep=';')

# For a tab-delimited file
df_csv = pd.read_csv('path/to/your/file.csv', sep='\t')
```

- **Parsing Dates:**
    - If your data contains dates, you can instruct Pandas to parse them:

    ```
    df_csv = pd.read_csv('path/to/your/file.csv', parse_dates=['date_column'])
    ```

3.2 Writing Data

After processing and analyzing data with Pandas, you'll often need to export your DataFrame to share your results or for further use. Here are the common methods for writing data.

Exporting DataFrames to CSV, Excel, and Other Formats

- **Exporting to CSV:**
    - **Basic Export:**

    ```
    # Write DataFrame to a CSV file
    df_csv.to_csv('path/to/save/file.csv', index=False)
    ```

    - **Additional Options:**
        - `index=False` prevents Pandas from writing row indices to the file.
        - `sep` can be used to specify a different delimiter.

        ```
        df_csv.to_csv('path/to/save/file.csv', sep=';', index=False)
        ```

- **Exporting to Excel:**

- **Basic Export:**

  ```
  # Write DataFrame to an Excel file
  df_excel.to_excel('path/to/save/file.xlsx', index=False)
  ```

- **Multiple Sheets:**
  - Use `ExcelWriter` to export multiple DataFrames to different sheets within the same workbook.

    ```
    with pd.ExcelWriter('path/to/save/file.xlsx') as writer:
        df1.to_excel(writer, sheet_name='Sheet1', index=False)
        df2.to_excel(writer, sheet_name='Sheet2', index=False)
    ```

- **Exporting to JSON:**
  - **Basic Export:**

    ```
    # Write DataFrame to a JSON file
    df_json.to_json('path/to/save/file.json', orient='records', lines=True)
    ```

  - **Parameters:**
    - `orient`: Controls the format of the JSON string. Common values include `'records'`, `'split'`, `'index'`, etc.
    - `lines=True`: Writes JSON objects separated by newline characters, which is useful for large datasets.
- **Other Formats:**
  - **HTML:**

    ```
    df_html.to_html('path/to/save/file.html')
    ```

  - **Pickle:**

    ```
    df_pickle.to_pickle('path/to/save/file.pkl')
    ```

- **Parquet:**

  ```
  df_parquet.to_parquet('path/to/save/file.parquet')
  ```

Best Practices for Data Export and File Handling

- **File Naming and Organization:**
  - Use descriptive file names and organize exports in a structured folder hierarchy.
  - Consider adding timestamps or version numbers to file names for version control.
- **Handling Large Datasets:**
  - For very large datasets, consider exporting to binary formats like Parquet or using compression.

    ```
    df_csv.to_csv('path/to/save/file.csv.gz', compression='gzip', index=False)
    ```

  - When working with large Excel files, be aware of Excel's row and column limitations.
- **Ensuring Data Integrity:**
  - Always verify the output file by reloading it to ensure data integrity.

    ```
    df_check = pd.read_csv('path/to/save/file.csv')
    print(df_check.head())
    ```

- **Avoiding Data Loss:**
  - Be cautious when using parameters like `index=False` if the row index holds important information.
  - If your DataFrame contains sensitive data, ensure that files are stored in secure directories with proper access controls.
- **Documentation:**

- Comment your export code to indicate the purpose of each file, especially when working in collaborative environments.

IV. Data Exploration and Summary

Effective data exploration is a critical first step in any data analysis workflow. Pandas offers several built-in methods to inspect your DataFrames and quickly understand the underlying structure and summary statistics. Additionally, Pandas integrates well with visualization libraries like Matplotlib and Seaborn, enabling you to visualize your data effortlessly.

4.1 Inspecting DataFrames

Inspecting your DataFrame helps you understand its structure, data types, and the presence of missing values. Here are some key methods:

Using `.head()`, `.tail()`, and `.info()`

- `.head()`
  - Displays the first few rows of the DataFrame.
  - **Default:** Returns the first 5 rows.
  - **Example:**

```
import pandas as pd

# Sample DataFrame creation
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Age': [25, 30, 35, 40, 45, 50],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix', 'P
hiladelphia']
})
```

```
# Display the first 5 rows
print(df.head())
```

- `.tail()`
    - Displays the last few rows of the DataFrame.
    - **Default:** Returns the last 5 rows.
    - **Example:**

    ```
    # Display the last 5 rows
    print(df.tail())
    ```

- `.info()`
    - Provides a concise summary of the DataFrame.
    - **Details include:**
        - Number of rows and columns.
        - Data types of each column.
        - Non-null count for each column.
        - Memory usage.
    - **Example:**

    ```
    # Display summary information about the DataFrame
    df.info()
    ```

Descriptive Statistics with `.describe()`

- **Purpose:**
    - Offers a quick statistical summary for numeric columns.
    - **Statistics include:**
        - Count, mean, standard deviation, min, max, and quartiles.
    - **Example:**

```
# Display descriptive statistics for numeric columns
print(df.describe())
```

- **Additional Options:**
    - To include all columns (numeric and non-numeric):

    ```
    print(df.describe(include='all'))
    ```

    - For percentiles other than the default quartiles, you can specify:

    ```
    print(df.describe(percentiles=[0.1, 0.5, 0.9]))
    ```

4.2 Data Visualization Integration

Visualization is a powerful tool for understanding patterns, distributions, and relationships in your data. Pandas provides basic plotting functions, and its integration with libraries like Matplotlib and Seaborn enhances visualization capabilities.

Basic Plotting with Pandas

- **Built-in Plotting:**
    - DataFrames and Series come with a built-in `.plot()` method that acts as a wrapper around Matplotlib.
    - **Example – Line Plot:**

    ```
    import matplotlib.pyplot as plt

    # Create a simple DataFrame
    df_plot = pd.DataFrame({
        'Year': [2016, 2017, 2018, 2019, 2020],
        'Sales': [250, 300, 350, 400, 450]
    })
    ```

```
# Set the 'Year' column as the index for better plotting
df_plot.set_index('Year', inplace=True)

# Create a line plot
df_plot.plot(kind='line', marker='o')
plt.title('Yearly Sales Trend')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.grid(True)
plt.show()
```

- **Other Plot Types:**
  - **Bar Plot:**

    ```
    df_plot.plot(kind='bar', legend=False)
    plt.title('Sales by Year')
    plt.xlabel('Year')
    plt.ylabel('Sales')
    plt.show()
    ```

  - **Histogram:**

    ```
    df_plot['Sales'].plot(kind='hist', bins=5, alpha=0.7)
    plt.title('Distribution of Sales')
    plt.xlabel('Sales')
    plt.show()
    ```

  - **Scatter Plot:**

    ```
    # For scatter plot, ensure you have two numerical columns
    df_scatter = pd.DataFrame({
        'Advertising': [10, 20, 30, 40, 50],
        'Sales': [15, 25, 35, 45, 55]
    })
    ```

```
df_scatter.plot(kind='scatter', x='Advertising', y='Sales')
plt.title('Sales vs. Advertising')
plt.xlabel('Advertising Budget')
plt.ylabel('Sales')
plt.show()
```

Integration with Matplotlib/Seaborn for Enhanced Visualizations

- **Matplotlib Customizations:**
  - Since Pandas plotting uses Matplotlib under the hood, you can use Matplotlib functions to further customize your plots.
  - **Example – Customizing Plot Appearance:**

```
ax = df_plot.plot(kind='line', marker='o', figsize=(8, 5))
ax.set_title('Yearly Sales Trend', fontsize=16)
ax.set_xlabel('Year', fontsize=12)
ax.set_ylabel('Sales', fontsize=12)
ax.grid(True)
plt.show()
```

- **Seaborn Integration:**
  - Seaborn is a statistical data visualization library built on top of Matplotlib.
  - It works seamlessly with Pandas DataFrames and provides attractive and informative plots.
  - **Example – Seaborn Bar Plot:**

```
import seaborn as sns

# Using the same df_plot DataFrame
sns.set(style="whitegrid")
sns.barplot(x=df_plot.index, y='Sales', data=df_plot.reset_index())
plt.title('Sales by Year')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.show()
```

- **Additional Seaborn Examples:**
  - **Box Plot:**

```
# Create a sample DataFrame with multiple groups
df_box = pd.DataFrame({
    'Category': ['A', 'A', 'B', 'B', 'C', 'C'],
    'Value': [10, 15, 10, 20, 15, 25]
})

sns.boxplot(x='Category', y='Value', data=df_box)
plt.title('Box Plot of Values by Category')
plt.show()
```

  - **Heatmap:**

```
# Create a correlation matrix for demonstration
df_corr = pd.DataFrame({
    'A': [1, 2, 3, 4, 5],
    'B': [5, 4, 3, 2, 1],
    'C': [2, 3, 4, 3, 2]
})
corr = df_corr.corr()
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

V. Data Cleaning and Preprocessing

Data cleaning and preprocessing are critical steps in any data analysis or machine learning workflow. Clean, well-structured data leads to more reliable insights and robust models. This section covers four key areas:

5.1 Handling Missing Data

Missing data can occur for various reasons—errors during data collection, manual entry mistakes, or system issues. Pandas provides powerful methods to detect, count, and handle missing values.

Identifying and Counting Missing Values

- **Detecting Missing Values:**
  - **`isnull()` / `isna()`:**
    - Returns a DataFrame or Series of Boolean values (`True` if the value is missing).
    - **Example:**

```
import pandas as pd

# Sample DataFrame with missing values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', None],
    'Age': [25, None, 35, 40],
    'Salary': [50000, 60000, None, 80000]
}
df = pd.DataFrame(data)

# Detect missing values
missing_mask = df.isnull()  # or df.isna()
print("Missing Values Mask:")
print(missing_mask)
```

- **Counting Missing Values:**
  - **Per Column:**

        - Use `df.isnull().sum()` to count missing values in each column.

          ```
          missing_counts = df.isnull().sum()
          print("Missing Values Per Column:")
          print(missing_counts)
          ```

    - **Per Row:**
        - Count missing values across rows using `axis=1`.

          ```
          missing_per_row = df.isnull().sum(axis=1)
          print("Missing Values Per Row:")
          print(missing_per_row)
          ```

Techniques for Imputation or Removal

- **Removing Missing Data:**
    - `dropna()` **Method:**
        - **Drop Rows:** Remove rows that contain any missing values.

          ```
          df_drop_rows = df.dropna()
          print("DataFrame After Dropping Rows with Any Missing Values:")
          print(df_drop_rows)
          ```

        - **Drop Columns:** Remove columns that contain any missing values.

          ```
          df_drop_cols = df.dropna(axis=1)
          print("DataFrame After Dropping Columns with Any Missing Value
          s:")
          print(df_drop_cols)
          ```

        - **Threshold Option:**
            - Keep rows (or columns) that have at least a minimum number of non-missing values.

```
df_thresh = df.dropna(thresh=2)  # Keep rows with at least 2 non-n
ull values
print("DataFrame After Dropping Rows Not Meeting the Threshol
d:")
print(df_thresh)
```

- **Filling Missing Data:**
  - `fillna()` **Method:**
    - **Fill with a Constant Value:**

      ```
      df_filled_const = df.fillna(0)
      print("DataFrame After Filling Missing Values with 0:")
      print(df_filled_const)
      ```

    - **Forward Fill (Propagate Last Valid Observation):**

      ```
      df_ffill = df.fillna(method='ffill')
      print("DataFrame After Forward Fill:")
      print(df_ffill)
      ```

    - **Backward Fill (Use Next Valid Observation):**

      ```
      df_bfill = df.fillna(method='bfill')
      print("DataFrame After Backward Fill:")
      print(df_bfill)
      ```

    - **Fill with a Computed Value (e.g., Mean or Median):**

      ```
      # Filling missing values in the 'Age' column with the mean age
      mean_age = df['Age'].mean()
      df['Age'] = df['Age'].fillna(mean_age)
      print("DataFrame After Filling Missing 'Age' with Mean:")
      print(df)
      ```

- **Interpolation:**
  - `interpolate()` **Method:**
    - Estimate missing values using various interpolation methods (e.g., linear, time-based).

      ```
      # For a time series or ordered numerical data
      df_interpolated = df.interpolate(method='linear')
      print("DataFrame After Linear Interpolation:")
      print(df_interpolated)
      ```

Best Practices and Considerations

- **Understand the Data Context:**
  - Assess the significance of missing values and decide whether removal or imputation is appropriate.
- **Document Imputation Strategies:**
  - Keep records of the methods used (e.g., why a particular value was chosen) to ensure reproducibility.
- **Combine Strategies if Needed:**
  - Sometimes, a combination of dropping and imputing is necessary based on the nature and volume of missing data.
- **Mind Data Types:**
  - Ensure that the chosen method (especially imputation) respects the data types in your DataFrame.

5.2 Data Transformation

Data transformation involves converting data into a format suitable for analysis. This includes converting data types, formatting values, and applying functions to modify or derive new features.

Changing Data Types and Formatting

- **Type Conversion:**
  - **Using `astype()`:**
    - Convert a column to a specific data type.

      ```
      # Convert 'Age' from float to integer (if appropriate)
      df['Age'] = df['Age'].astype('int', errors='ignore')
      print("DataFrame After Converting 'Age' to int:")
      print(df.dtypes)
      ```

  - **Datetime Conversion:**
    - Convert strings or numeric values to datetime objects using `pd.to_datetime()`.

      ```
      # Example DataFrame with date strings
      df_dates = pd.DataFrame({
          'date_str': ['2023-01-01', '2023-02-15', '2023-03-30']
      })
      df_dates['date'] = pd.to_datetime(df_dates['date_str'], format='%Y
      -%m-%d')
      print("DataFrame with Converted Datetime:")
      print(df_dates)
      ```

  - **Categorical Data:**
    - Convert columns to the 'category' dtype for memory efficiency and performance.

      ```
      df['Name'] = df['Name'].astype('category')
      print("DataFrame After Converting 'Name' to Category:")
      print(df.dtypes)
      ```

Applying Functions to Columns and Rows

- **Using `apply()`:**

- **Column-wise Application:**
  - Apply a function to each element in a column.

    ```
    # Define a simple function to double a number
    def double_value(x):
        return x * 2

    # Apply the function to the 'Age' column
    df['Age_doubled'] = df['Age'].apply(double_value)
    print("DataFrame After Doubling 'Age':")
    print(df)
    ```

- **Row-wise Application:**
  - Use `axis=1` to apply a function across each row.

    ```
    # Define a function that combines multiple columns
    def combine_info(row):
        return f"{row['Name']} is {row['Age']} years old"

    # Apply the function row-wise
    df['Info'] = df.apply(combine_info, axis=1)
    print("DataFrame with Combined Information:")
    print(df)
    ```

- **Using `map()`:**
  - **Mapping Values in a Series:**
    - Replace values using a mapping dictionary or a function.

      ```
      # Create a mapping dictionary for renaming or transforming values
      mapping = {'Alice': 'Alicia', 'Bob': 'Robert', 'Charlie': 'Charles'}
      df['Name'] = df['Name'].map(mapping)
      print("DataFrame After Mapping Names:")
      print(df)
      ```

- **Using `applymap()` for DataFrames:**
    - **Element-wise Application on the Entire DataFrame:**
        - Useful for applying a function to every element.

        ```
        # For example, converting all numeric values to string representati
        ons with a suffix
        df_transformed = df.applymap(lambda x: str(x) + "_value" if isinsta
        nce(x, (int, float)) else x)
        print("DataFrame After Applying Element-wise Transformation:")
        print(df_transformed)
        ```

5.3 String Manipulation

Working with text data is common in preprocessing. Pandas provides a set of vectorized string functions under the `.str` accessor, enabling efficient string operations on Series.

Using Vectorized String Methods for Text Data

- **Basic Operations:**
    - **Lowercase / Uppercase:**

    ```
    df['Name_lower'] = df['Name'].str.lower()
    print("Names in Lowercase:")
    print(df['Name_lower'])
    ```

    - **Splitting Strings:**

    ```
    # Split the 'Name' column into lists based on whitespace
    df['Name_split'] = df['Name'].str.split()
    print("Names Split into Lists:")
    print(df['Name_split'])
    ```

    - **Replacing Substrings:**

```
df['Name_replaced'] = df['Name'].str.replace('Alicia', 'Alice', regex=False)
print("Names After Replacement:")
print(df['Name_replaced'])
```

- **Advanced Operations:**
  - **Extracting Substrings:**
    - Use `.str.extract()` with a regular expression to capture parts of strings.

    ```
    # Extract the first word from the 'Name' column
    df['First_Name'] = df['Name'].str.extract(r'(\w+)', expand=False)
    print("Extracted First Names:")
    print(df['First_Name'])
    ```

  - **String Length and Contains:**

  ```
  df['Name_length'] = df['Name'].str.len()
  df['Contains_a'] = df['Name'].str.contains('a', case=False)
  print("Name Lengths and Contains Check:")
  print(df[['Name_length', 'Contains_a']])
  ```

Regular Expressions in Pandas

- **Using Regex with `.str`:**
  - **Pattern Matching and Extraction:**

  ```
  # Example: Extract numeric values from a string column (assuming 'Salary' as string)
  df['Salary_str'] = df['Salary'].astype('str')
  df['Extracted_Salary'] = df['Salary_str'].str.extract(r'(\d+)', expand=False)
  print("Extracted Numeric Salary Values:")
  print(df[['Salary_str', 'Extracted_Salary']])
  ```

- **Replacing Patterns with Regex:**

  ```
  # Remove all non-numeric characters from 'Salary_str'
  df['Clean_Salary'] = df['Salary_str'].str.replace(r'\D', '', regex=True)
  print("Cleaned Salary Values:")
  print(df[['Salary_str', 'Clean_Salary']])
  ```

- **Finding All Matches:**

  ```
  # Find all digit sequences in 'Salary_str'
  df['All_Digits'] = df['Salary_str'].str.findall(r'\d+')
  print("All Digit Sequences in Salary:")
  print(df[['Salary_str', 'All_Digits']])
  ```

Best Practices for String Manipulation

- **Use Vectorized Methods:**
  - Always prefer the `.str` accessor over Python loops for efficiency.
- **Test Regex Patterns:**
  - Utilize regex testing tools to ensure your patterns work as intended.
- **Handle Missing Data:**
  - Consider filling missing values (e.g., with empty strings) before applying string operations if necessary.

5.4 Dealing with Outliers

Outliers are data points that differ significantly from other observations. They can skew analyses and adversely affect model performance. Detecting and treating outliers is essential for robust data analysis.

Detection Methods

- **Statistical Methods:**
  - **Z-Score:**

        - Calculate the Z-score to determine how many standard deviations a value is from the mean.

```
from scipy import stats
import numpy as np

# Convert 'Salary' to numeric (if not already)
df['Salary_numeric'] = pd.to_numeric(df['Salary'], errors='coerce')

# Calculate Z-scores (ignoring missing values)
df['Z_score'] = np.abs(stats.zscore(df['Salary_numeric'].dropna()))

# Identify outliers where Z-score > 3 (common threshold)
outliers_z = df[df['Z_score'] > 3]
print("Outliers Detected Using Z-score:")
print(outliers_z)
```

    - **Interquartile Range (IQR):**
        - Compute the IQR and flag values below Q1 - 1.5×IQR or above Q3 + 1.5×IQR.

```
Q1 = df['Salary_numeric'].quantile(0.25)
Q3 = df['Salary_numeric'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

outliers_iqr = df[(df['Salary_numeric'] < lower_bound) | (df['Salary_
numeric'] > upper_bound)]
print("Outliers Detected Using IQR:")
print(outliers_iqr)
```

- **Visual Methods:**
    - **Box Plots:**

    - Visualize distributions and easily spot outliers.

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

      sns.boxplot(x=df['Salary_numeric'])
      plt.title("Box Plot of Salary")
      plt.show()
      ```

  - **Scatter Plots:**
    - Identify outliers in the context of other variables.

      ```
      sns.scatterplot(data=df, x='Age', y='Salary_numeric')
      plt.title("Scatter Plot: Age vs. Salary")
      plt.show()
      ```

Strategies for Outlier Treatment

- **Removal:**
  - **Drop Outliers:**
    - Remove outlier rows if they are clearly erroneous or not relevant.

      ```
      df_no_outliers = df[(df['Salary_numeric'] >= lower_bound) & (df['S
      alary_numeric'] <= upper_bound)]
      print("DataFrame After Removing Outliers:")
      print(df_no_outliers)
      ```

- **Capping/Clipping:**
  - **Cap Extreme Values:**
    - Replace values beyond the threshold with the boundary value.

      ```
      df['Salary_capped'] = df['Salary_numeric'].clip(lower_bound, uppe
      r_bound)
      ```

```
print("DataFrame After Capping Outliers in Salary:")
print(df[['Salary_numeric', 'Salary_capped']])
```

- **Transformation:**
  - **Log Transformation:**
    - Apply a logarithmic transformation to reduce the impact of extreme values.

      ```
      df['Salary_log'] = np.log(df['Salary_numeric'].replace(0, np.nan))
      print("DataFrame After Log Transformation of Salary:")
      print(df[['Salary_numeric', 'Salary_log']])
      ```

- **Imputation:**
  - **Replace Outliers with Statistical Values:**
    - For some analyses, replacing outlier values with the median or mean may be preferable.

      ```
      median_salary = df['Salary_numeric'].median()
      df.loc[(df['Salary_numeric'] < lower_bound) | (df['Salary_numeric']
      > upper_bound), 'Salary_numeric'] = median_salary
      print("DataFrame After Replacing Outliers with Median Salary:")
      print(df[['Salary_numeric']])
      ```

Best Practices for Handling Outliers

- **Analyze Context:**
  - Not every outlier is an error; understand your data's domain context before treating them.
- **Visual Inspection:**
  - Always complement statistical methods with visualizations (box plots, scatter plots) to inspect outliers.
- **Document Changes:**

    - Record any modifications or removals to maintain transparency in your data preprocessing.
- **Sensitivity Analysis:**
    - Consider analyzing your data both with and without outlier treatment to understand their impact.