Pandas

Agenda

1. Advanced Data Manipulations
    - Merging DataFrames
    - Concatenation
    - GroupBy Operations
    - Pivot Tables
2. Working with Dates
    - Converting to DateTime
    - Date Operations
3. Input/Output Operations
    - Reading and Writing Files
4. Performance Optimization
    - Vectorization
    - Memory Usage Reduction
5. Visualization with Pandas
    - Basic Plotting
    - Advanced Visualization

Advanced Data Manipulations

Merging DataFrames

- Merging DataFrames in Pandas is a crucial operation for combining datasets from different sources or performing complex data analysis. Pandas provides several methods for merging DataFrames, each tailored to different types of merging operations.

1. Merging on Multiple Keys
    - You can merge on multiple columns by passing a list to the on parameter.

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Type': ['A', 'B', 'C'], 'Value': [10, 20, 30]})
df2 = pd.DataFrame({'ID': [1, 2, 4], 'Type': ['A', 'B', 'D'], 'Amount': [100, 200, 400]})
df_merged = pd.merge(df1, df2, on=['ID', 'Type'])
```

2. Merging with Different Column Names
    - If the columns to merge on have different names in each DataFrame, use left_on and right_on.

```
df1 = pd.DataFrame({'ID_1': [1, 2, 3], 'Value': [10, 20, 30]})
df2 = pd.DataFrame({'ID_2': [1, 2, 4], 'Amount': [100, 200, 400]})
df_merged = pd.merge(df1, df2, left_on='ID_1', right_on='ID_2')
```

3. Merging with Suffixes

- When merging DataFrames with overlapping column names, use the suffixes parameter to differentiate them.

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Value': [10, 20, 30]})
df2 = pd.DataFrame({'ID': [1, 2, 3], 'Value': [100, 200, 300]})
df_merged = pd.merge(df1, df2, on='ID', suffixes=('_df1', '_df2'))
```

Concatenation

Concatenation is another way to combine DataFrames, particularly useful for appending rows or columns.

- Concatenate along Rows

```
df_concat = pd.concat([df1, df2], axis=0)  # Append rows of df2 to df1
```

- Concatenate along Columns

```
df_concat = pd.concat([df1, df2], axis=1)  # Append columns of df2 to df1
```

GroupBy operations

- GroupBy operations in Pandas are powerful tools for aggregating, transforming, and analyzing data based on specific criteria. They allow you to split data into groups, apply

functions to each group, and combine the results back into a DataFrame.

1. Basic GroupBy Operation
    - Syntax:
        - df.groupby('key_column')
    - Parameters:
        - 'key_column': The column or columns to group by.

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Alice', 'Bob', 'Alice'],
    'Age': [25, 30, 25, 30, 22],
    'Salary': [50000, 60000, 52000, 62000, 49000]
})

grouped = df.groupby('Name')
```

2. Aggregation Functions
    - After grouping, you can apply aggregation functions to summarize the data.
    - Common Aggregation Functions:
        - mean(): Mean of each group
        - sum(): Sum of each group
        - count(): Number of elements in each group
        - max(): Maximum value in each group
        - min(): Minimum value in each group

```
mean_salary = grouped['Salary'].mean()
```

3. Filtering
    - Filtering allows you to exclude groups based on some criteria.

```
filtered = df.groupby('Name').filter(lambda x: len(x) > 2)
```

Pivot Tables

- Pivot tables are an advanced way to perform GroupBy operations with multiple aggregations and columns.
- A pivot table is a data analysis tool that allows you to take columns of raw data from a pandas DataFrame, summarize them, and then analyze the summary data to reveal its insights.
- Pivot tables allow you to perform common aggregate statistical calculations such as sums, counts, averages, and so on.

```
# Create a simple dataframe

# importing pandas as pd
import pandas as pd
import numpy as np

# creating a dataframe
df = pd.DataFrame({'A': ['John', 'Boby', 'Mina', 'Peter', 'Nicky'],
      'B': ['Masters', 'Graduate', 'Graduate', 'Masters', 'Graduate'],
      'C': [27, 23, 21, 23, 24]})
df
```

	A	B	C
0	John	Masters	27
1	Boby	Graduate	23
2	Mina	Graduate	21
3	Peter	Masters	23
4	Nicky	Graduate	24

```
# Simplest pivot table must have a dataframe
# and an index/list of index.
table = pd.pivot_table(df, index =['A', 'B'])

table
```

		C
A	**B**	
Boby	**Graduate**	23.0
John	**Masters**	27.0
Mina	**Graduate**	21.0
Nicky	**Graduate**	24.0
Peter	**Masters**	23.0

```
# Creates a pivot table dataframe
table = pd.pivot_table(df, values ='A', index =['B', 'C'],
                       columns =['B'], aggfunc = np.sum)

table
```

```
<ipython-input-53-875b3a4c309a>:2: FutureWarning: The provided callable <function sum at
  table = pd.pivot_table(df, values ='A', index =['B', 'C'],
```

	B	Graduate	Masters
B	**C**		
Graduate	**21**	Mina	NaN
	23	Boby	NaN
	24	Nicky	NaN
Masters	**23**	NaN	Peter
	27	NaN	John

Working with Dates

- Pandas provide a different set of tools using which we can perform all the necessary tasks on date-time data. Let’s try to understand with the examples discussed below.

Converting to DateTime

- In Python, converting data to a datetime object is a common task when working with date and time information. Here's a guide on how to do it using the datetime module and pandas library.

```
import pandas as pd

# Example DataFrame
df = pd.DataFrame({
    'date_column': ['2024-08-17 15:30:00', '2024-08-18 16:45:00']
})

# Convert column to datetime
df['date_column'] = pd.to_datetime(df['date_column'])

print(df)
```

```
           date_column
0 2024-08-17 15:30:00
1 2024-08-18 16:45:00
```

Date Operations

- Extracting Date Parts
  - Extract specific parts of a date (e.g., year, month, day).

```
# Example DataFrame
df = pd.DataFrame({'dates': pd.to_datetime(['2024-08-17', '2024-08-18'])})

# Extract year, month, and day
df['year'] = df['dates'].dt.year
df['month'] = df['dates'].dt.month
df['day'] = df['dates'].dt.day

print(df)
```

```
       dates  year  month  day
0 2024-08-17  2024      8   17
1 2024-08-18  2024      8   18
```

- Adding and Subtracting Time
  - You can add or subtract time intervals to/from datetime objects.

```
import pandas as pd

# Example date
date = pd.to_datetime('2024-08-17')

# Adding 10 days
new_date_add = date + pd.Timedelta(days=10)
print(new_date_add)  # Output: 2024-08-27 00:00:00
```

```
# Subtracting 5 days
new_date_sub = date - pd.Timedelta(days=5)
print(new_date_sub)
```

```
2024-08-27 00:00:00
2024-08-12 00:00:00
```

- Calculating Date Differences
    - Compute the difference between two dates.

```
# Example dates
date1 = pd.to_datetime('2024-08-17')
date2 = pd.to_datetime('2024-08-27')

# Difference in days
difference = (date2 - date1).days
print(difference)
```

```
10
```

Input/Output Operations

- Pandas provides various methods for input and output (I/O) operations to read and write data in different formats. Here's a guide to handling some of the most common formats:

Reading Data

- Reading data refers to the process of loading data from a storage medium (such as files, databases, or other sources) into a program's memory, typically into a data structure like a DataFrame. In Pandas, this involves importing data from formats such as CSV, Excel, JSON, SQL, or Parquet files into a DataFrame, which is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure.

1. CSV Files
    - We have 'example.csv' file so we are going to load here. If file is in your local system you can put the full path of file

```
import pandas as pd

# Read CSV file
```

```
df = pd.read_csv('data.csv')

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

2. Excel Files

- Pandas can read and write excel files, keep in mind, this only imports data. Not formulas or images, having images or macros may cause this read_excel method to crash.

```
# Read Excel file
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')

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

3. JSON Files

- For a simple JSON file where each line is a JSON object (or a JSON array of objects), you can use pd.read_json().

```
# Read JSON file
df = pd.read_json('data.json')

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

4. SQL Databases

- When working with SQL databases in pandas, you can read data from a database file using SQL queries.

```
from sqlalchemy import create_engine

# Create a database engine
engine = create_engine('sqlite:///database.db')

# Read SQL table into DataFrame
df = pd.read_sql('table_name', con=engine)

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

Writing Data

- Writing data refers to the process of saving or exporting data from a program's memory (such as a DataFrame) to a storage medium (such as files or databases). In Pandas, this involves saving data to formats like CSV, Excel, JSON, SQL, or Parquet files.

1. CSV Files

```
# Write DataFrame to CSV
df.to_csv('data.csv', index=False)
```

2. Excel Files

```
# Write DataFrame to Excel
df.to_excel('data.xlsx', sheet_name='Sheet1', index=False)
```

3. JSON Files

```
# Write DataFrame to JSON
df.to_json('data.json', orient='records', lines=True)
```

4. SQL Databases

```
# Write DataFrame to SQL database
df.to_sql('table_name', con=engine, if_exists='replace', index=False)
```

Performance Optimization

- Performance optimization in Pandas is crucial for handling large datasets efficiently. Two key techniques for optimizing performance are vectorization and memory usage reduction. Here's an overview of each technique:

Vectorization

- Vectorization is the process of replacing explicit loops and iterative operations with vectorized operations that are performed on whole arrays or columns at once. This leverages

low-level optimizations in libraries like NumPy, which Pandas is built on, leading to significant performance improvements.

- Benefits:
  - Speed: Vectorized operations are typically much faster than Python loops because they use optimized C and Fortran code.
  - Readability: Code using vectorized operations is often cleaner and easier to understand.

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Vectorized operation: add columns
df['C'] = df['A'] + df['B']
```

Memory Usage Reduction

- Memory usage reduction involves optimizing the amount of memory used by DataFrames and Series to improve performance and efficiency. This can be particularly important when working with large datasets.
- Techniques:

1. Data Type Optimization:
   - Downcasting Numeric Types: Convert float64 to float32 or int64 to int32 where possible.

```
df['A'] = pd.to_numeric(df['A'], downcast='integer')
```

2. Memory-Efficient Reading
   - Specify Data Types on Read: When reading large files, specify data types to reduce memory usage.

```
df = pd.read_csv('data.csv', dtype={'col1': 'int32', 'col2': 'float32'})
```

3. Efficient Use of DataFrame Operations
   - Avoid Creating Unnecessary Copies: Use in-place operations where possible.

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

- Filter Data Early: Filter large datasets as early as possible to reduce the size of DataFrames.

```
df_filtered = df[df['A'] > 10]
```

Visualization with Pandas

- Pandas offers powerful and flexible tools for data visualization directly through its built-in plotting functions, which are built on top of Matplotlib. Here's an overview of how to create various types of visualizations using Pandas

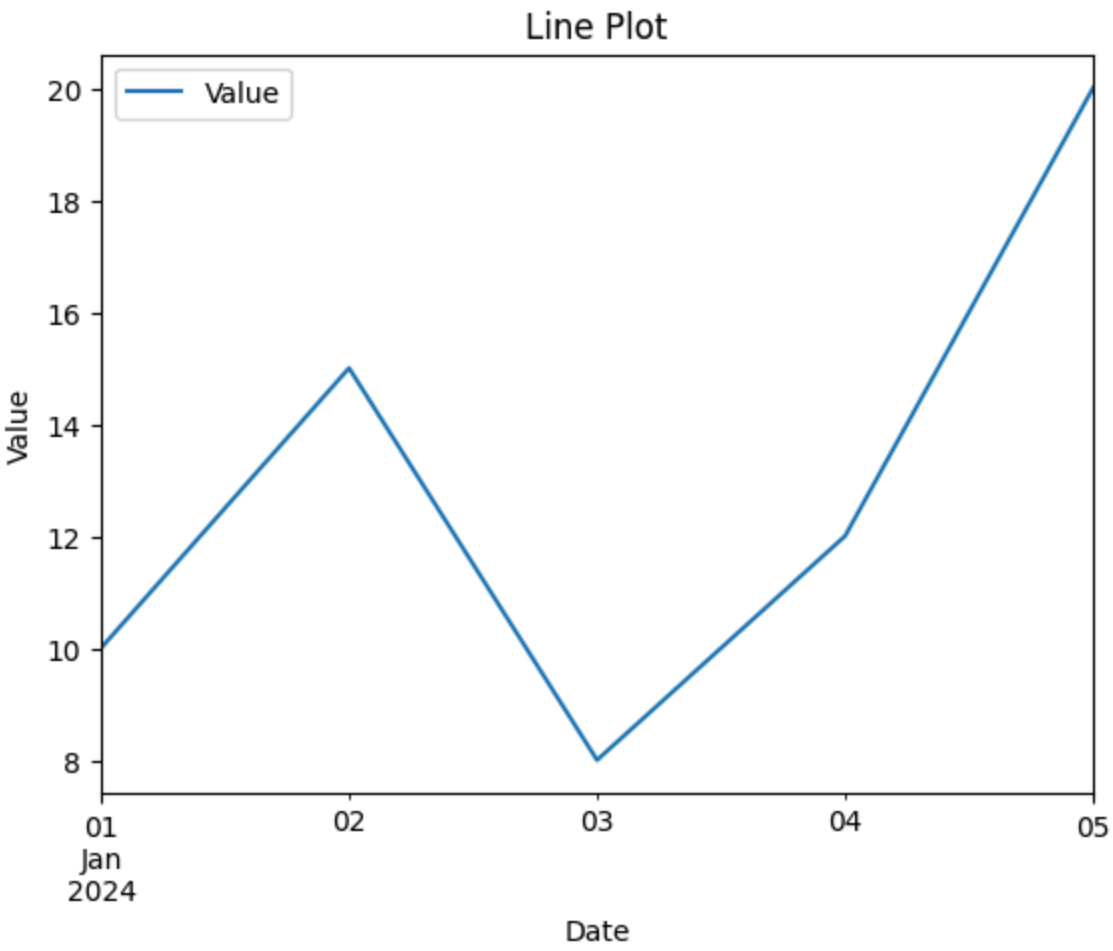
Basic Plotting

1. Line Plot
    - A line plot displays data points connected by straight lines, useful for showing trends over time.

```
import pandas as pd
import matplotlib.pyplot as plt

# Create a DataFrame
df = pd.DataFrame({
    'Date': pd.date_range(start='2024-01-01', periods=5),
    'Value': [10, 15, 8, 12, 20]
})

# Plot a line graph
df.plot(x='Date', y='Value', kind='line')
plt.title('Line Plot')
plt.xlabel('Date')
plt.ylabel('Value')
plt.show()
```

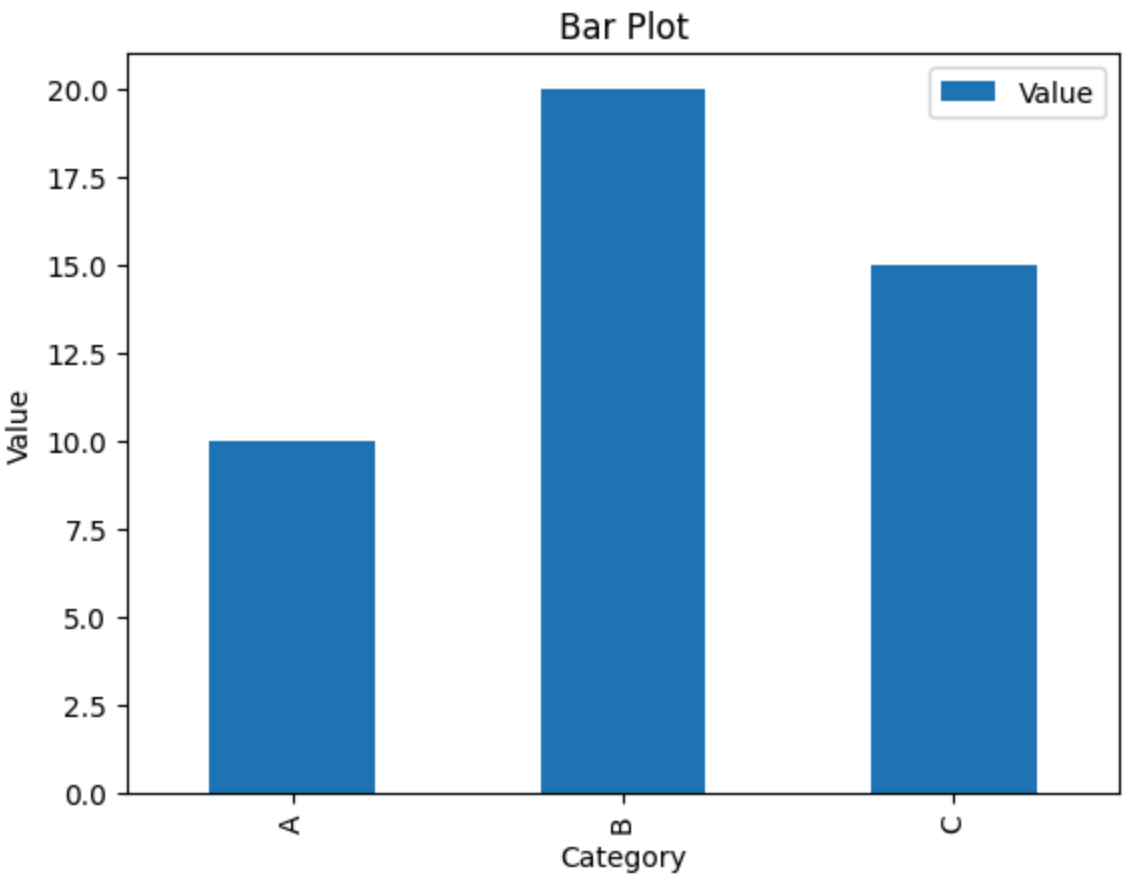


2. Bar Plot

   - A bar plot displays rectangular bars with lengths proportional to the values they represent, suitable for comparing different categories.

```
df = pd.DataFrame({
    'Category': ['A', 'B', 'C'],
    'Value': [10, 20, 15]
})

# Plot a bar graph
df.plot(x='Category', y='Value', kind='bar')
plt.title('Bar Plot')
plt.xlabel('Category')
plt.ylabel('Value')
plt.show()
```

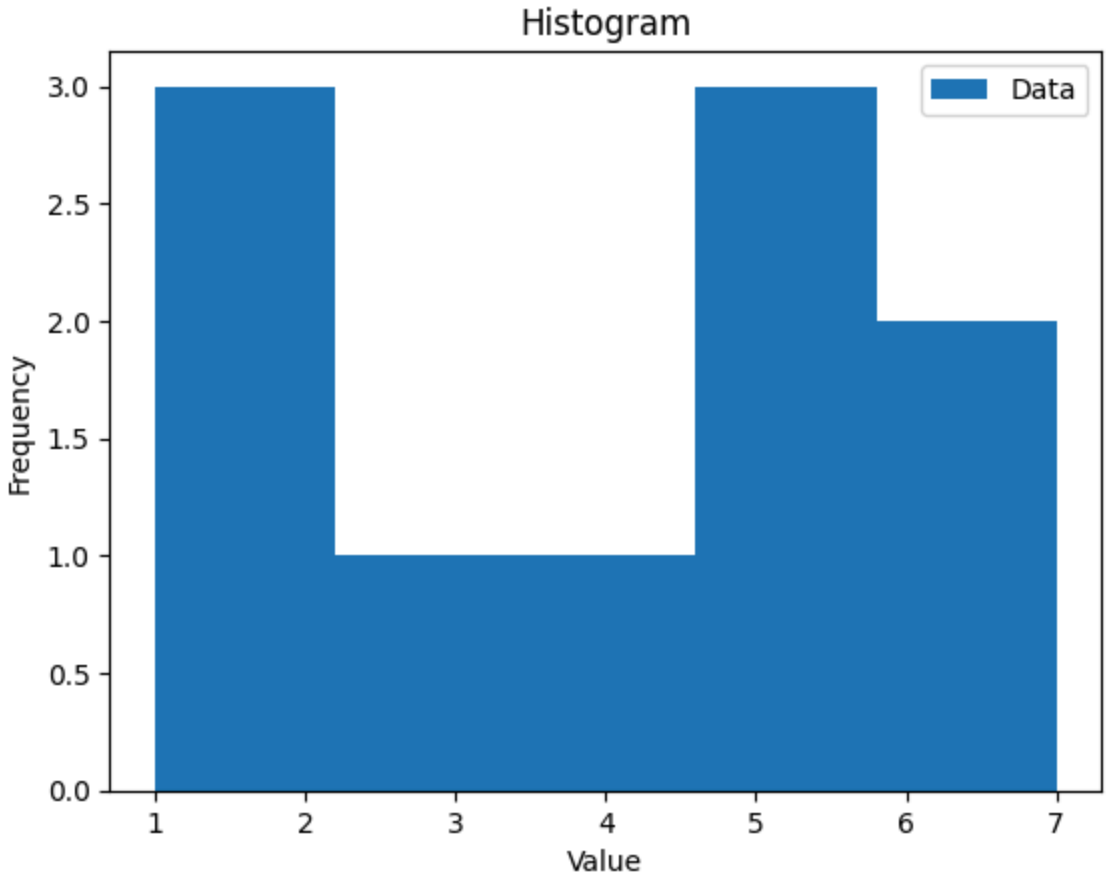


3. Histogram

   - A histogram displays the distribution of numerical data by dividing it into bins and counting the number of data points in each bin.

```
df = pd.DataFrame({
    'Data': [1, 2, 2, 3, 4, 5, 5, 5, 6, 7]
})

# Plot a histogram
df.plot(kind='hist', bins=5)
plt.title('Histogram')
plt.xlabel('Value')
plt.show()
```

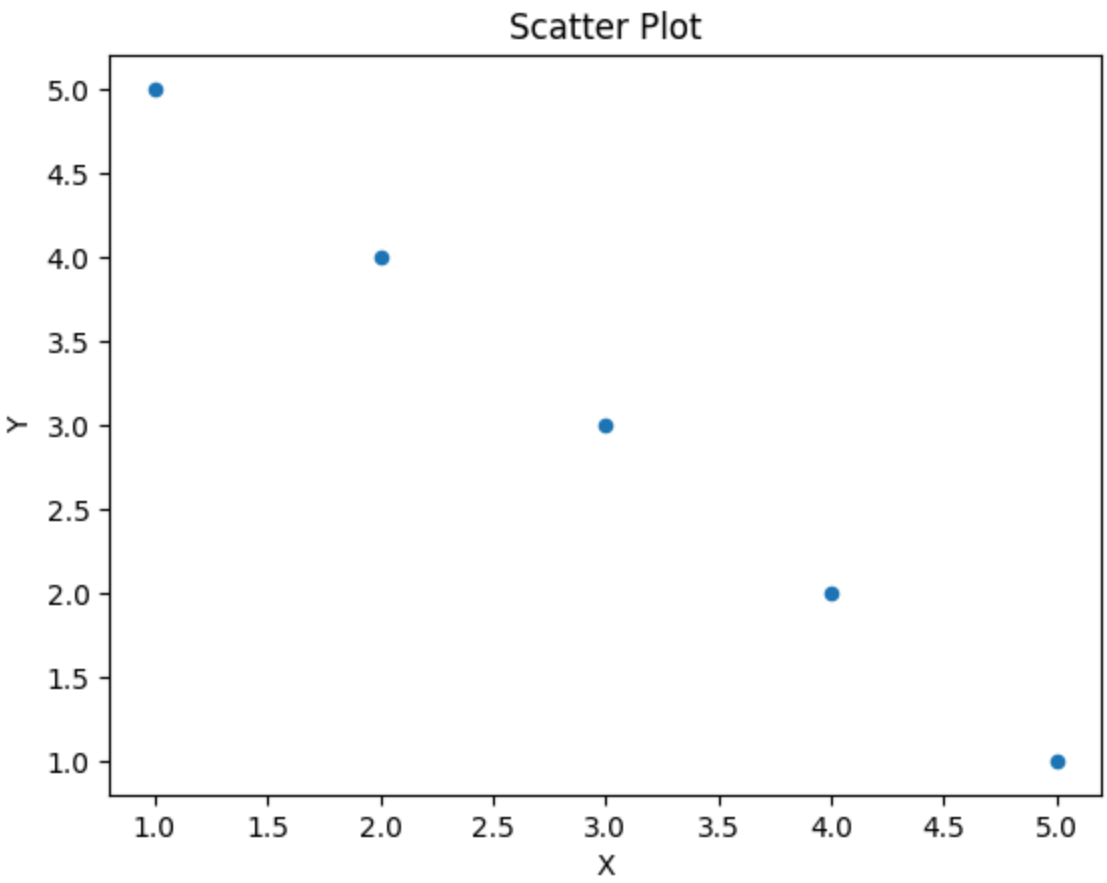


4. Scatter Plot

   - A scatter plot shows individual data points on a Cartesian plane, useful for observing relationships between two variables.

```
df = pd.DataFrame({
    'X': [1, 2, 3, 4, 5],
    'Y': [5, 4, 3, 2, 1]
})

# Plot a scatter plot
df.plot(kind='scatter', x='X', y='Y')
plt.title('Scatter Plot')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



Advanced Visualization

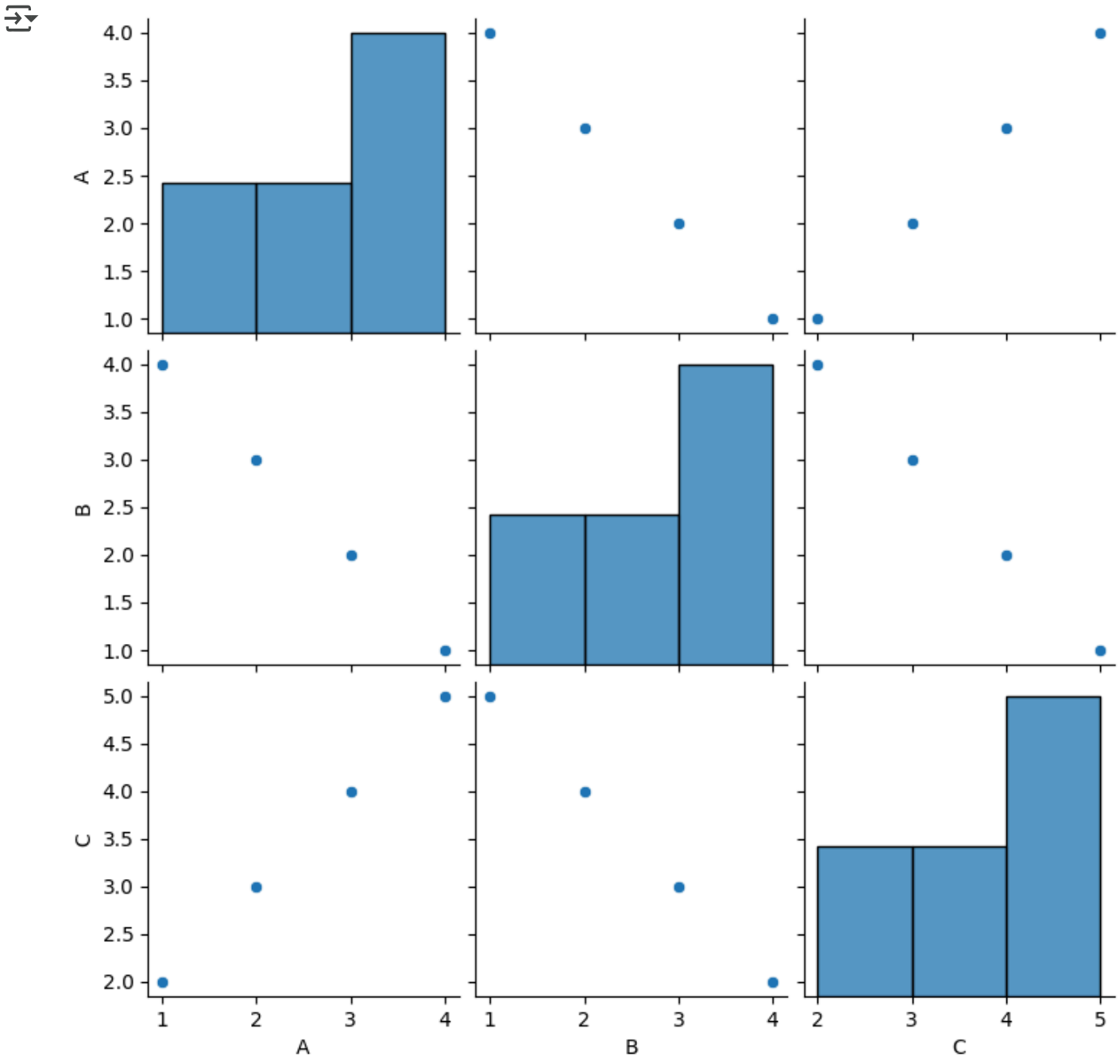
- For advanced data visualization, you can use additional libraries that build on Matplotlib or provide their own high-level functionalities. Here's a look at some popular libraries and techniques for creating advanced visualizations

1. Seaborn
    - Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.
    - Features:
        - Complex Plots: Easily create complex visualizations like pair plots, violin plots, and heatmaps.
        - Aesthetics: Built-in themes and color palettes for more attractive plots.

```
# Pair Plot: To visualize pairwise relationships in a dataset.
import seaborn as sns
import pandas as pd
```

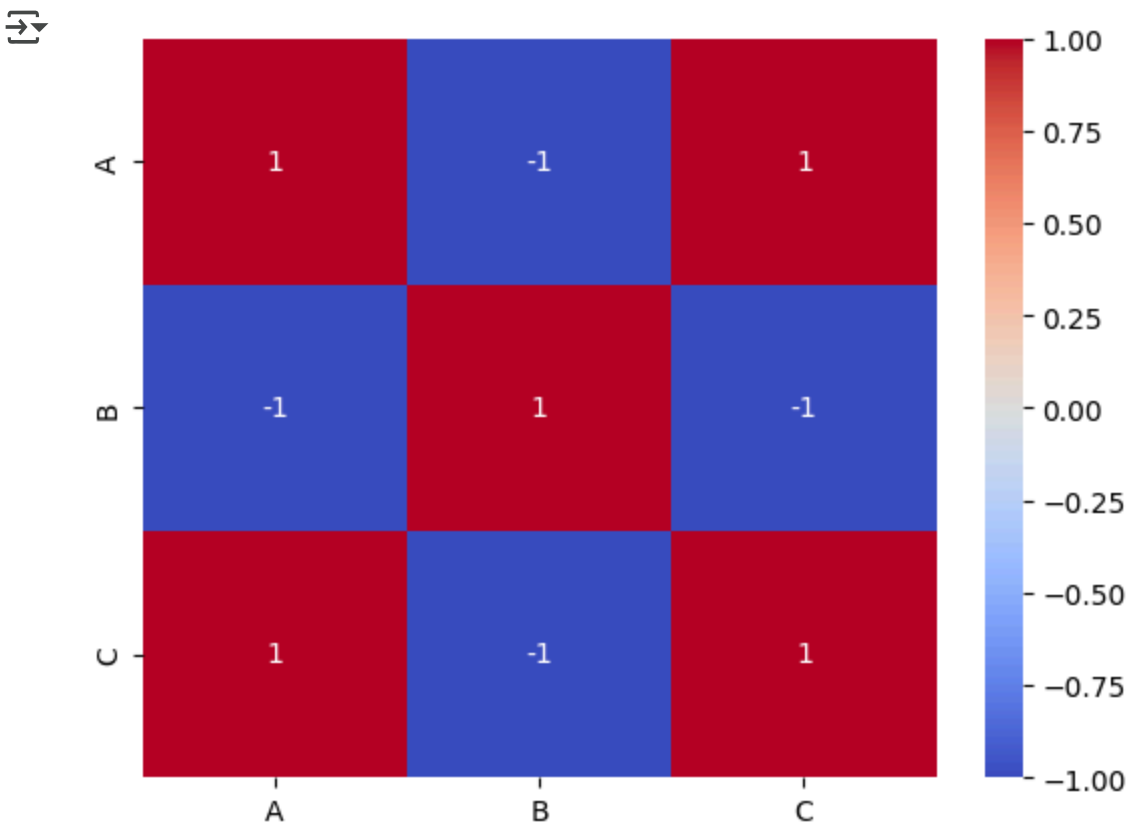
```
df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [4, 3, 2, 1],
    'C': [2, 3, 4, 5]
})

sns.pairplot(df)
plt.show()
```

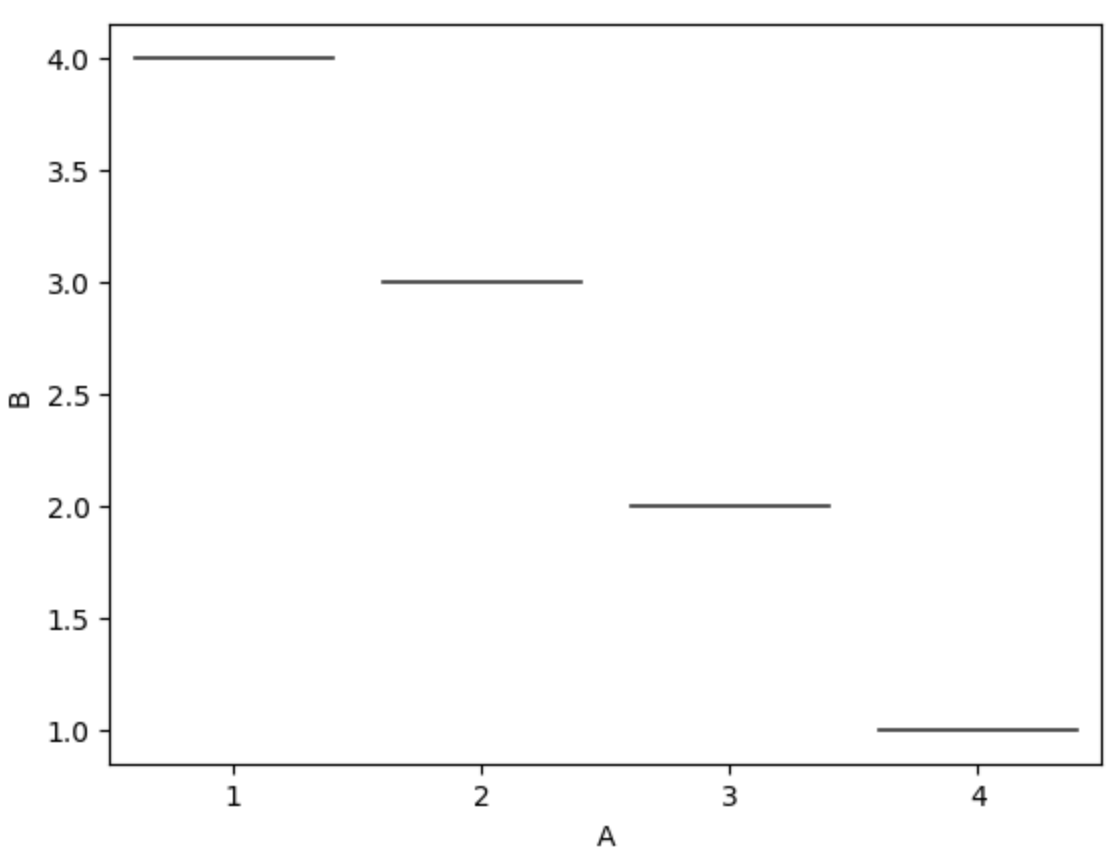



```
# Heatmap: To visualize matrix data.
corr = df.corr()
```

```
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.show()
```




```
# Violin Plot: To visualize the distribution of the data.
sns.violinplot(x='A', y='B', data=df)
plt.show()
```



2. Plotly
    - Plotly is an interactive graphing library that provides a range of complex plots and is highly customizable. It supports both offline and online modes.
    - Features:
        - Interactivity: Interactive features such as zooming, panning, and tooltips.
        - Wide Range of Plots: Includes scatter plots, 3D plots, heatmaps, and more.

```
# Interactive Scatter Plot:
import plotly.express as px
import pandas as pd

df = pd.DataFrame({
    'X': [1, 2, 3, 4],
    'Y': [4, 3, 2, 1]
})

fig = px.scatter(df, x='X', y='Y', title='Interactive Scatter Plot')
fig.show()
```

Interactive Scatter Plot