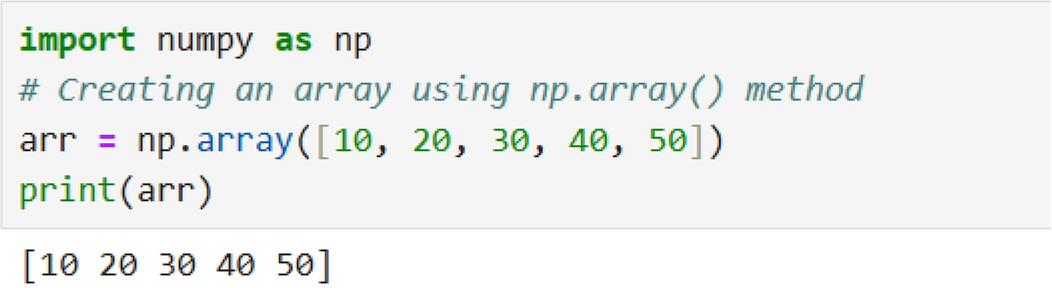
# Python Numpy

## 1. Introduction to NumPy

NumPy (Numerical Python) is a fundamental package for numerical computing in Python. It provides an efficient array object (ndarray), as well as functions to manipulate these arrays. NumPy is heavily used in data analysis, machine learning, scientific computing, and data manipulation tasks.

**Eg:**



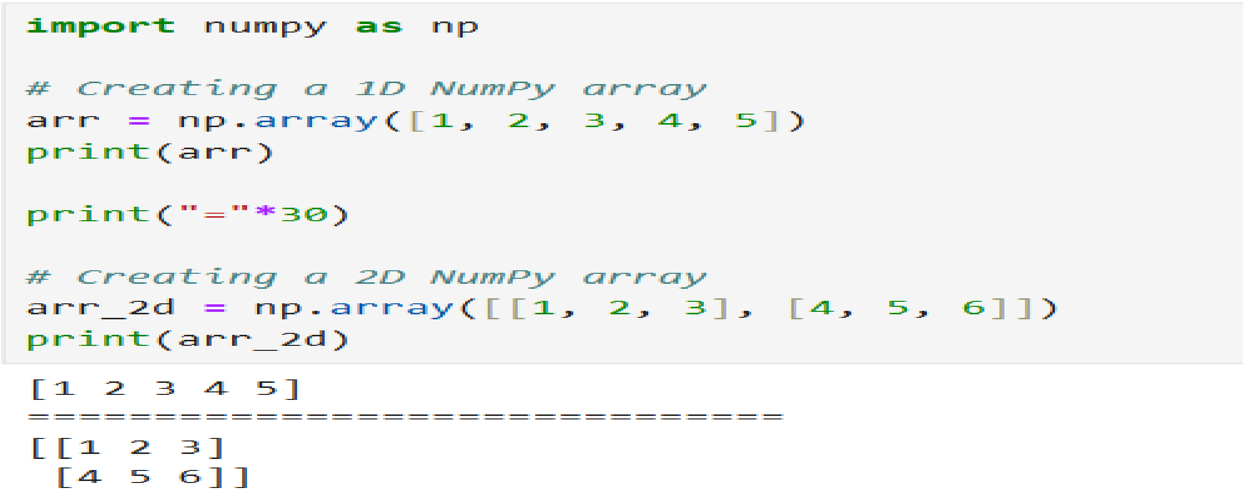
```python
import numpy as np
# Creating an array using np.array() method
arr = np.array([10, 20, 30, 40, 50])
print(arr)
```

```
[10 20 30 40 50]
```

## 2. Basic Concepts

### NumPy Arrays

The core data structure in NumPy is the ndarray, which is a multi-dimensional array. It can handle large datasets and is more efficient than Python lists.

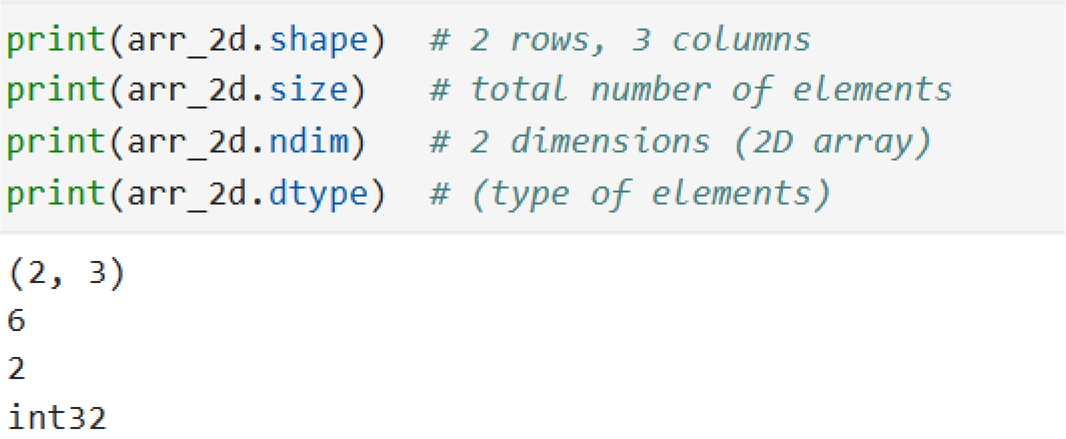


```python
import numpy as np

# Creating a 1D NumPy array
arr = np.array([1, 2, 3, 4, 5])
print(arr)

print("="*30)

# Creating a 2D NumPy array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d)
```

```
[1 2 3 4 5]
==============================
[[1 2 3]
 [4 5 6]]
```

### Array Attributes

NumPy arrays have useful attributes like shape, size, ndim, etc.



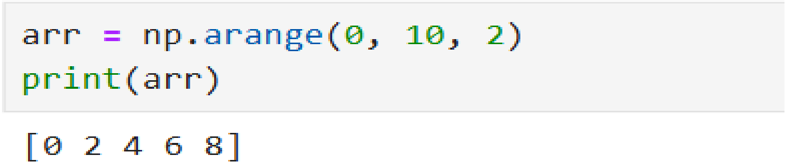
```
print(arr_2d.shape)  # 2 rows, 3 columns
print(arr_2d.size)   # total number of elements
print(arr_2d.ndim)   # 2 dimensions (2D array)
print(arr_2d.dtype)  # (type of elements)
```

```
(2, 3)
6
2
int32
```

### 3. Array Creation Functions

NumPy provides a range of functions for creating arrays.

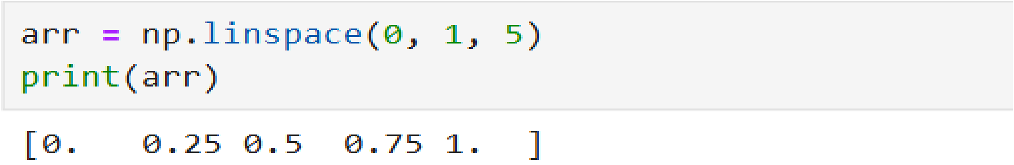
**arange():** Returns an array with evenly spaced values within a given range.



```
arr = np.arange(0, 10, 2)
print(arr)
```

```
[0 2 4 6 8]
```

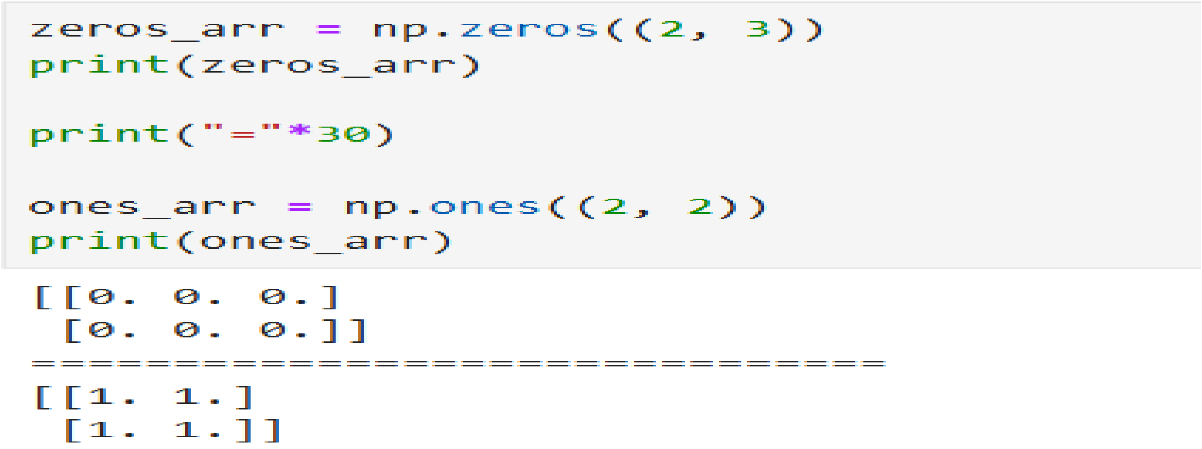
**linspace()**: Returns an array of evenly spaced numbers over a specified range.



```
arr = np.linspace(0, 1, 5)
print(arr)
```

```
[0.   0.25 0.5  0.75 1.  ]
```

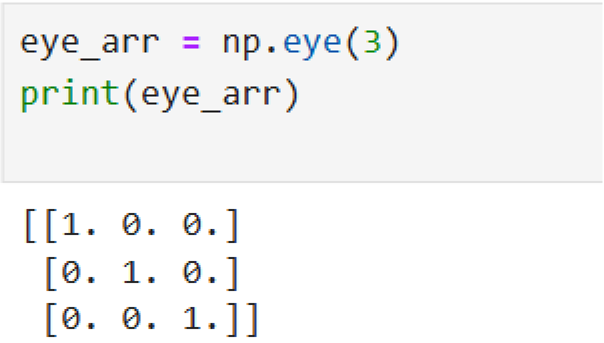
**zeros() and ones()**: Create arrays filled with zeros or ones.



```
zeros_arr = np.zeros((2, 3))
print(zeros_arr)

print("="*30)

ones_arr = np.ones((2, 2))
print(ones_arr)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
==============================
[[1. 1.]
 [1. 1.]]
```

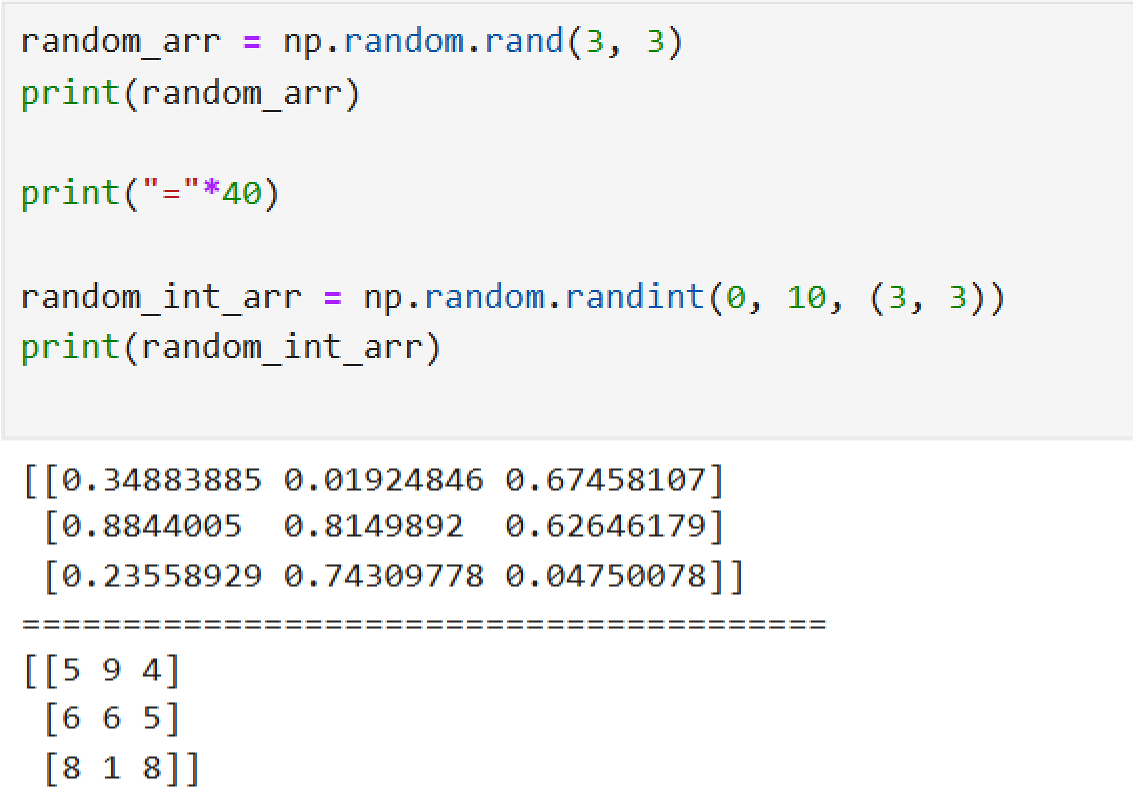
**eye():** Create an identity matrix.



```
eye_arr = np.eye(3)
print(eye_arr)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

**random functions**: Generate arrays with random numbers.



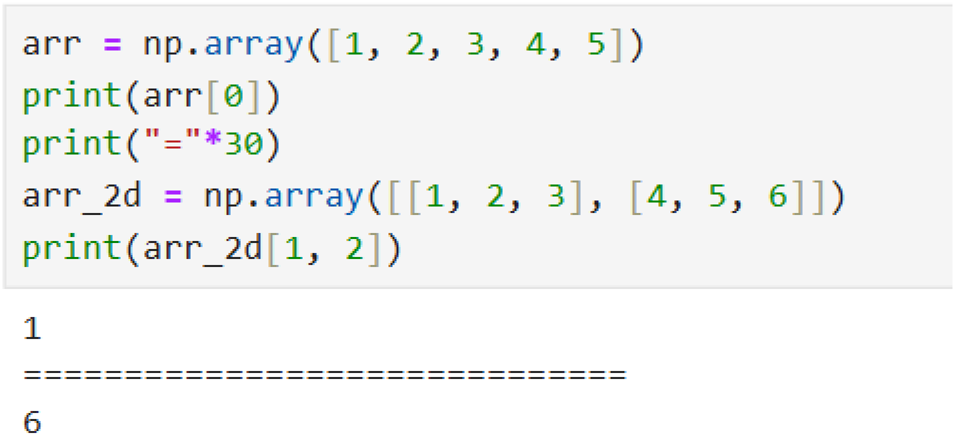
```
random_arr = np.random.rand(3, 3)
print(random_arr)

print("="*40)

random_int_arr = np.random.randint(0, 10, (3, 3))
print(random_int_arr)
```

```
[[0.34883885 0.01924846 0.67458107]
 [0.8844005  0.8149892  0.62646179]
 [0.23558929 0.74309778 0.04750078]]
========================================
[[5 9 4]
 [6 6 5]
 [8 1 8]]
```

## 4. Array Indexing and Slicing

### Basic Indexing

Indexing in NumPy is similar to Python lists, but it supports multi-dimensional arrays.

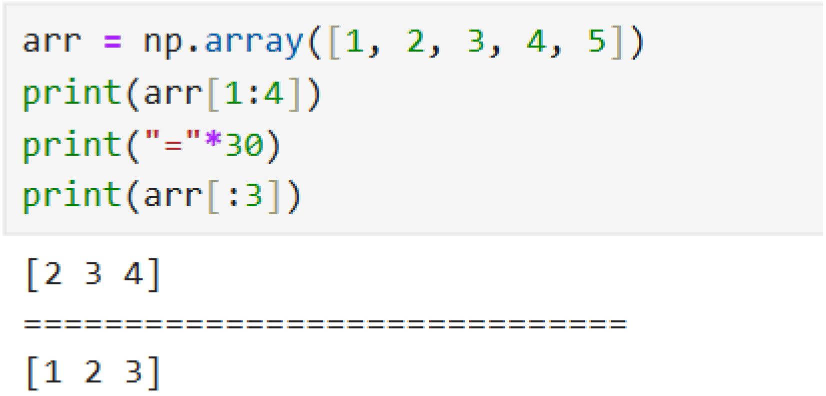


```
arr = np.array([1, 2, 3, 4, 5])
print(arr[0])
print("="*30)
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d[1, 2])
```

```
1
==============================
6
```

### Slicing Arrays

You can slice arrays to get a range of values.

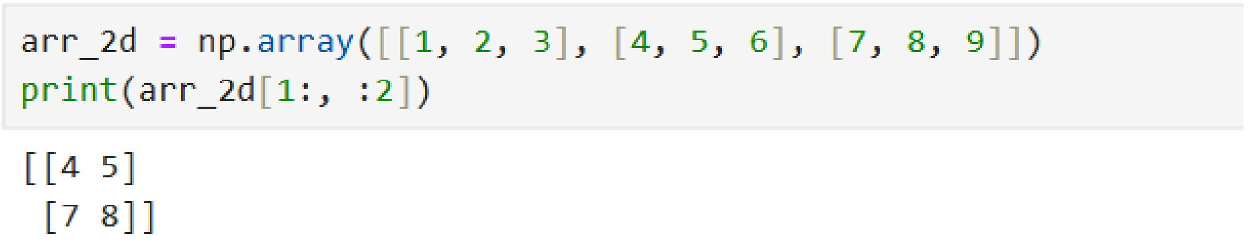


```
arr = np.array([1, 2, 3, 4, 5])
print(arr[1:4])
print("="*30)
print(arr[:3])
```

```
[2 3 4]
==============================
[1 2 3]
```

### Slicing Multi-dimensional Arrays

You can slice multidimensional arrays similarly.



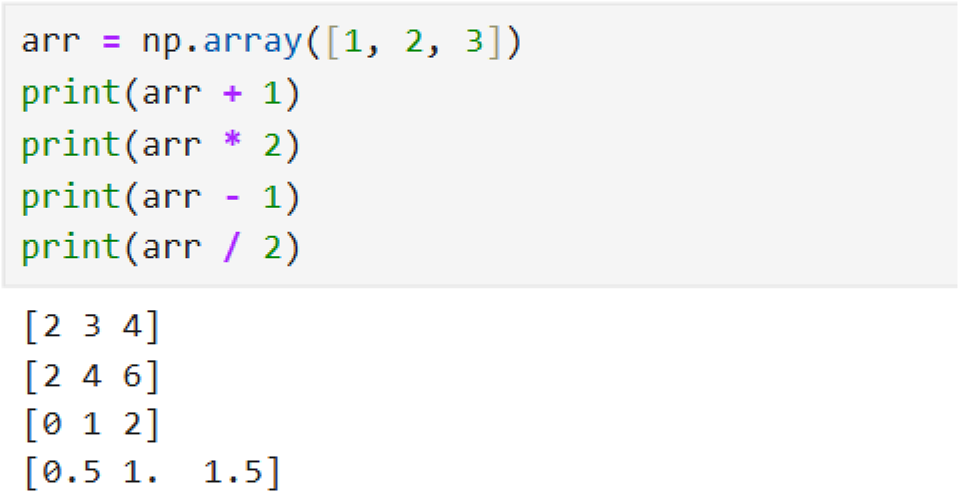
```
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr_2d[1:, :2])
```

```
[[4 5]
 [7 8]]
```

## 5. Array Operations

NumPy allows element-wise mathematical operations.

### Element-wise Arithmetic Operations

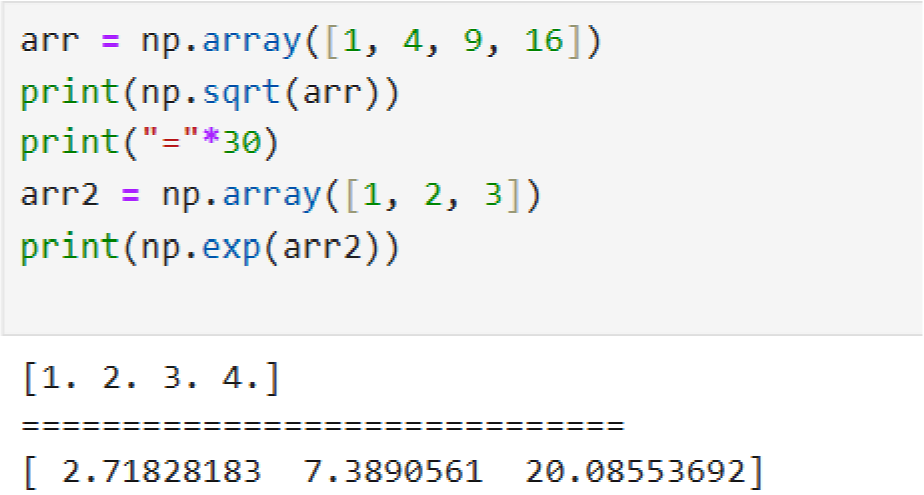


```
arr = np.array([1, 2, 3])
print(arr + 1)
print(arr * 2)
print(arr - 1)
print(arr / 2)
```

```
[2 3 4]
[2 4 6]
[0 1 2]
[0.5 1.  1.5]
```

### Universal Functions (ufuncs)

NumPy provides many universal functions for common operations like square root, logarithm, etc.

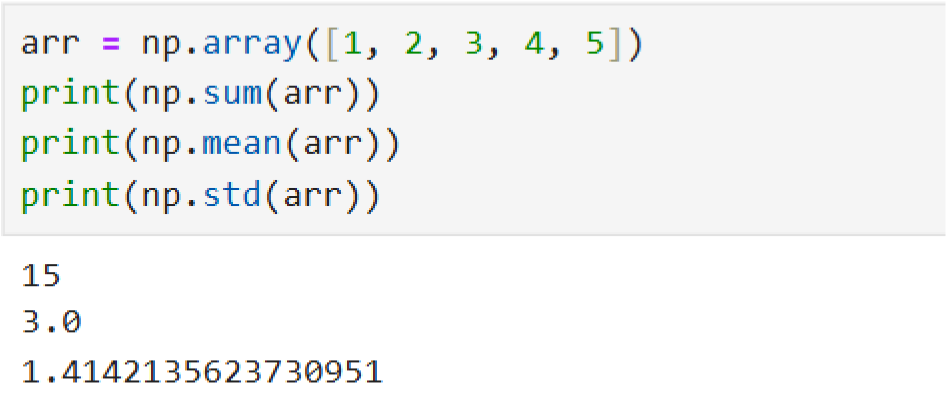


```
arr = np.array([1, 4, 9, 16])
print(np.sqrt(arr))
print("="*30)
arr2 = np.array([1, 2, 3])
print(np.exp(arr2))
```

```
[1. 2. 3. 4.]
==============================
[ 2.71828183  7.3890561  20.08553692]
```

### Aggregation Functions

NumPy provides several functions to compute aggregated values like sum, mean, standard deviation, etc.

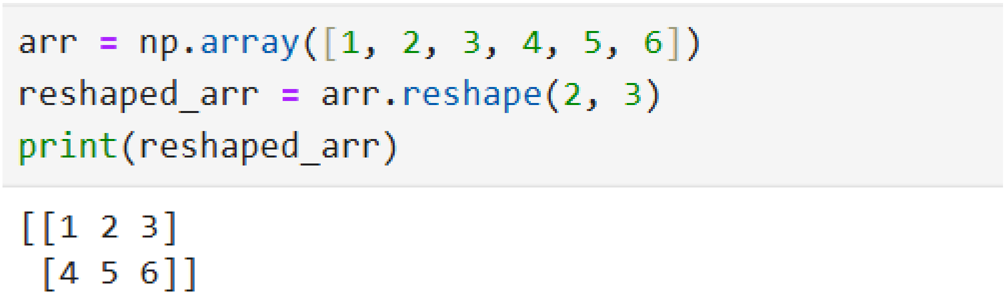


```
arr = np.array([1, 2, 3, 4, 5])
print(np.sum(arr))
print(np.mean(arr))
print(np.std(arr))
```

```
15
3.0
1.4142135623730951
```

## 6. Array Reshaping

You can reshape arrays into different dimensions.

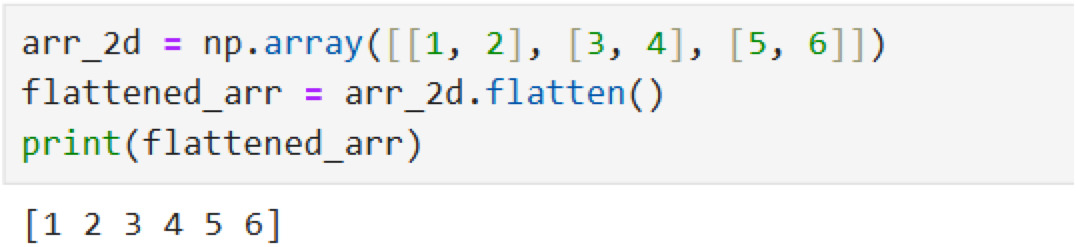


```
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(2, 3)
print(reshaped_arr)
```

```
[[1 2 3]
 [4 5 6]]
```

### Flattening

You can flatten a multi-dimensional array into a 1D array.



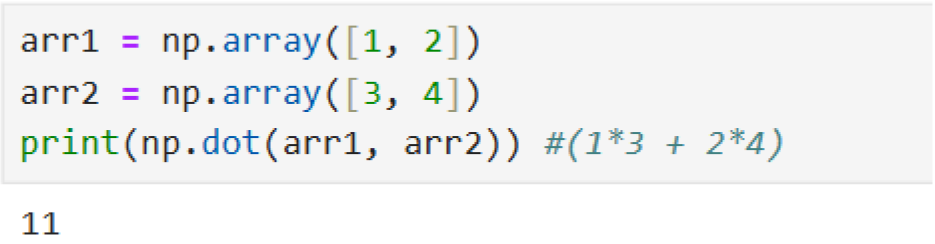
```
arr_2d = np.array([[1, 2], [3, 4], [5, 6]])
flattened_arr = arr_2d.flatten()
print(flattened_arr)
```

```
[1 2 3 4 5 6]
```

## 7. Mathematical and Linear Algebra Functions

### Dot Product

You can calculate the dot product of two arrays.

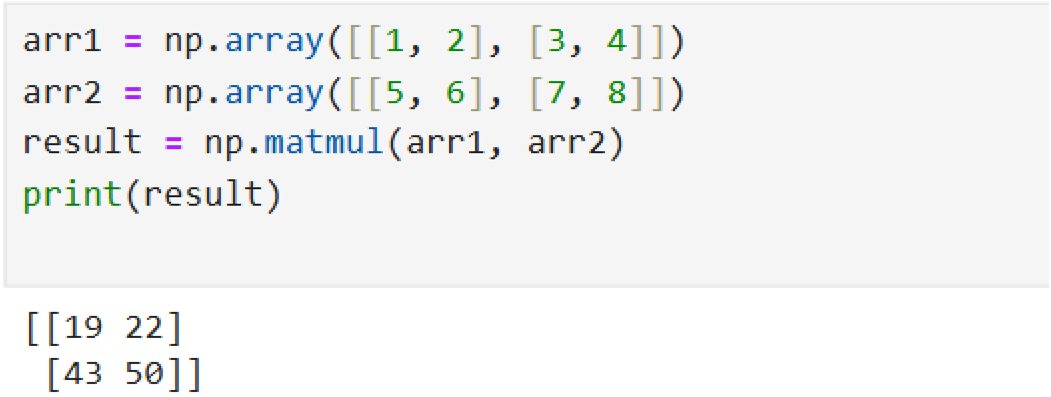


```
arr1 = np.array([1, 2])
arr2 = np.array([3, 4])
print(np.dot(arr1, arr2)) #(1*3 + 2*4)
```

```
11
```

### Matrix Multiplication

Matrix multiplication using np.matmul() or @ operator.

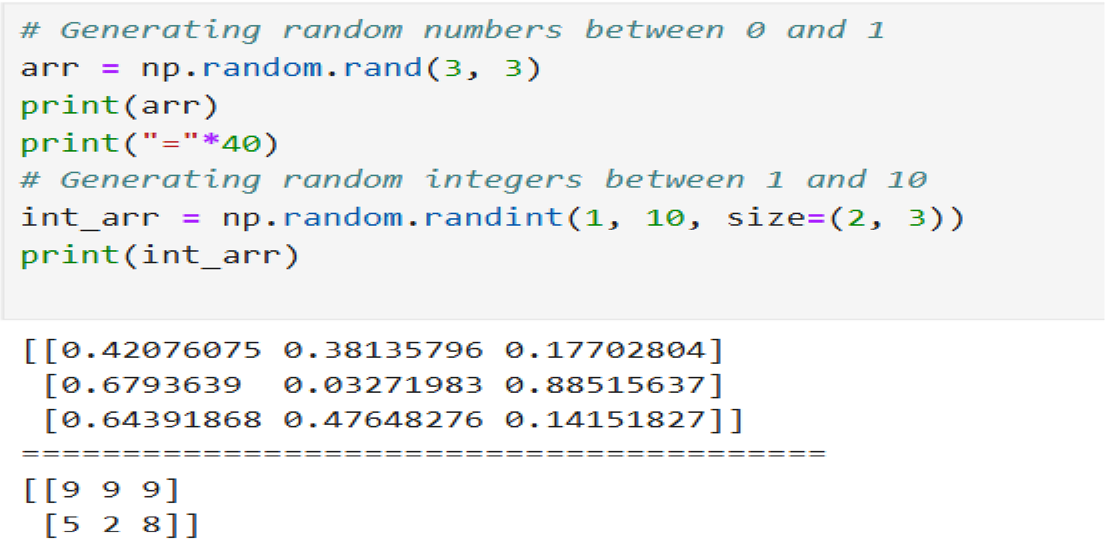


```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
result = np.matmul(arr1, arr2)
print(result)
```

```
[[19 22]
 [43 50]]
```

## 8. Random Sampling

NumPy provides random number generation functions.



```
# Generating random numbers between 0 and 1
arr = np.random.rand(3, 3)
print(arr)
print("="*40)
# Generating random integers between 1 and 10
int_arr = np.random.randint(1, 10, size=(2, 3))
print(int_arr)
```

```
[[0.42076075 0.38135796 0.17702804]
 [0.6793639  0.03271983 0.88515637]
 [0.64391868 0.47648276 0.14151827]]
========================================
[[9 9 9]
 [5 2 8]]
```

# Python Pandas Library

## 1. Introduction to Pandas

Pandas is an open-source library that provides high-performance data structures and data analysis tools. The main data structures in Pandas are:

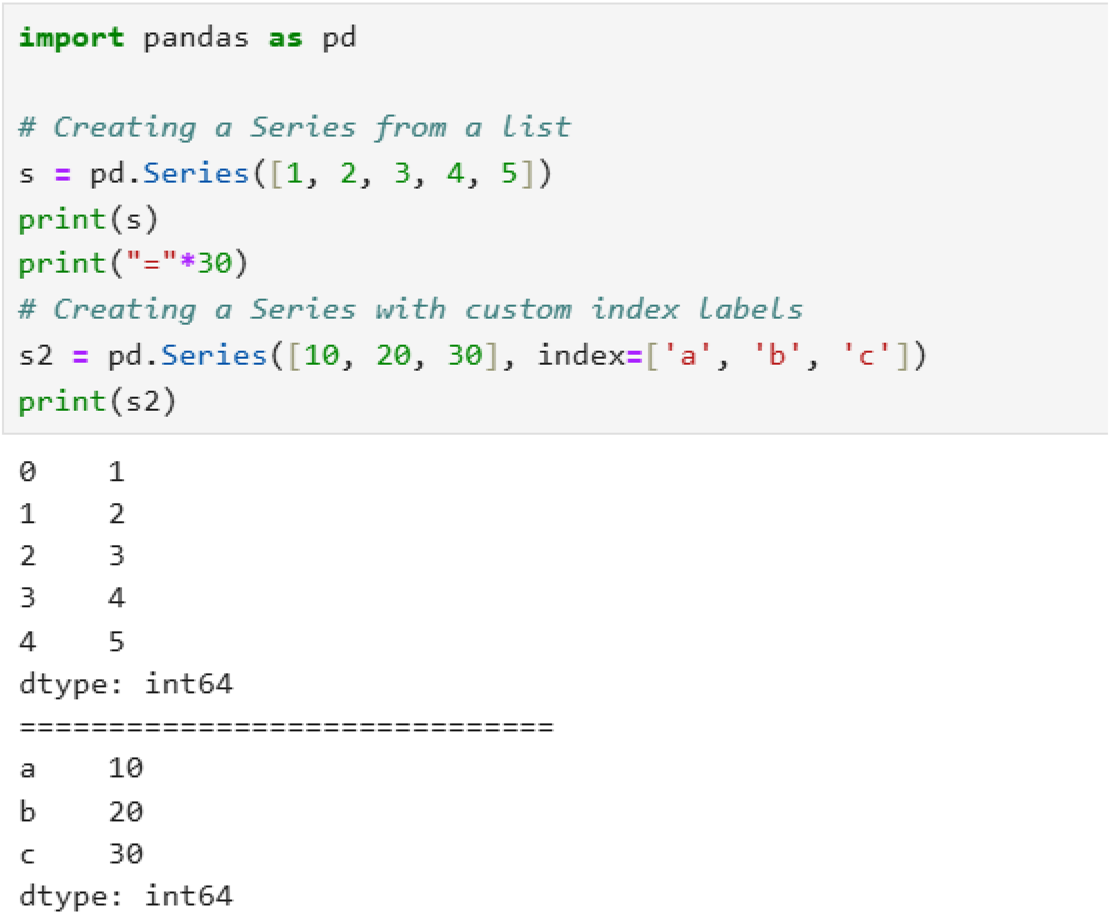
- Series: A one-dimensional labeled array, similar to a list or column in a table.
- DataFrame: A two-dimensional labeled data structure, similar to a table or spreadsheet.

Pandas is built on top of NumPy and provides powerful methods for data manipulation, cleaning, and analysis.

### 2. Pandas Data Structures

#### Series

A Series is a one-dimensional array-like object that holds data and an associated array of data labels (called indices).

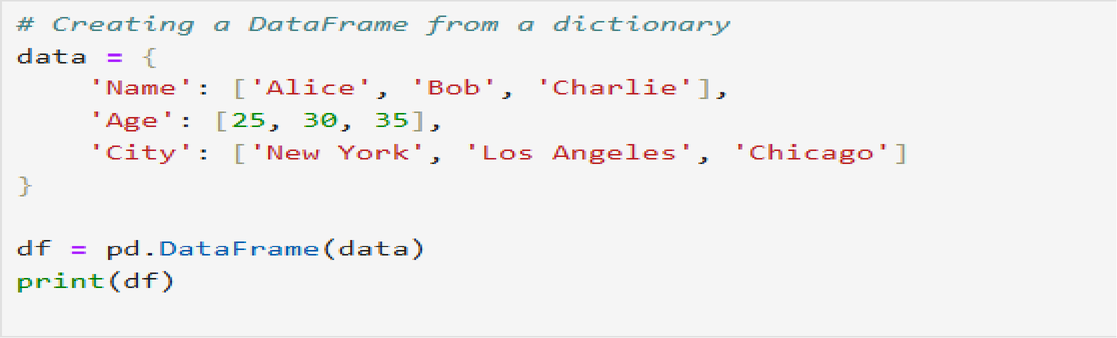


```python
import pandas as pd

# Creating a Series from a list
s = pd.Series([1, 2, 3, 4, 5])
print(s)
print("="*30)
# Creating a Series with custom index labels
s2 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
print(s2)
```

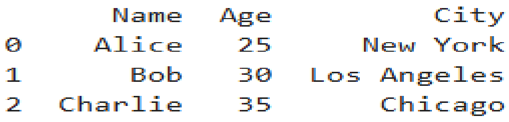
```
0    1
1    2
2    3
3    4
4    5
dtype: int64
==============================
a    10
b    20
c    30
dtype: int64
```

### DataFrame

A DataFrame is a two-dimensional table with labeled axes (rows and columns). You can think of it like an Excel sheet or SQL table.



```
# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)
print(df)
```

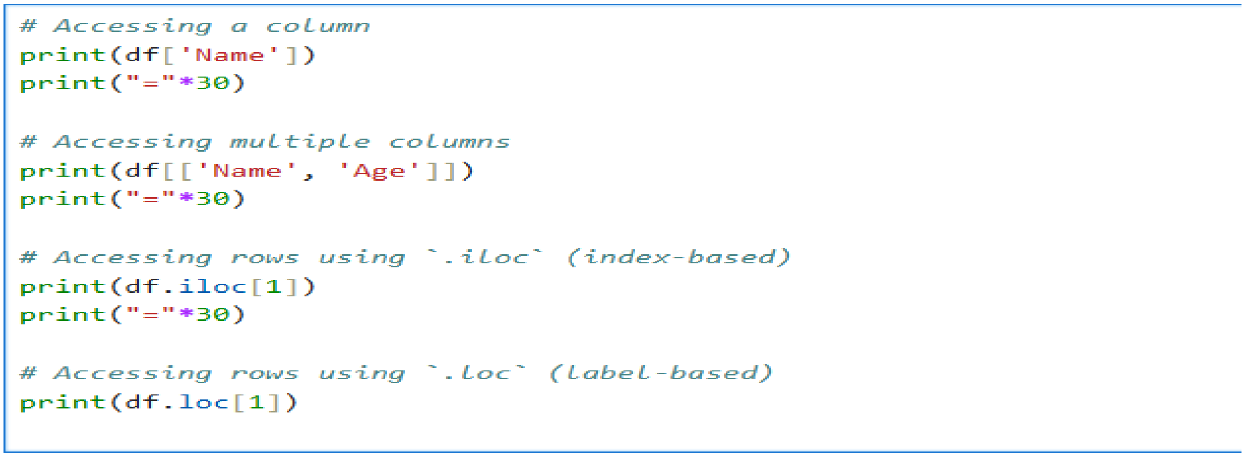


```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

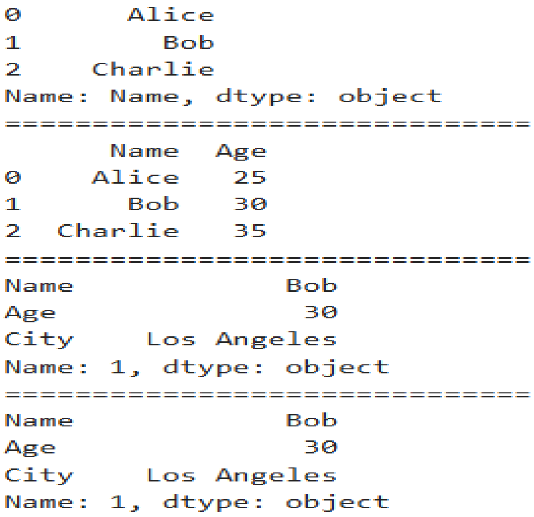
## 3. DataFrame Operations

Accessing Data in DataFrame

You can access individual columns or rows from a DataFrame using indexing or slicing.



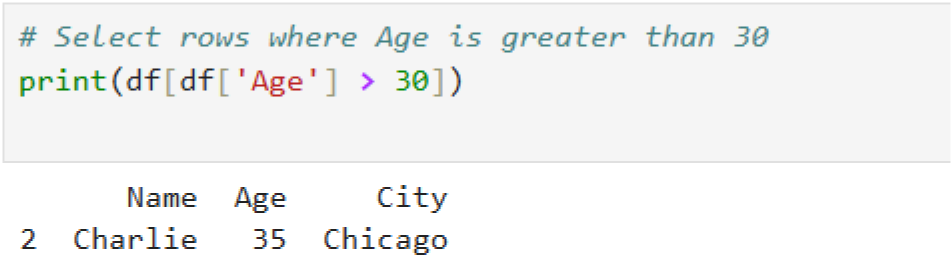
```
# Accessing a column
print(df['Name'])
print("="*30)

# Accessing multiple columns
print(df[['Name', 'Age']])
print("="*30)

# Accessing rows using `.iloc` (index-based)
print(df.iloc[1])
print("="*30)

# Accessing rows using `.loc` (label-based)
print(df.loc[1])
```



```
0      Alice
1        Bob
2    Charlie
Name: Name, dtype: object
==============================
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
==============================
Name            Bob
Age              30
City    Los Angeles
Name: 1, dtype: object
==============================
Name            Bob
Age              30
City    Los Angeles
Name: 1, dtype: object
```

### Selecting Rows Based on Conditions

You can filter rows based on conditions.

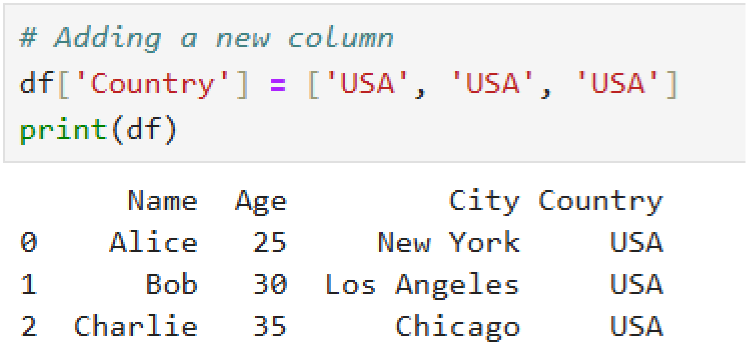


```
# Select rows where Age is greater than 30
print(df[df['Age'] > 30])
```

```
      Name  Age     City
2  Charlie   35  Chicago
```

### Adding Columns

You can add new columns to the DataFrame.

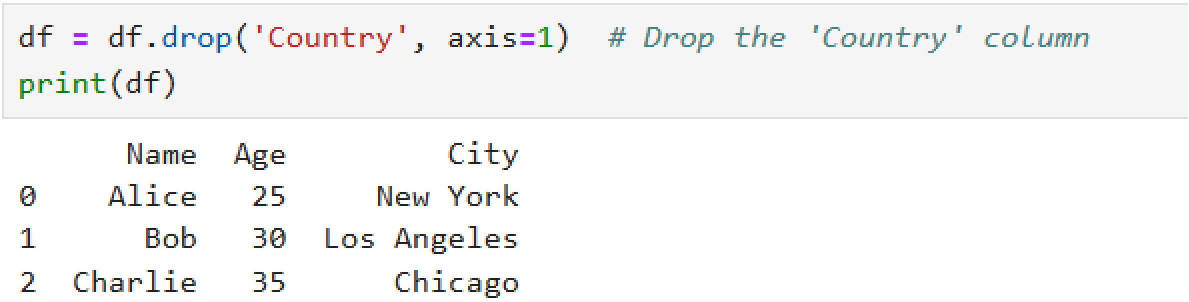


```
# Adding a new column
df['Country'] = ['USA', 'USA', 'USA']
print(df)
```

```
      Name  Age         City Country
0    Alice   25     New York     USA
1      Bob   30  Los Angeles     USA
2  Charlie   35      Chicago     USA
```

### Dropping Columns

You can remove columns using drop().



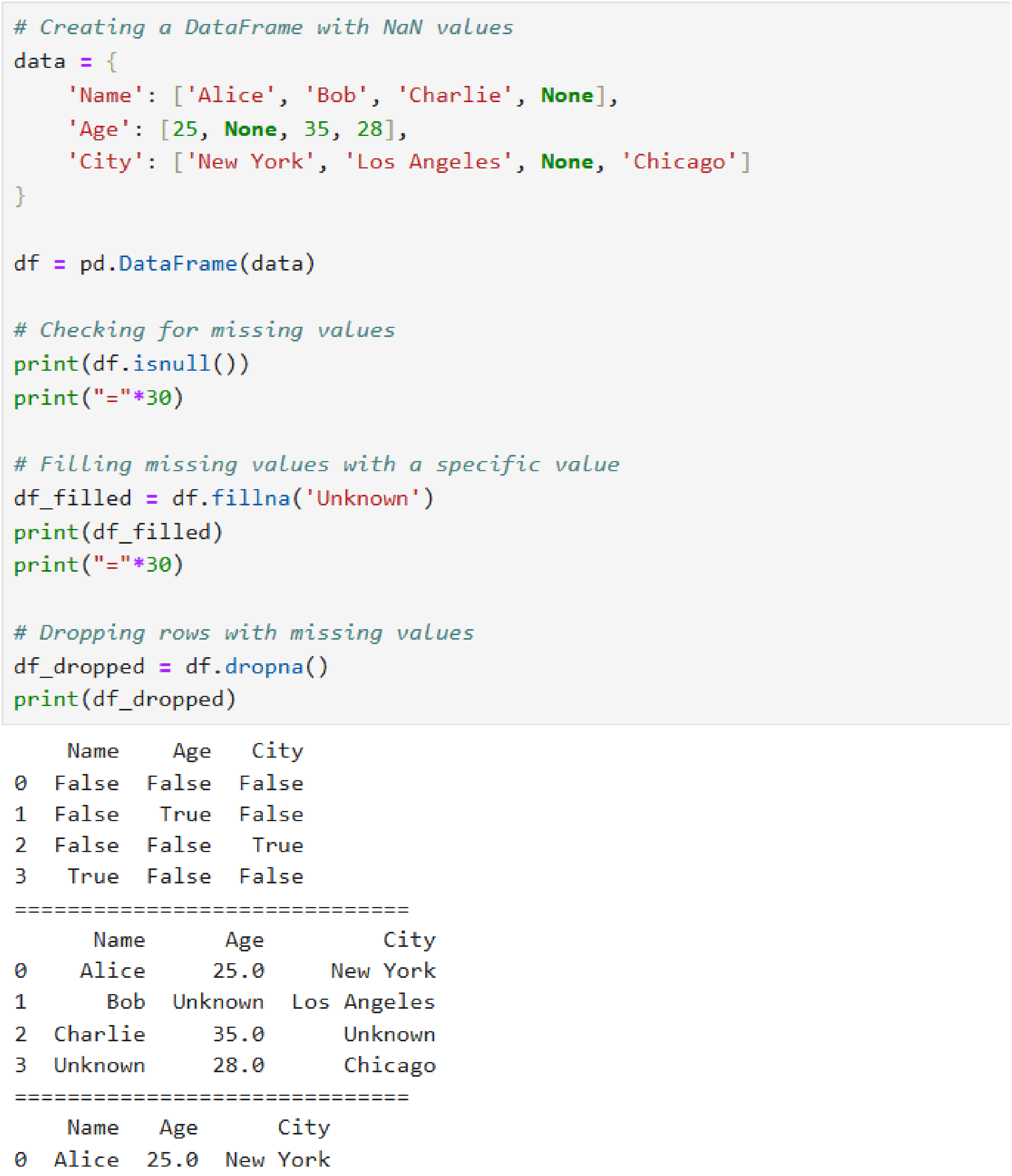
```
df = df.drop('Country', axis=1)  # Drop the 'Country' column
print(df)
```

```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

### 4. Data Cleaning with Pandas

Handling Missing Data

You often deal with datasets that have missing or NaN values. Pandas provides several functions to handle these cases.

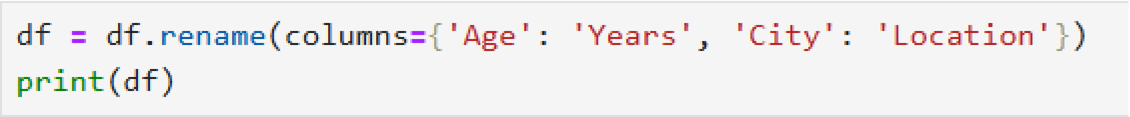


```python
# Creating a DataFrame with NaN values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', None],
    'Age': [25, None, 35, 28],
    'City': ['New York', 'Los Angeles', None, 'Chicago']
}

df = pd.DataFrame(data)

# Checking for missing values
print(df.isnull())
print("="*30)

# Filling missing values with a specific value
df_filled = df.fillna('Unknown')
print(df_filled)
print("="*30)

# Dropping rows with missing values
df_dropped = df.dropna()
print(df_dropped)
```

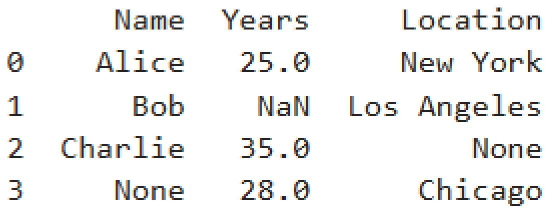
```
    Name    Age   City
0  False  False  False
1  False   True  False
2  False  False   True
3   True  False  False
==============================
      Name      Age         City
0    Alice     25.0     New York
1      Bob  Unknown  Los Angeles
2  Charlie     35.0      Unknown
3  Unknown     28.0      Chicago
==============================
    Name   Age      City
0  Alice  25.0  New York
```

### Renaming Columns

You can rename columns easily using the rename() method.



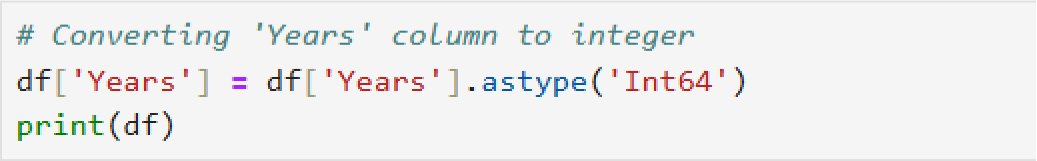
```
df = df.rename(columns={'Age': 'Years', 'City': 'Location'})
print(df)
```



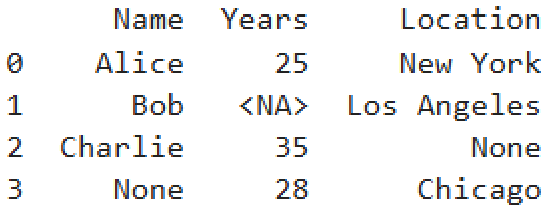
```
      Name  Years     Location
0    Alice   25.0     New York
1      Bob    NaN  Los Angeles
2  Charlie   35.0         None
3     None   28.0      Chicago
```

### Changing Data Types

You can convert the data type of columns using astype().



```
# Converting 'Years' column to integer
df['Years'] = df['Years'].astype('Int64')
print(df)
```



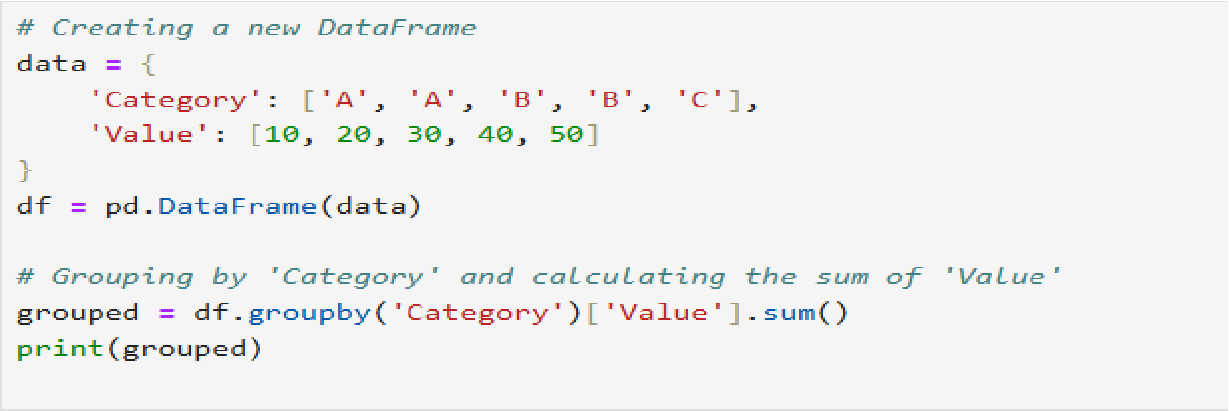
```
      Name  Years     Location
0    Alice     25     New York
1      Bob   <NA>  Los Angeles
2  Charlie     35         None
3     None     28      Chicago
```

### 5. Data Aggregation

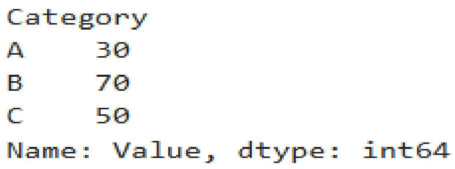
Pandas provides powerful methods for grouping and summarizing data.

### GroupBy

You can group data by a particular column and then apply aggregation functions like sum, mean, etc.



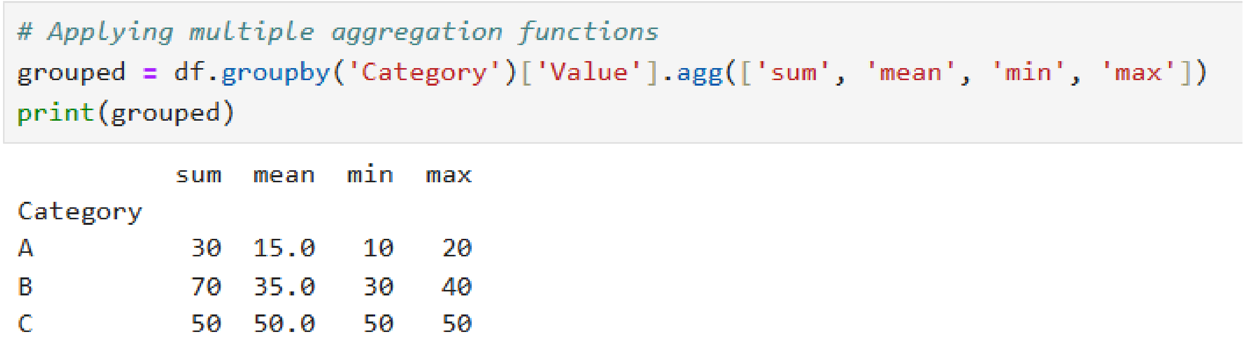
```
# Creating a new DataFrame
data = {
    'Category': ['A', 'A', 'B', 'B', 'C'],
    'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)

# Grouping by 'Category' and calculating the sum of 'Value'
grouped = df.groupby('Category')['Value'].sum()
print(grouped)
```



```
Category
A    30
B    70
C    50
Name: Value, dtype: int64
```

### Aggregating Multiple Functions

You can apply multiple aggregation functions at once.



```
# Applying multiple aggregation functions
grouped = df.groupby('Category')['Value'].agg(['sum', 'mean', 'min', 'max'])
print(grouped)
```

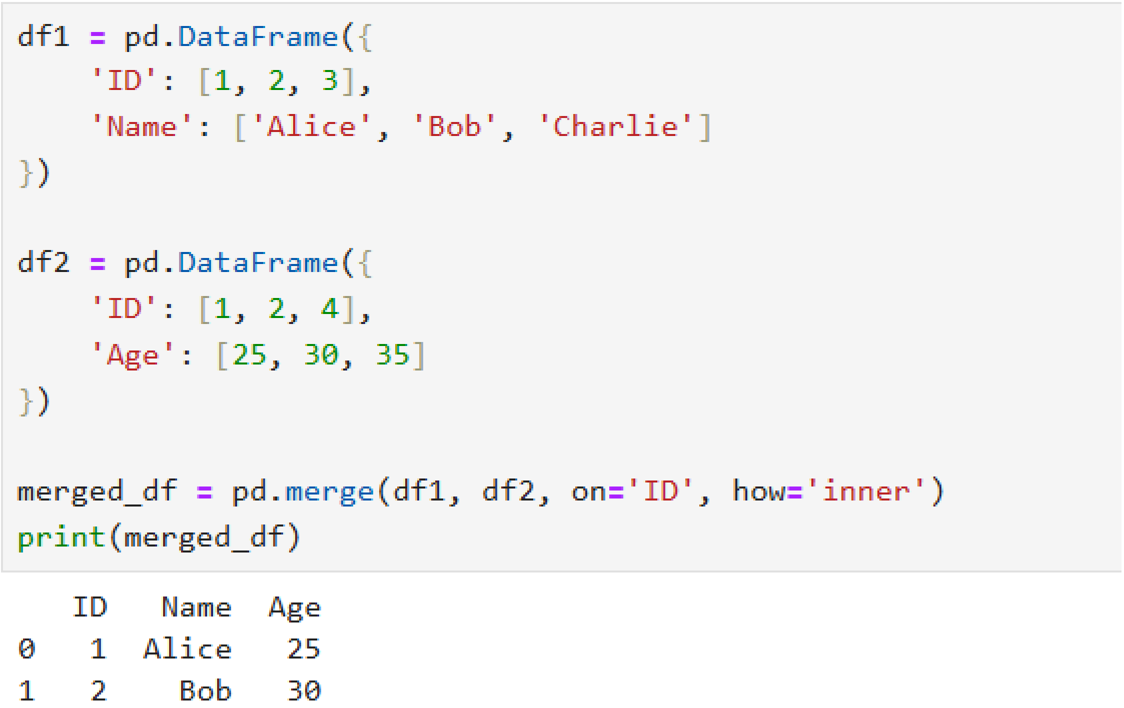
```
          sum  mean  min  max
Category
A          30  15.0   10   20
B          70  35.0   30   40
C          50  50.0   50   50
```

### 6. Merging and Joining DataFrames

Pandas provides functions for merging and joining DataFrames, similar to SQL operations.

### Merge

merge() is used to merge two DataFrames based on a common column.

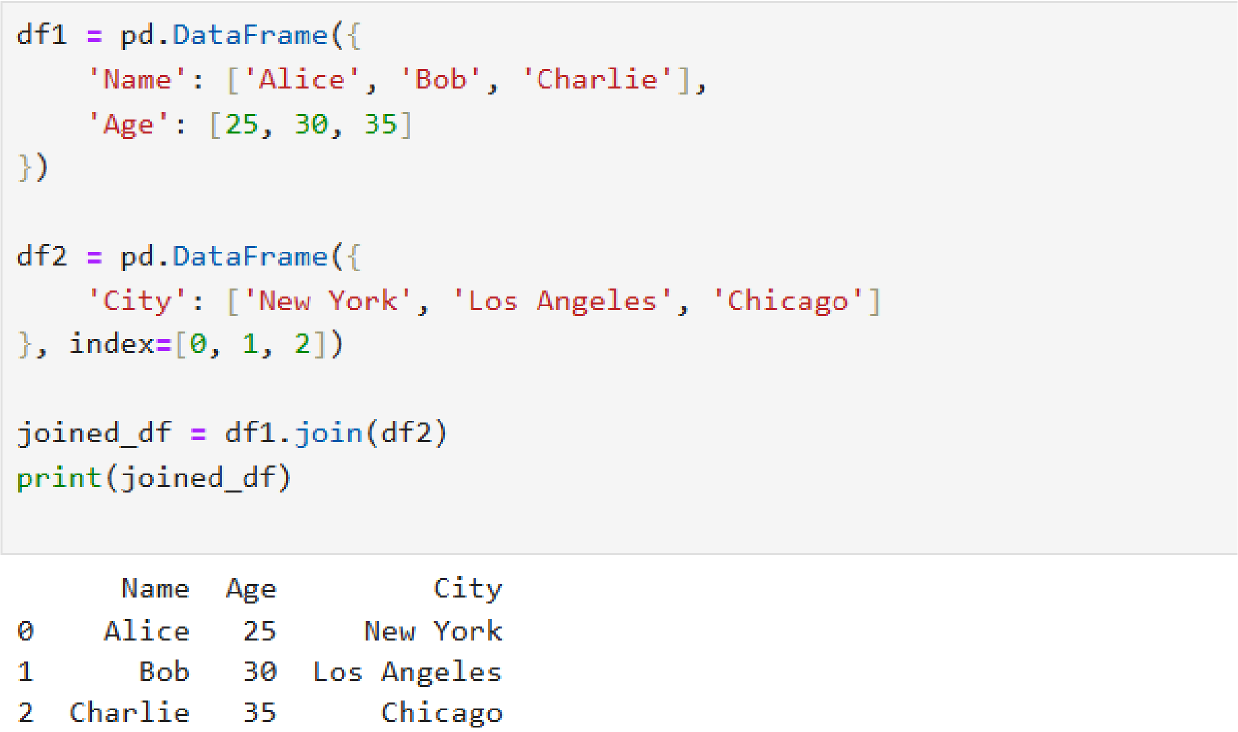


```
df1 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'ID': [1, 2, 4],
    'Age': [25, 30, 35]
})

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

```
   ID   Name  Age
0   1  Alice   25
1   2    Bob   30
```

### Join

The join() method is used to join DataFrames using their index.

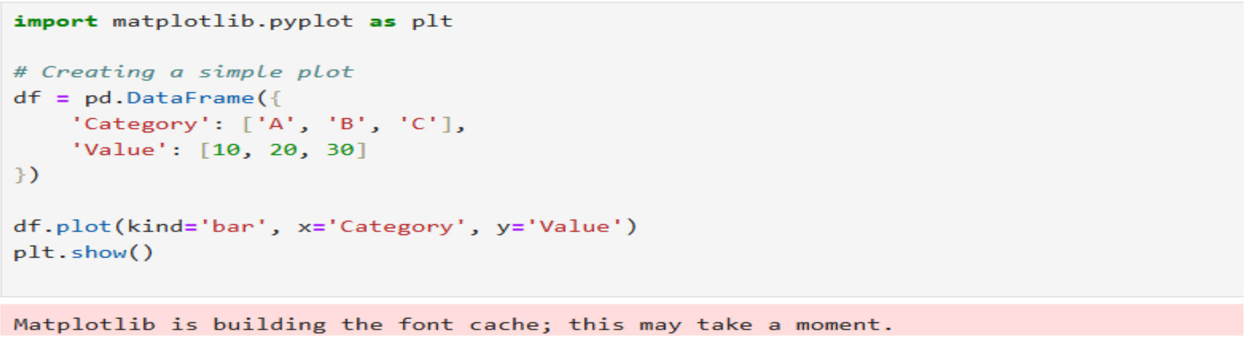


```
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})

df2 = pd.DataFrame({
    'City': ['New York', 'Los Angeles', 'Chicago']
}, index=[0, 1, 2])

joined_df = df1.join(df2)
print(joined_df)
```

```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

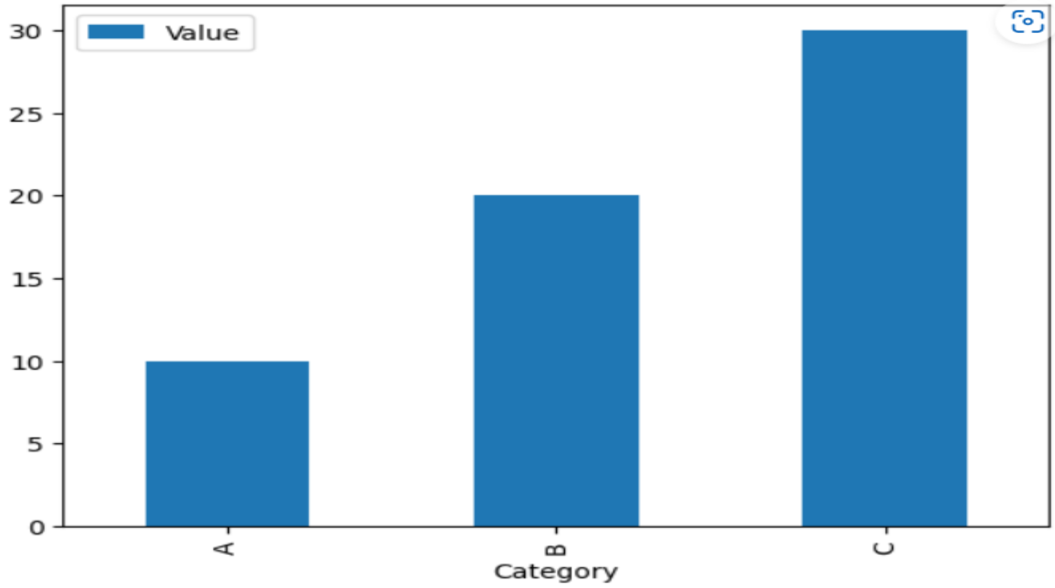
### 7. Plotting with Pandas

Pandas integrates with Matplotlib to plot data.



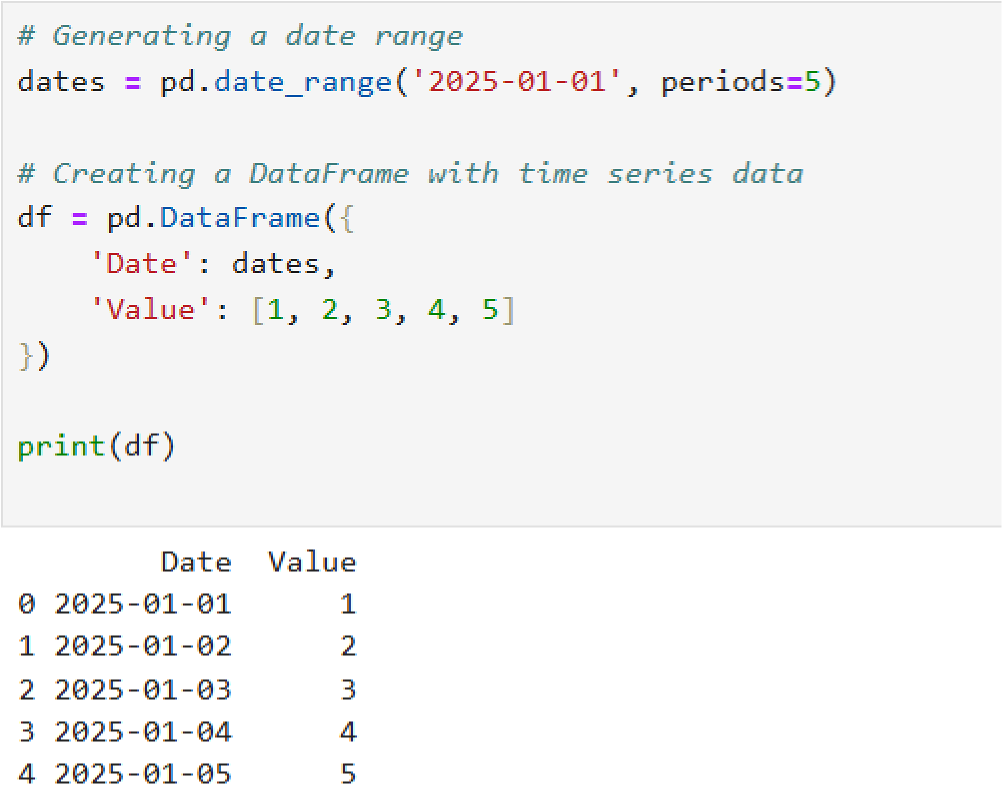
```
import matplotlib.pyplot as plt

# Creating a simple plot
df = pd.DataFrame({
    'Category': ['A', 'B', 'C'],
    'Value': [10, 20, 30]
})

df.plot(kind='bar', x='Category', y='Value')
plt.show()
```

```
Matplotlib is building the font cache; this may take a moment.
```



### 8. Handling Time Series Data

Pandas has powerful features for handling time series data.



```
# Generating a date range
dates = pd.date_range('2025-01-01', periods=5)

# Creating a DataFrame with time series data
df = pd.DataFrame({
    'Date': dates,
    'Value': [1, 2, 3, 4, 5]
})

print(df)
```

```
        Date  Value
0 2025-01-01      1
1 2025-01-02      2
2 2025-01-03      3
3 2025-01-04      4
4 2025-01-05      5
```

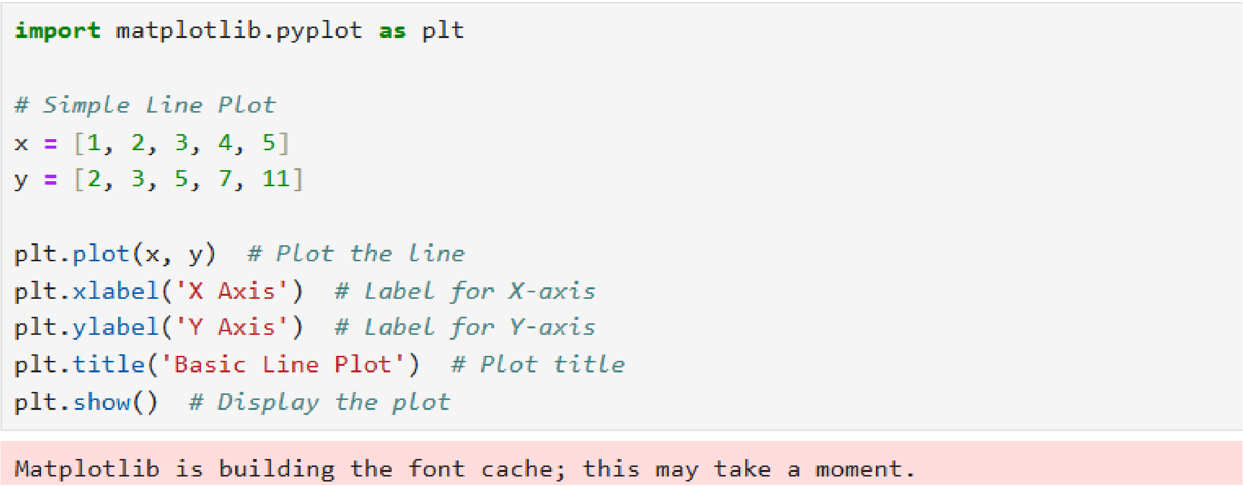
# Python Matplotlib

### Introduction of Matplotlib

Matplotlib is one of the most widely used libraries in Python for data visualization. It provides an object-oriented API for embedding plots into applications or using it interactively in a Python script, Jupyter notebook, or other environments. The core functionality of Matplotlib lies in its ability to generate a wide variety of static, animated, and interactive plots.

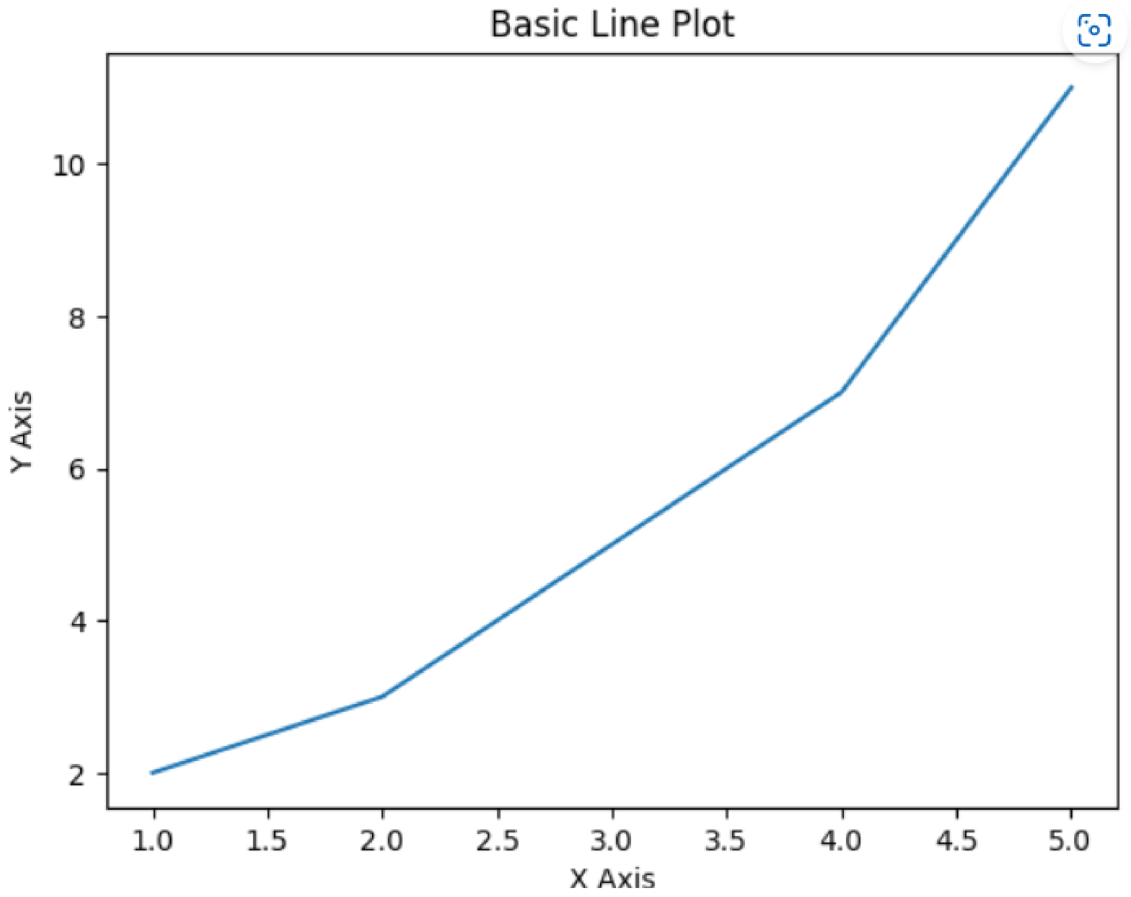
### 1. Basic Plotting Functions

- **plot():** This is the most basic function for creating line plots.



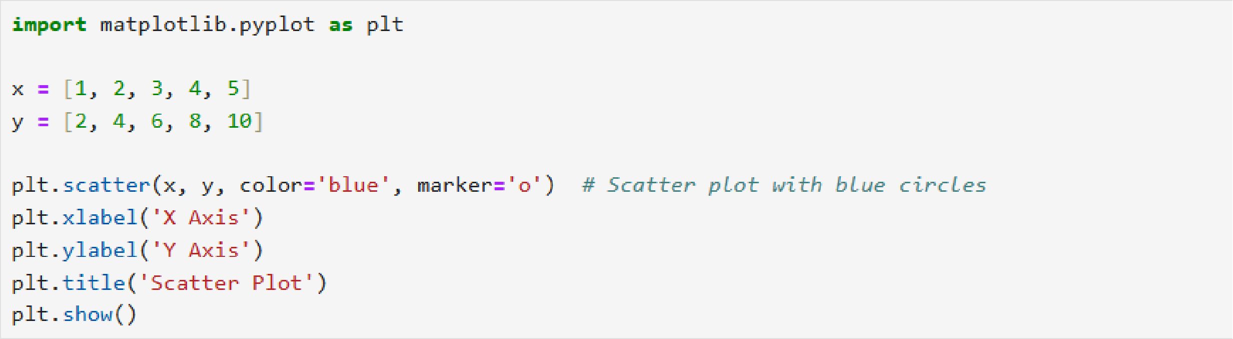
```python
import matplotlib.pyplot as plt

# Simple Line Plot
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

plt.plot(x, y)  # Plot the line
plt.xlabel('X Axis')  # Label for X-axis
plt.ylabel('Y Axis')  # Label for Y-axis
plt.title('Basic Line Plot')  # Plot title
plt.show()  # Display the plot
```

```
Matplotlib is building the font cache; this may take a moment.
```

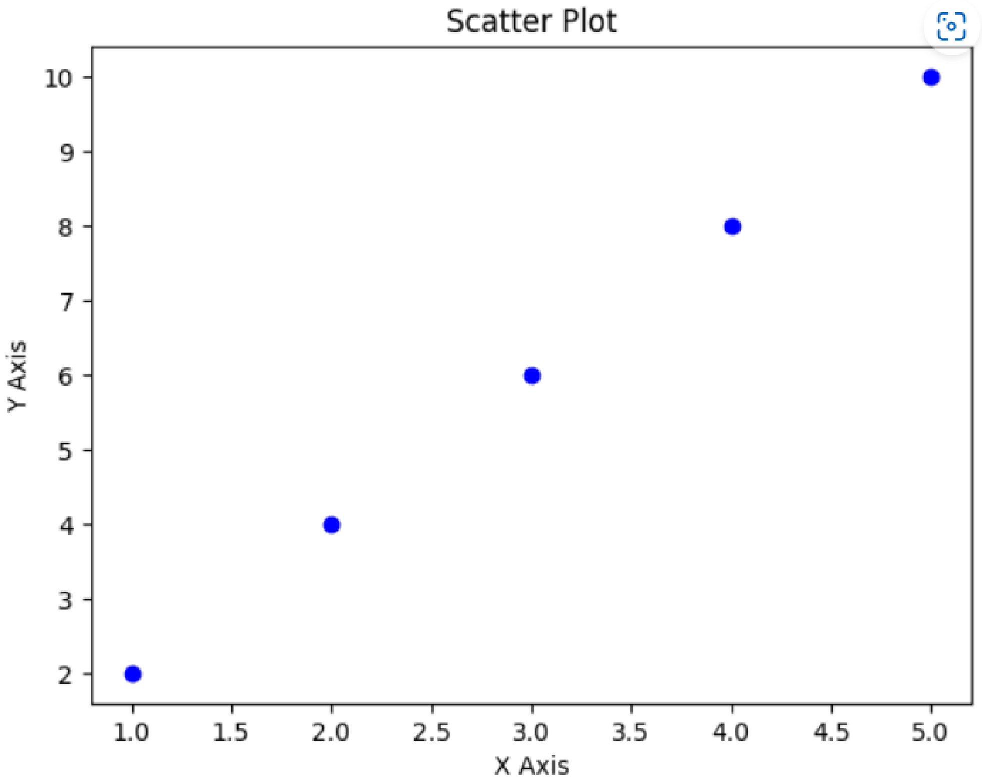


## 2. Scatter Plot

- **scatter():** Used to create scatter plots.

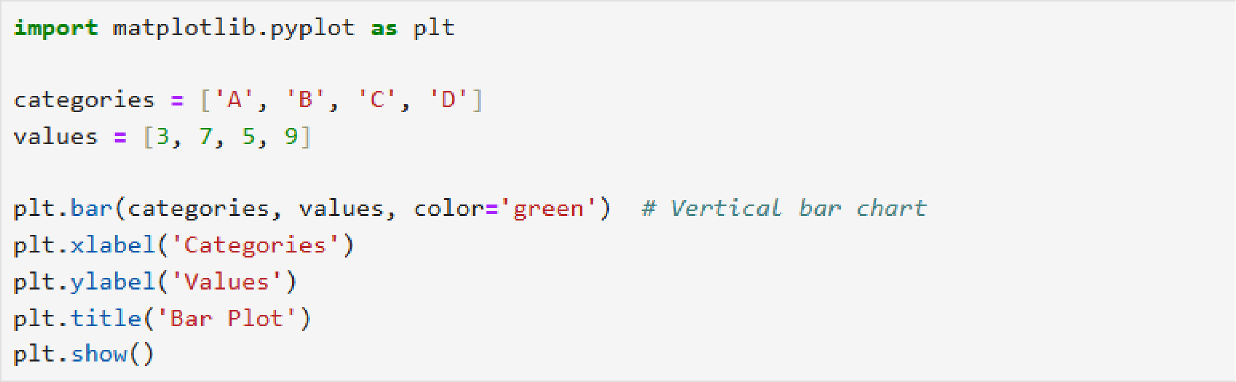


```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.scatter(x, y, color='blue', marker='o')  # Scatter plot with blue circles
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Scatter Plot')
plt.show()
```

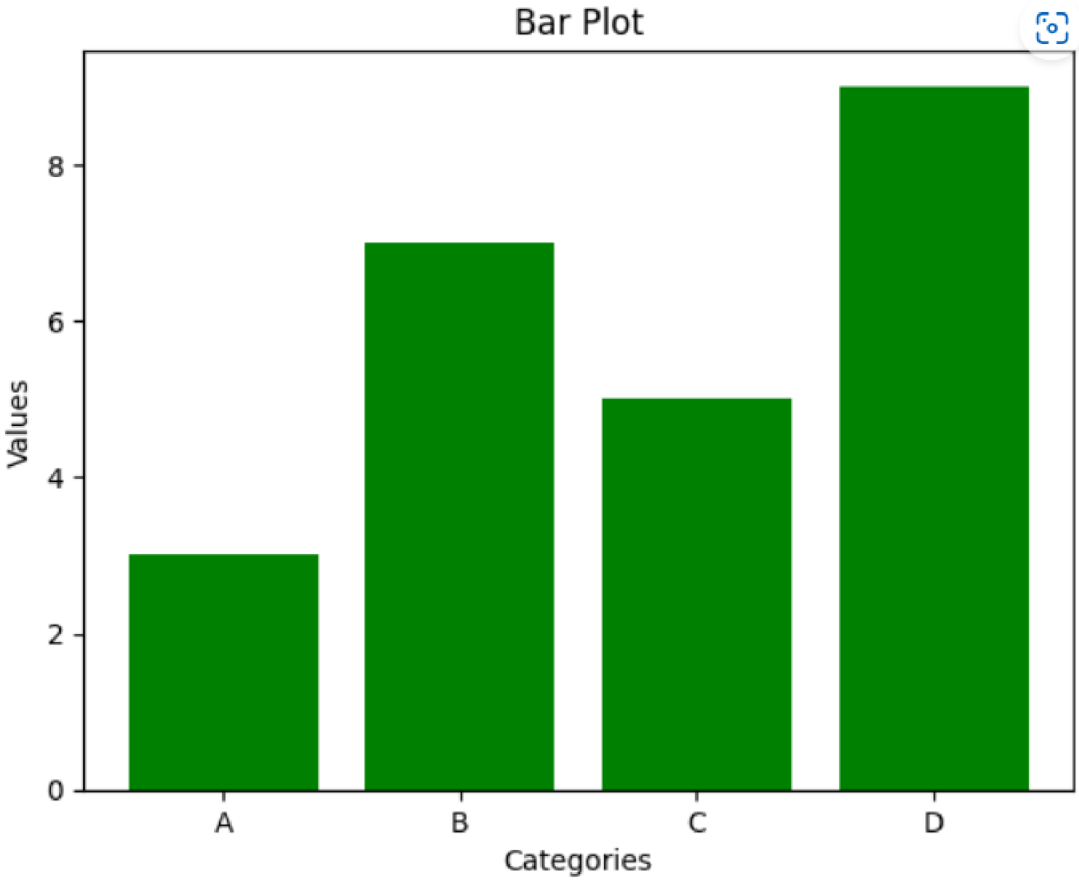


### 3. Bar Plot

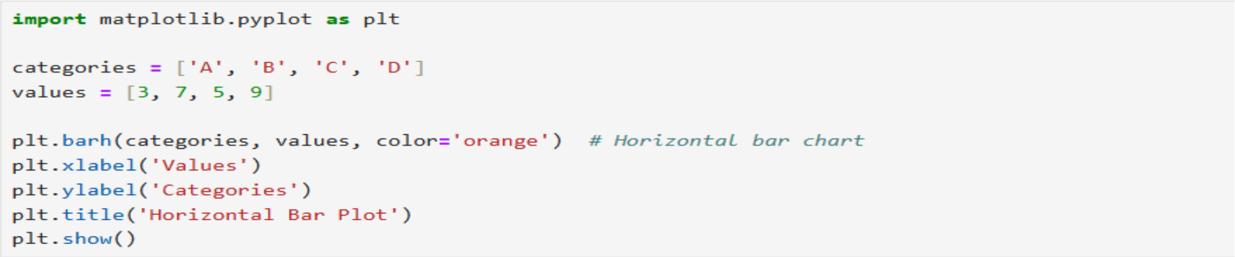
- **bar():** Used to create bar charts.



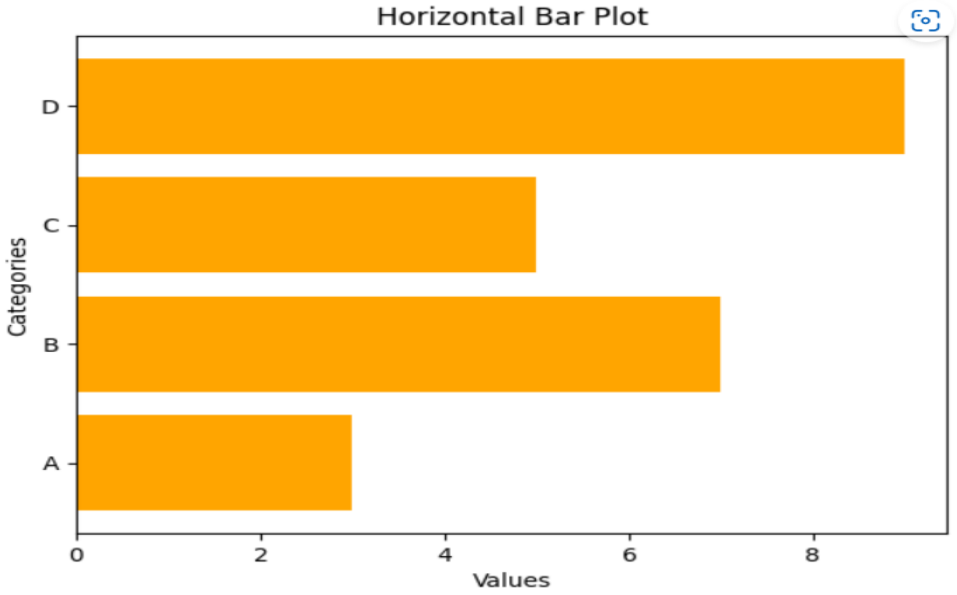
```
import matplotlib.pyplot as plt

categories = ['A', 'B', 'C', 'D']
values = [3, 7, 5, 9]

plt.bar(categories, values, color='green')  # Vertical bar chart
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot')
plt.show()
```



**barh():** Used to create horizontal bar charts.

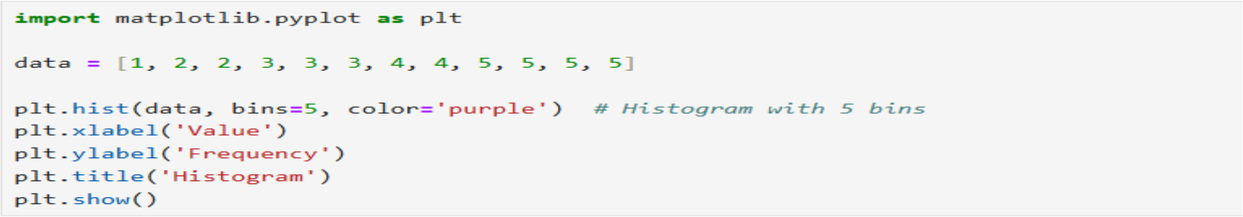


```
import matplotlib.pyplot as plt

categories = ['A', 'B', 'C', 'D']
values = [3, 7, 5, 9]

plt.barh(categories, values, color='orange')  # Horizontal bar chart
plt.xlabel('Values')
plt.ylabel('Categories')
plt.title('Horizontal Bar Plot')
plt.show()
```

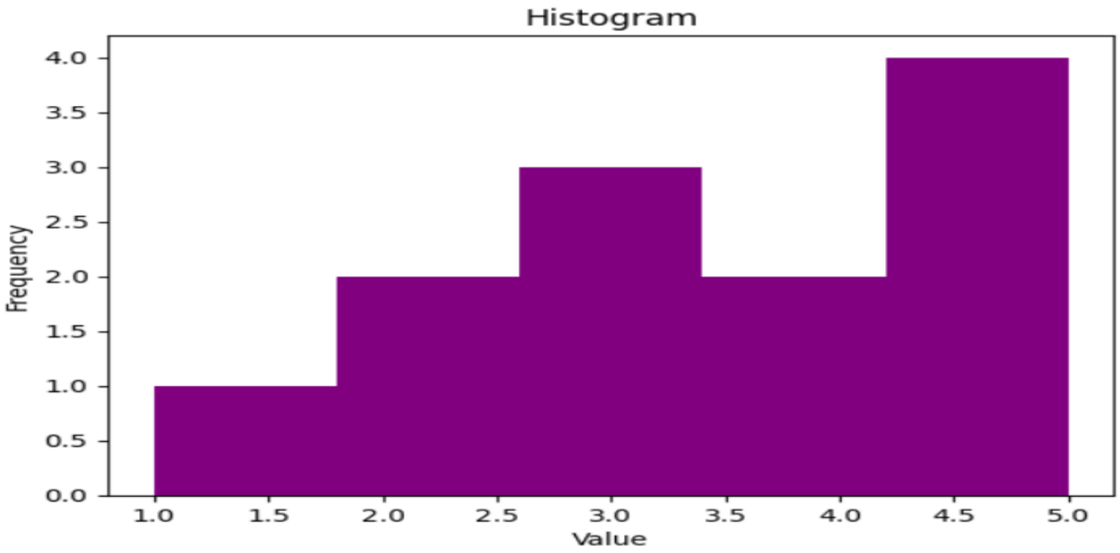


## 4. Histogram

- **hist():** Used to create histograms to visualize the distribution of data.

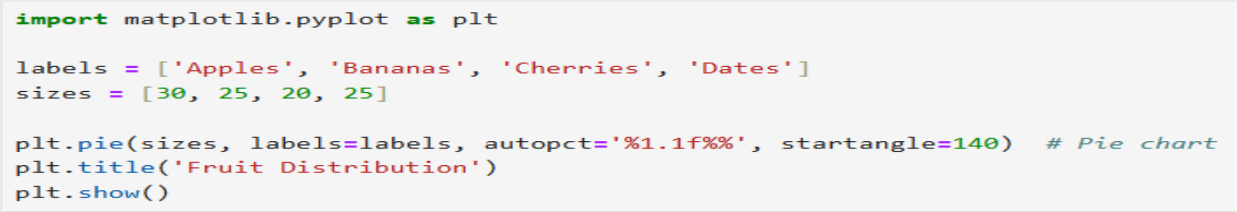


```
import matplotlib.pyplot as plt

data = [1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5]

plt.hist(data, bins=5, color='purple')  # Histogram with 5 bins
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram')
plt.show()
```

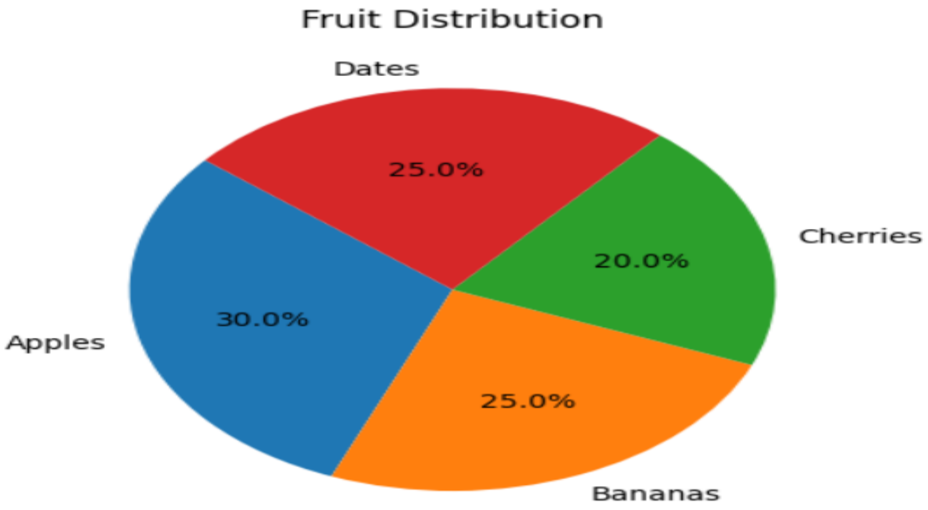


### 5. Pie Chart

- **pie():** Used to create pie charts.

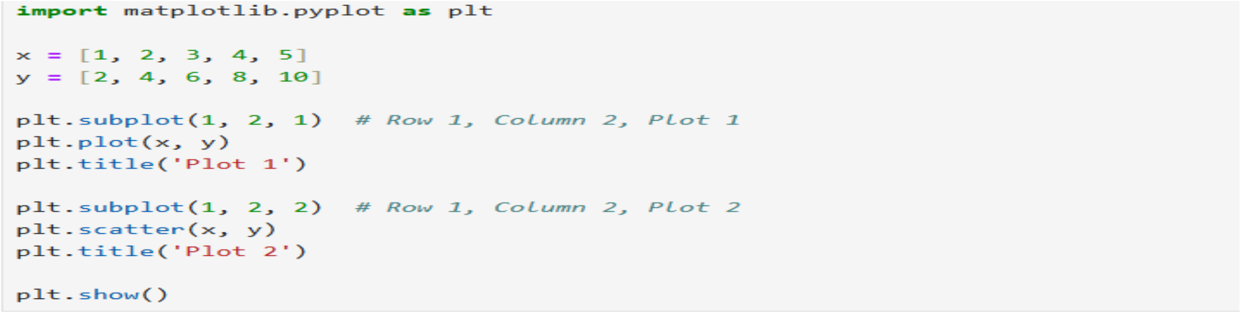


```
import matplotlib.pyplot as plt

labels = ['Apples', 'Bananas', 'Cherries', 'Dates']
sizes = [30, 25, 20, 25]

plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140)  # Pie chart
plt.title('Fruit Distribution')
plt.show()
```

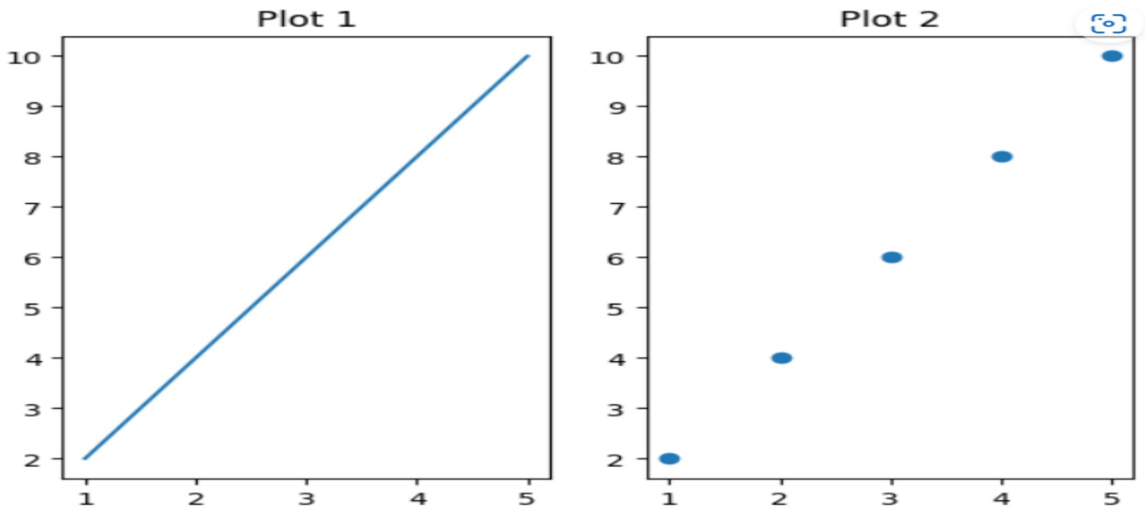


### 6. Subplots

- **subplot():** Allows you to create multiple plots in a single figure by arranging them in a grid.

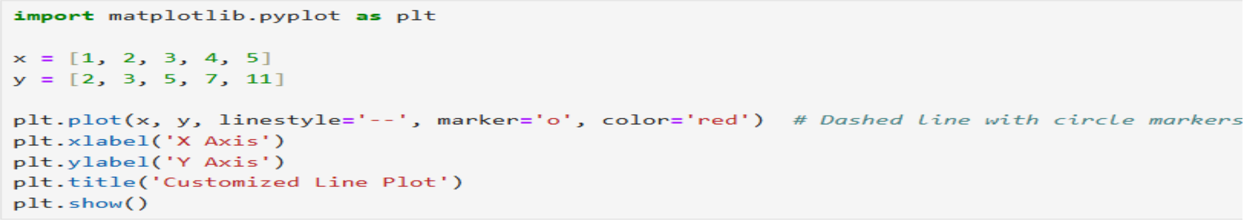


```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.subplot(1, 2, 1)  # Row 1, Column 2, Plot 1
plt.plot(x, y)
plt.title('Plot 1')

plt.subplot(1, 2, 2)  # Row 1, Column 2, Plot 2
plt.scatter(x, y)
plt.title('Plot 2')

plt.show()
```

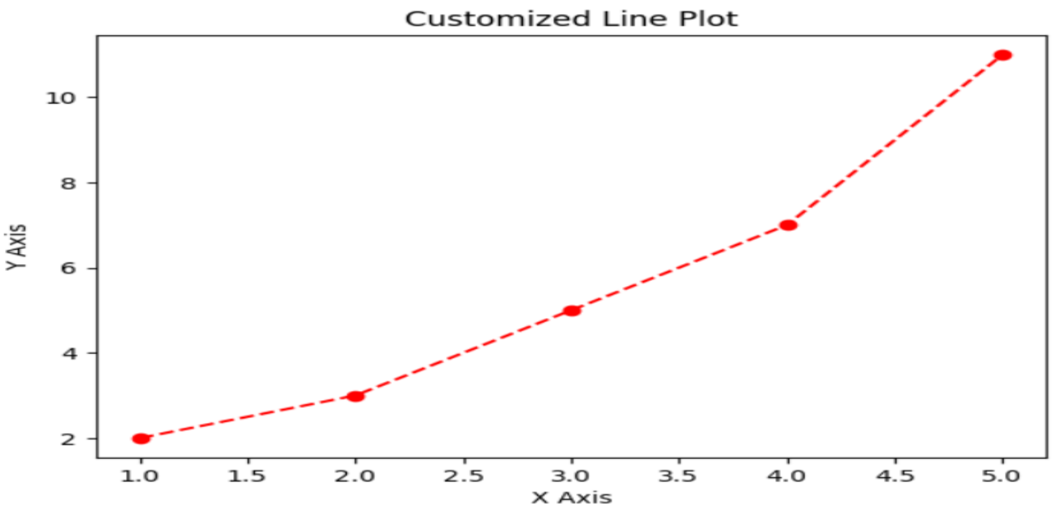


## 7. Styling and Customization

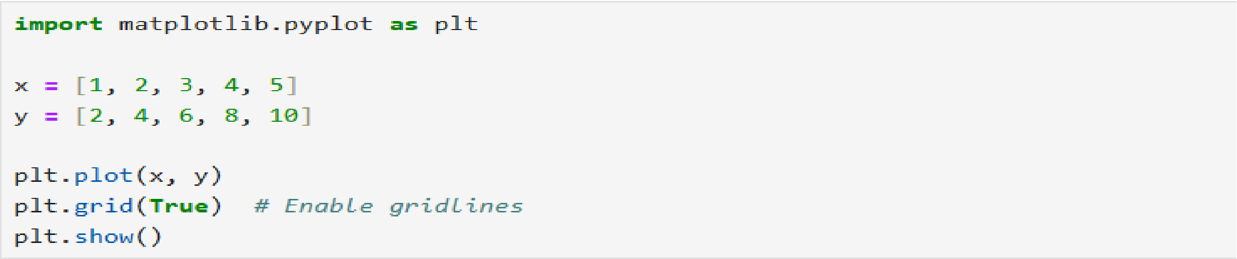
- Changing Line Styles and Markers: Matplotlib allows you to customize your plots by modifying the line styles, markers, colors, etc.



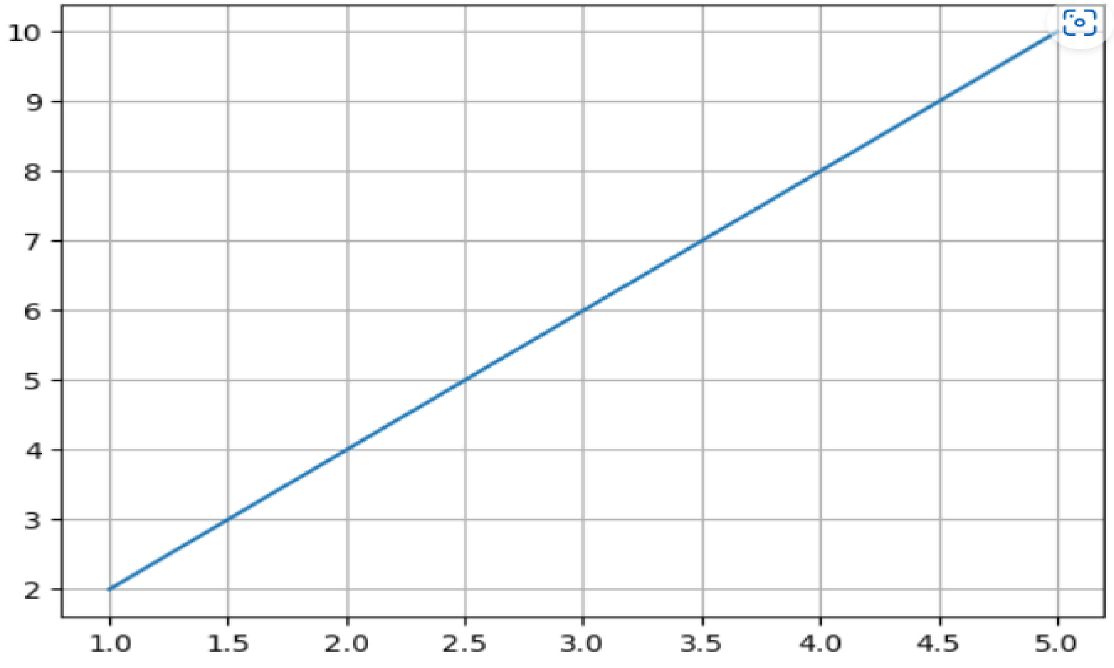
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

plt.plot(x, y, linestyle='--', marker='o', color='red')  # Dashed line with circle markers
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Customized Line Plot')
plt.show()
```



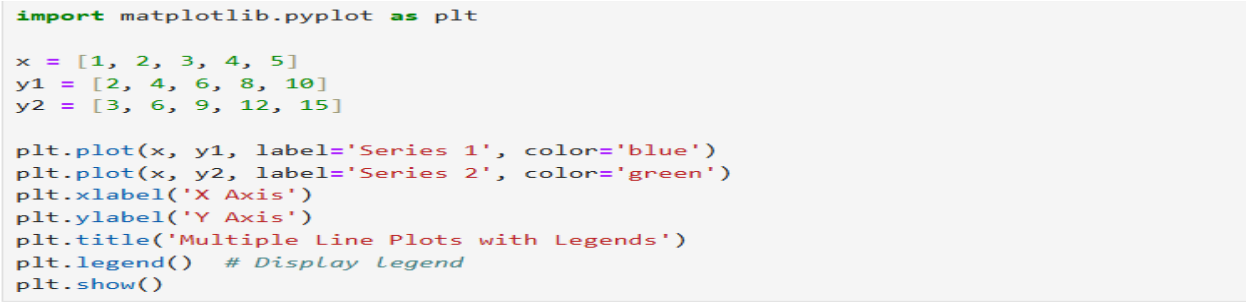
**Gridlines:** You can add gridlines to your plot for better readability.



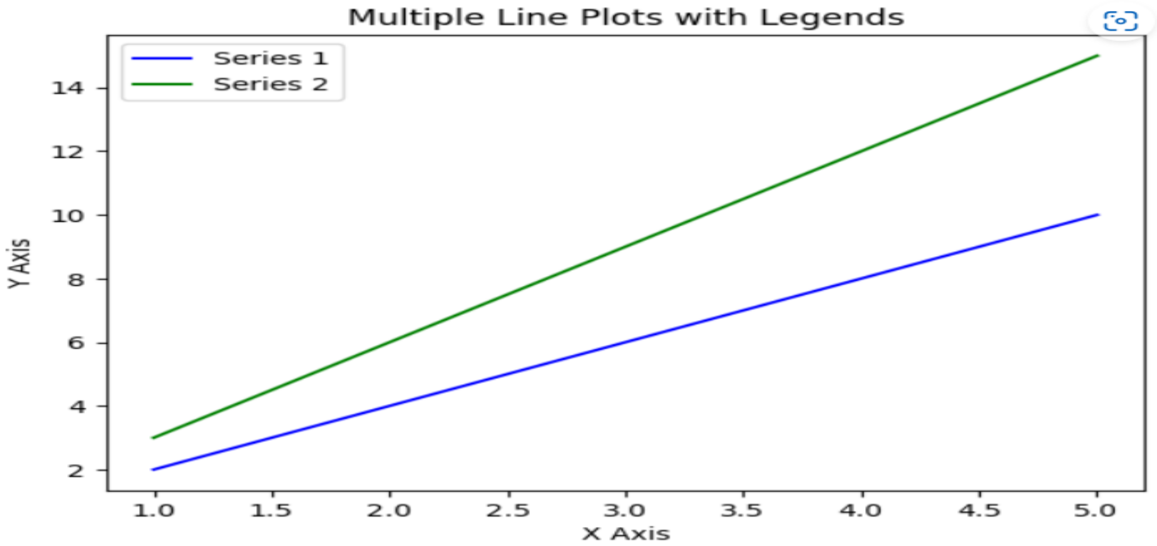
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.plot(x, y)
plt.grid(True)   # Enable gridlines
plt.show()
```



**Legends:** Add legends to differentiate multiple data series in a plot.

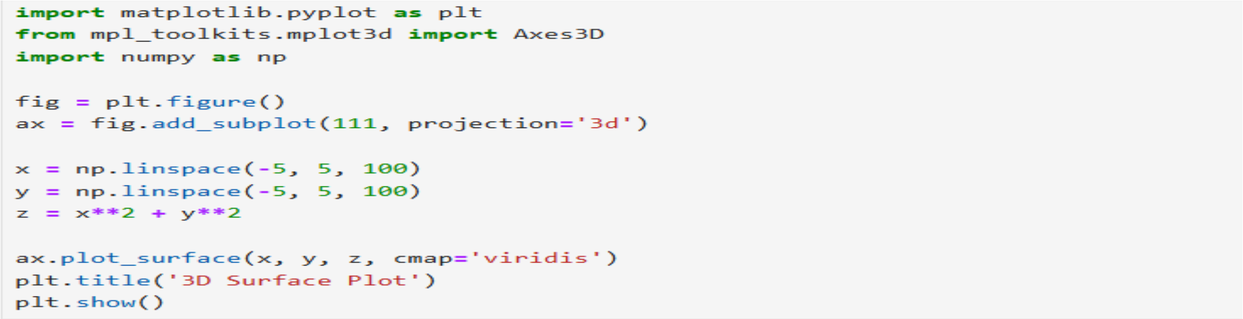


```python
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
y2 = [3, 6, 9, 12, 15]

plt.plot(x, y1, label='Series 1', color='blue')
plt.plot(x, y2, label='Series 2', color='green')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Multiple Line Plots with Legends')
plt.legend()  # Display legend
plt.show()
```

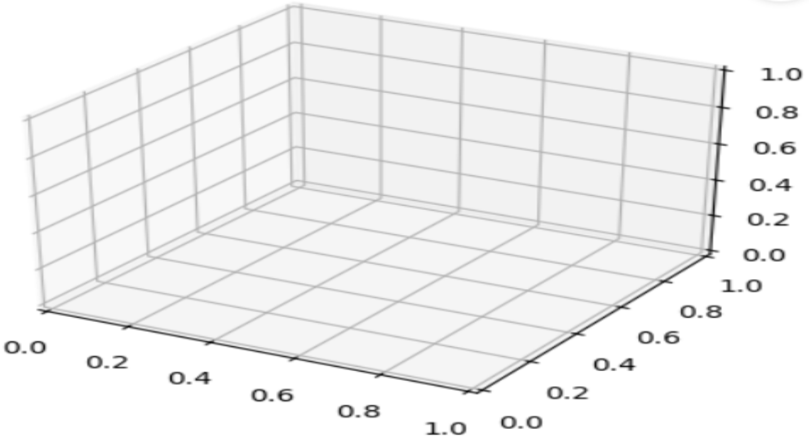


### 8. 3D Plotting

- **Axes3D**: To plot 3D data, you need to use the Axes3D class from the mpl_toolkits.mplot3d module.



```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
z = x**2 + y**2

ax.plot_surface(x, y, z, cmap='viridis')
plt.title('3D Surface Plot')
plt.show()
```

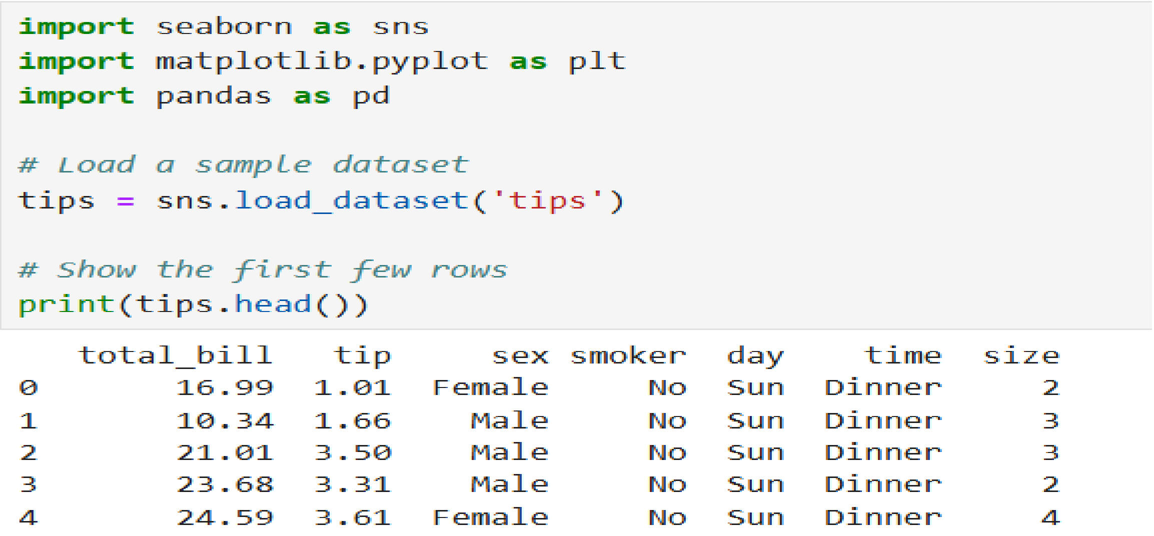


## Python Seaborn

Seaborn is a powerful Python visualization library built on top of Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics. It simplifies the process of creating complex visualizations like heatmaps, bar plots, and violin plots. Seaborn integrates seamlessly with pandas DataFrames, making it easier to visualize data directly from structured datasets.

### 1. Setting up Seaborn and Importing Data

To use Seaborn, we need to import it along with other necessary libraries like Matplotlib and pandas.

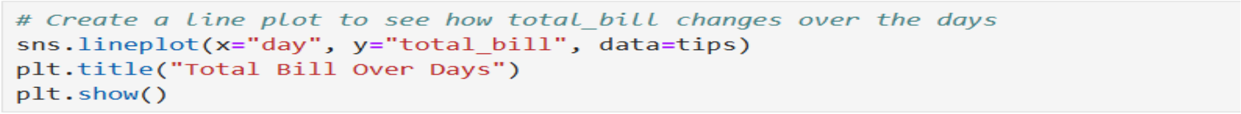


```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Load a sample dataset
tips = sns.load_dataset('tips')

# Show the first few rows
print(tips.head())
```

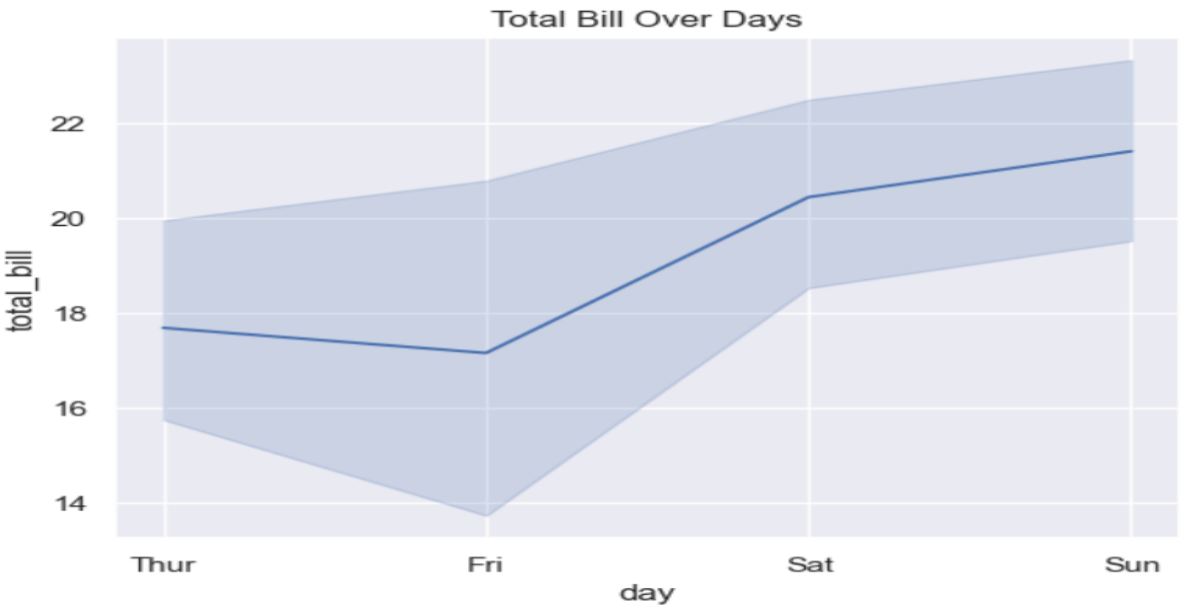
```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

### 2. Line Plot (sns.lineplot):

Let's plot a line chart showing how the total bill changes over the days of the week.

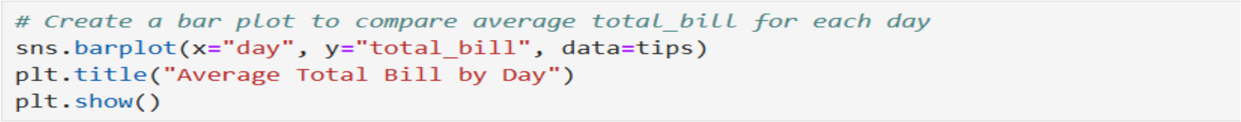


```python
# Create a line plot to see how total_bill changes over the days
sns.lineplot(x="day", y="total_bill", data=tips)
plt.title("Total Bill Over Days")
plt.show()
```

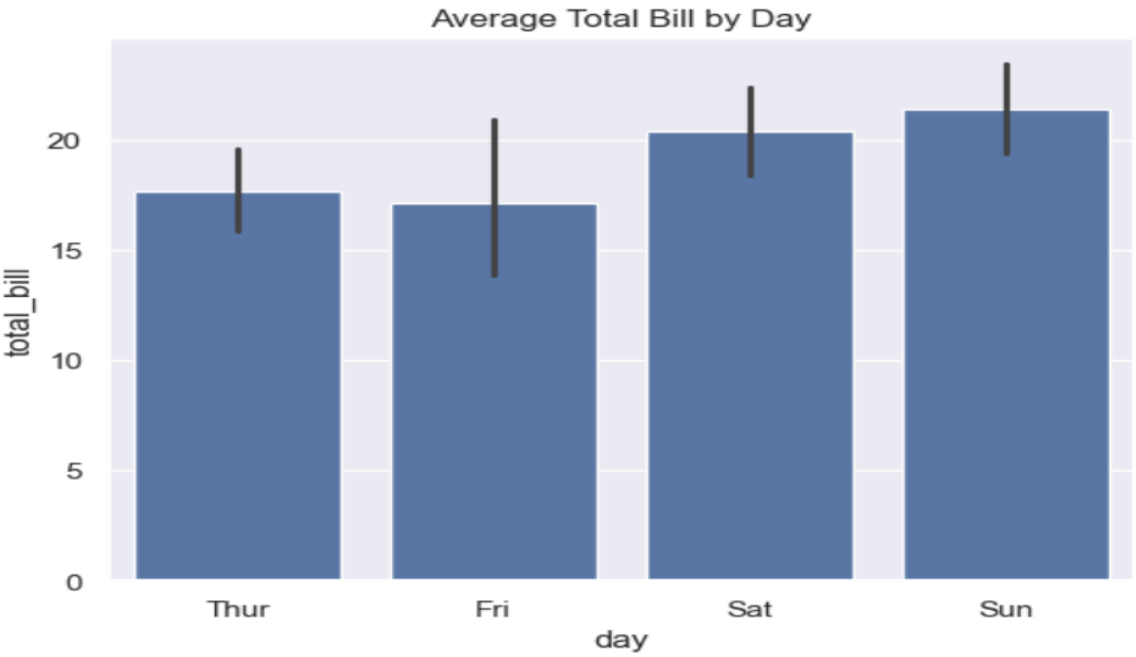


### 3. Bar Plot (sns.barplot):

A bar plot can be used to compare the average total bill for each day.

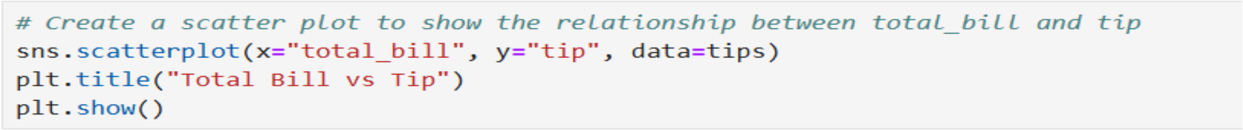


```
# Create a bar plot to compare average total_bill for each day
sns.barplot(x="day", y="total_bill", data=tips)
plt.title("Average Total Bill by Day")
plt.show()
```

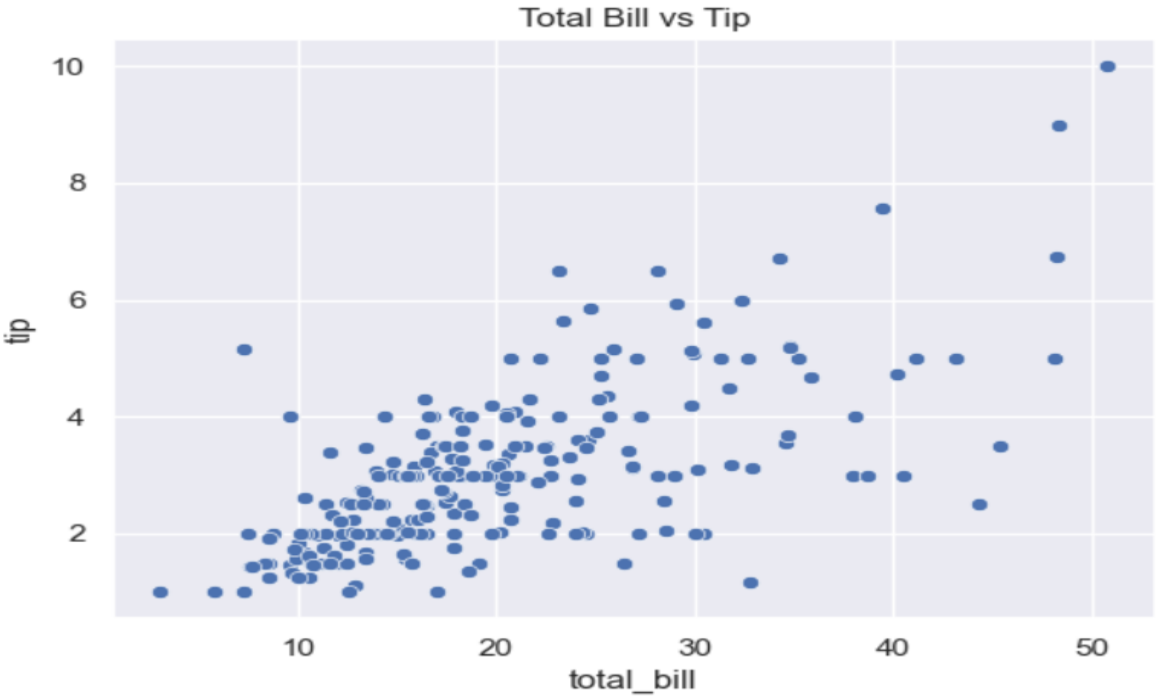


### 4. Scatter Plot (sns.scatterplot):

A scatter plot is useful to see the relationship between total bill and tip.

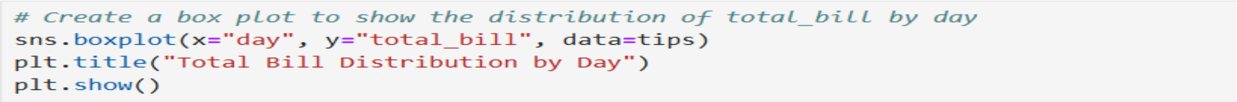


```
# Create a scatter plot to show the relationship between total_bill and tip
sns.scatterplot(x="total_bill", y="tip", data=tips)
plt.title("Total Bill vs Tip")
plt.show()
```

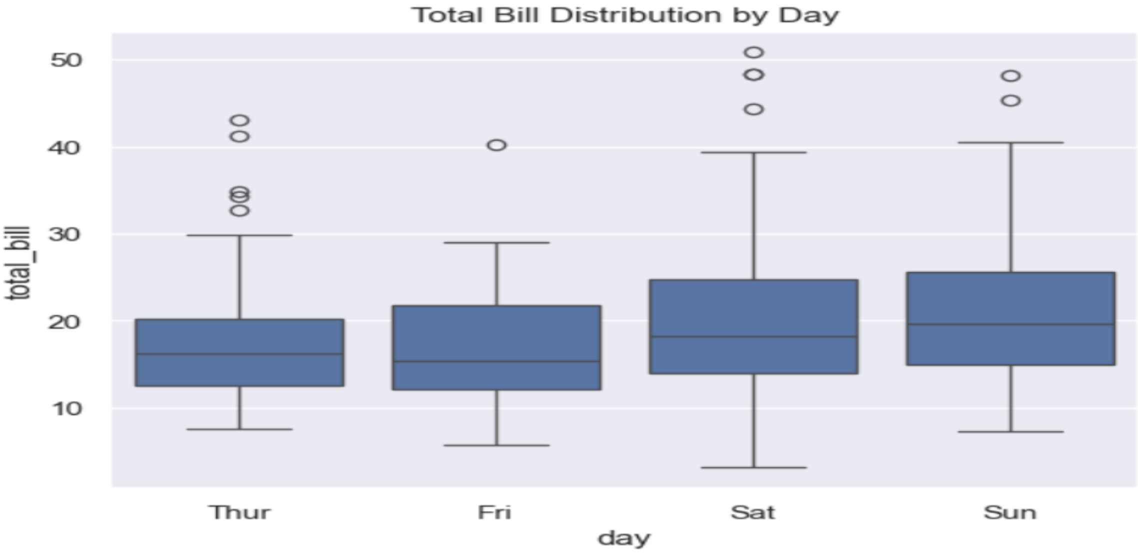


### 5. Box Plot (sns.boxplot):

Box plots show the distribution of total bill amounts for each day, which can help us understand the spread of the data.

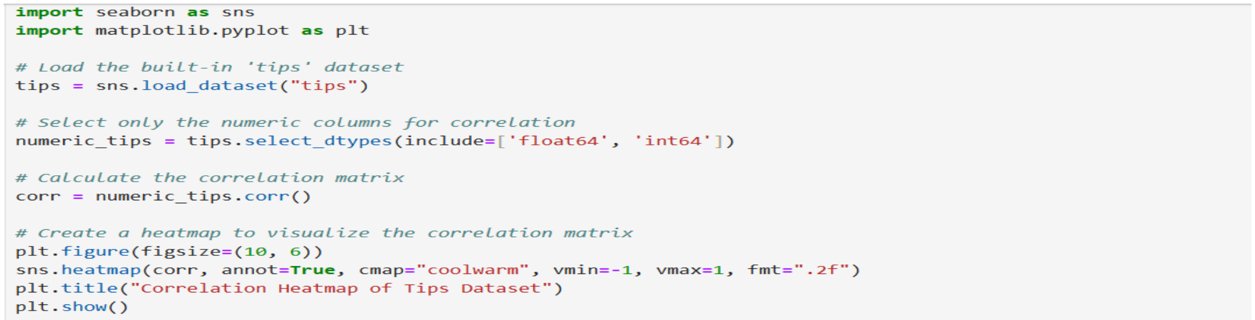


```python
# Create a box plot to show the distribution of total_bill by day
sns.boxplot(x="day", y="total_bill", data=tips)
plt.title("Total Bill Distribution by Day")
plt.show()
```

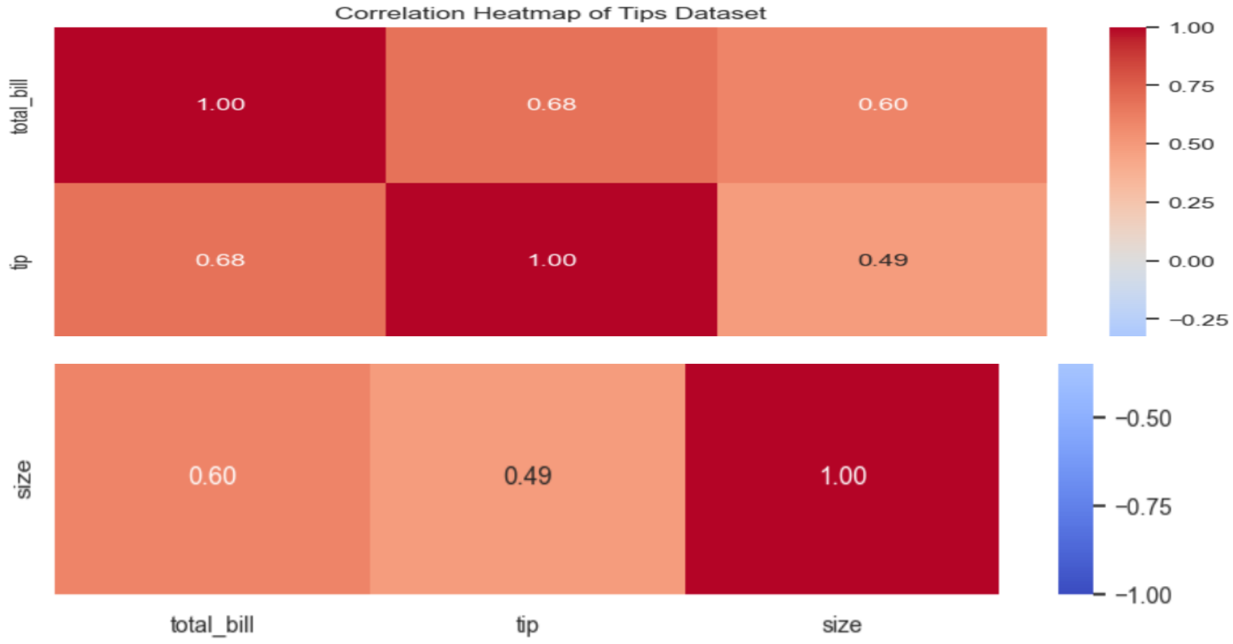


### 6. Heatmap (sns.heatmap):

A heatmap can show the correlation between the numeric columns in the dataset.

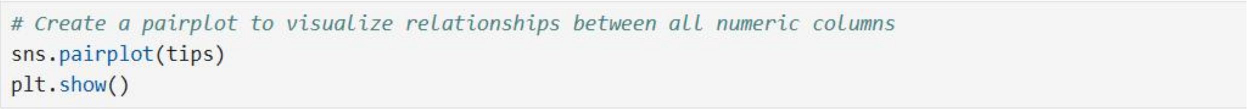


```python
import seaborn as sns
import matplotlib.pyplot as plt

# Load the built-in 'tips' dataset
tips = sns.load_dataset("tips")

# Select only the numeric columns for correlation
numeric_tips = tips.select_dtypes(include=['float64', 'int64'])

# Calculate the correlation matrix
corr = numeric_tips.corr()

# Create a heatmap to visualize the correlation matrix
plt.figure(figsize=(10, 6))
sns.heatmap(corr, annot=True, cmap="coolwarm", vmin=-1, vmax=1, fmt=".2f")
plt.title("Correlation Heatmap of Tips Dataset")
plt.show()
```

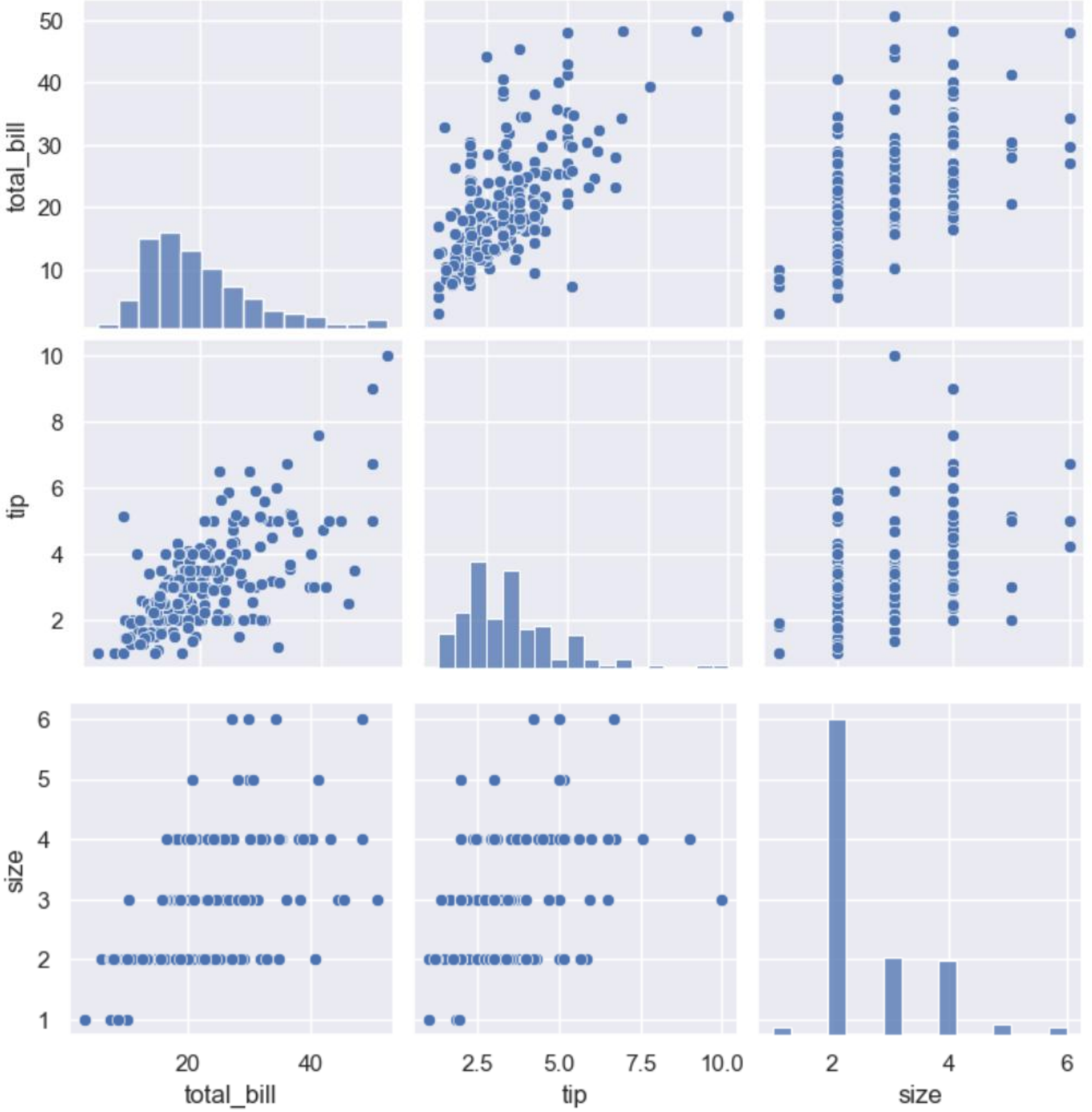


**7. Pairplot (sns.pairplot):**

A pairplot allows you to see how all the numerical columns in the dataset relate to each other.

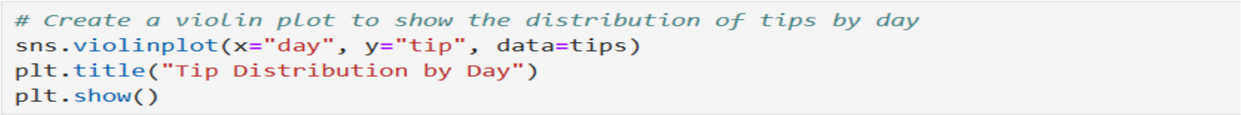


```
# Create a pairplot to visualize relationships between all numeric columns
sns.pairplot(tips)
plt.show()
```

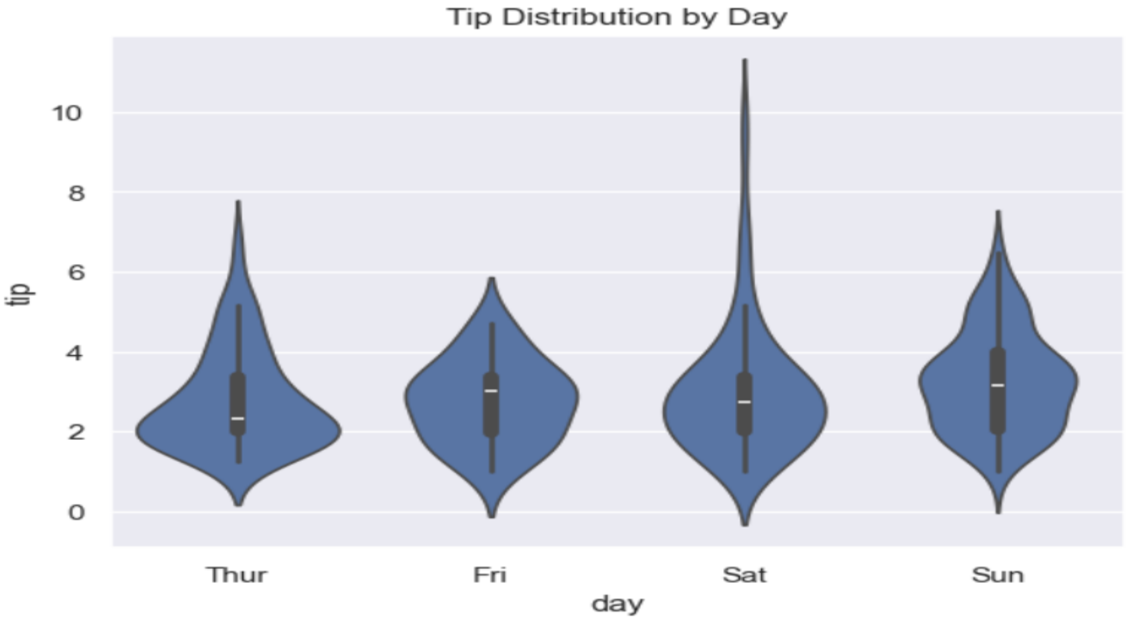


### 8. Violin Plot (sns.violinplot):

A violin plot combines aspects of both box plots and density plots. It shows the distribution of a numeric variable for different categories.



```
# Create a violin plot to show the distribution of tips by day
sns.violinplot(x="day", y="tip", data=tips)
plt.title("Tip Distribution by Day")
plt.show()
```

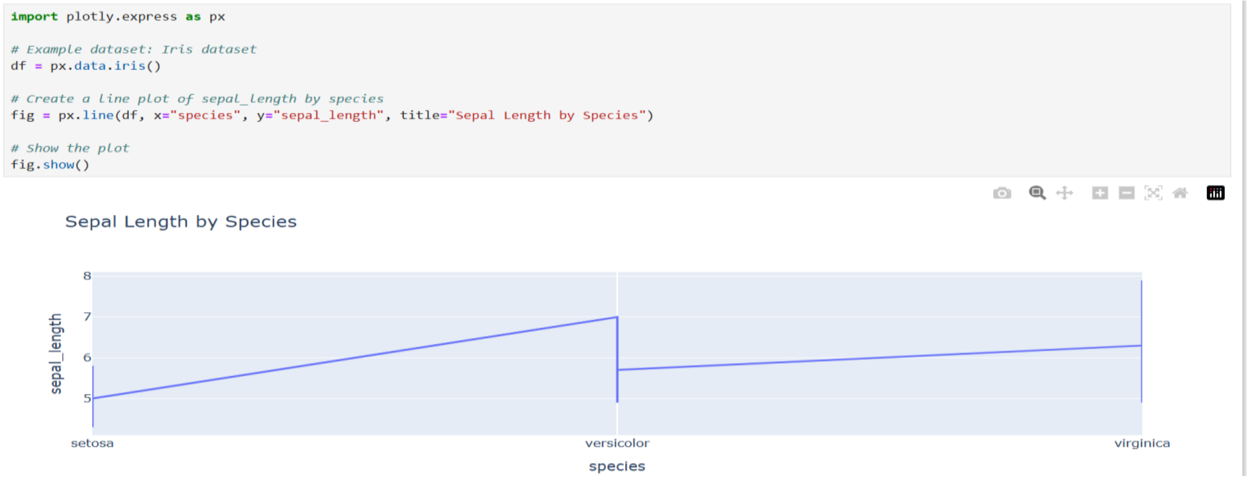


## Python Plotly Library

Plotly is a powerful and interactive visualization library in Python that enables you to create a wide range of static, animated, and interactive plots. It is widely used for creating dashboards, web applications, and various types of charts, such as line plots, bar plots, pie charts, scatter plots, heatmaps, and more.

## 1. Basic Example using Plotly Express

Plotly Express is a simple way to create charts with minimal code. Here's an example using Plotly Express to create a basic line plot.

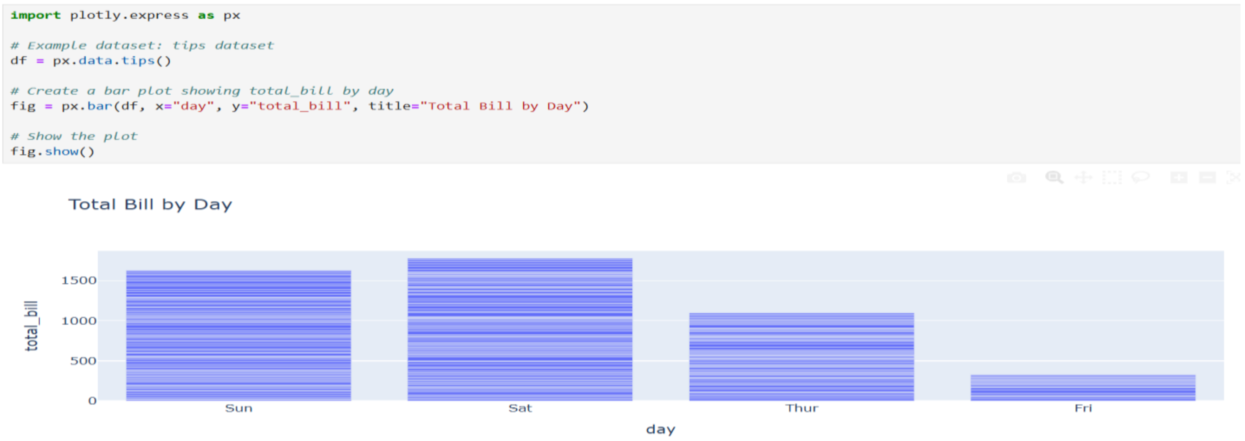


```
import plotly.express as px

# Example dataset: Iris dataset
df = px.data.iris()

# Create a line plot of sepal_length by species
fig = px.line(df, x="species", y="sepal_length", title="Sepal Length by Species")

# Show the plot
fig.show()
```

**Explanation**:

- **px.data.iris()**: Loads the built-in iris dataset.
- **px.line()**: Creates a line plot using species as the x-axis and sepal_length as the y-axis.
- **fig.show()**: Displays the plot.

## 2. Bar Plot using Plotly Express

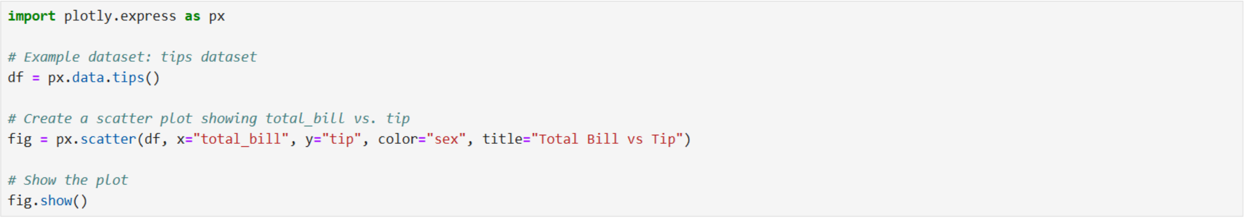


```
import plotly.express as px

# Example dataset: tips dataset
df = px.data.tips()

# Create a bar plot showing total_bill by day
fig = px.bar(df, x="day", y="total_bill", title="Total Bill by Day")

# Show the plot
fig.show()
```

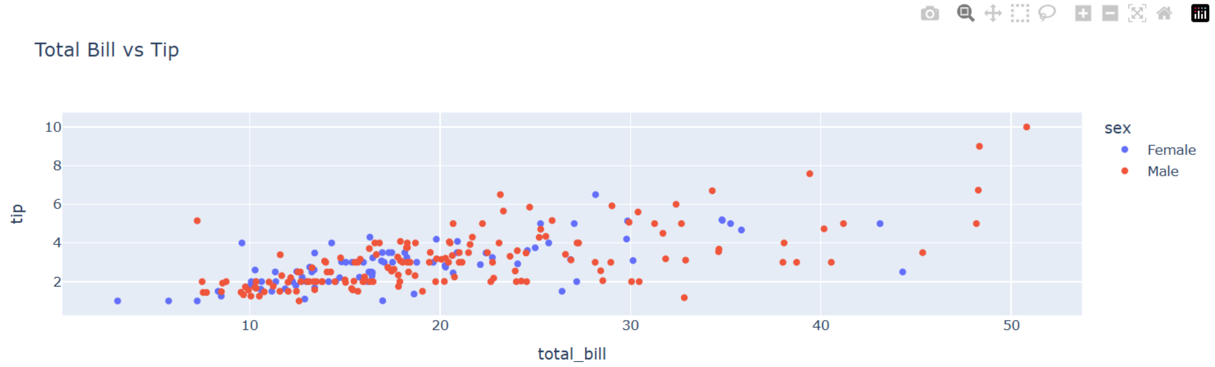
## Explanation:

- **px.bar()**: Creates a bar plot, where the x-axis represents day, and the y-axis represents total_bill.

## 3. **Scatter Plot using Plotly Express**



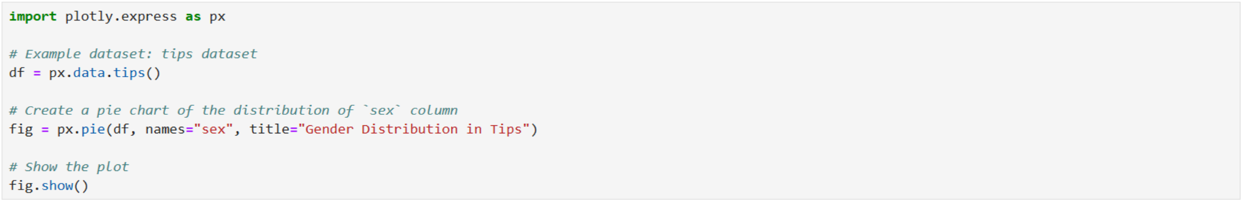
```
import plotly.express as px

# Example dataset: tips dataset
df = px.data.tips()

# Create a scatter plot showing total_bill vs. tip
fig = px.scatter(df, x="total_bill", y="tip", color="sex", title="Total Bill vs Tip")

# Show the plot
fig.show()
```



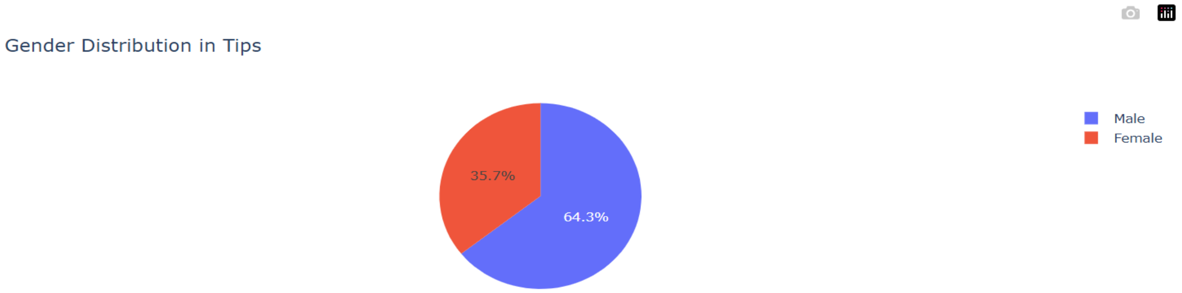
## Explanation:

- **px.scatter()**: Creates a scatter plot with total_bill on the x-axis and tip on the y-axis. The color argument is used to differentiate the points based on sex.

## 4. **Pie Chart using Plotly Express**



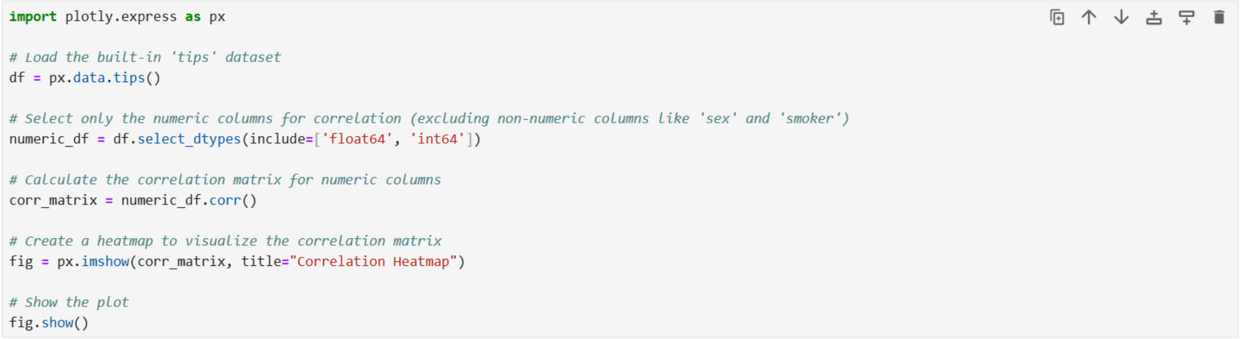
```
import plotly.express as px

# Example dataset: tips dataset
df = px.data.tips()

# Create a pie chart of the distribution of `sex` column
fig = px.pie(df, names="sex", title="Gender Distribution in Tips")

# Show the plot
fig.show()
```



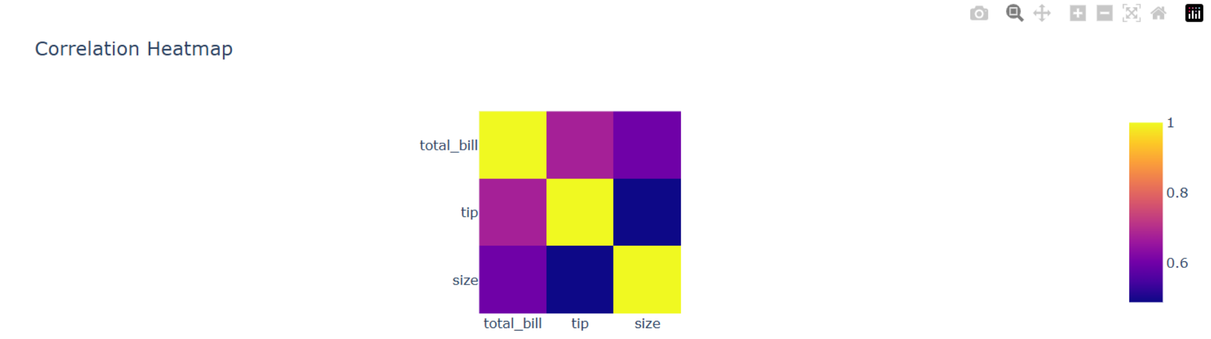
## Explanation:

- **px.pie()**: Creates a pie chart where each slice represents the proportion of male and female entries in the sex column.

## 5. **Heatmap using Plotly Express**



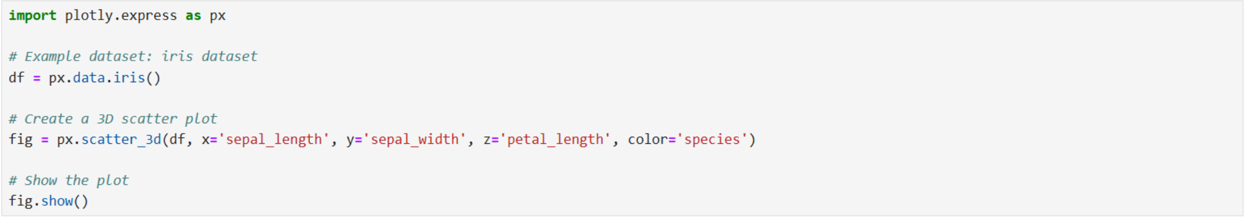
```
import plotly.express as px

# Load the built-in 'tips' dataset
df = px.data.tips()

# Select only the numeric columns for correlation (excluding non-numeric columns like 'sex' and 'smoker')
numeric_df = df.select_dtypes(include=['float64', 'int64'])

# Calculate the correlation matrix for numeric columns
corr_matrix = numeric_df.corr()

# Create a heatmap to visualize the correlation matrix
fig = px.imshow(corr_matrix, title="Correlation Heatmap")

# Show the plot
fig.show()
```



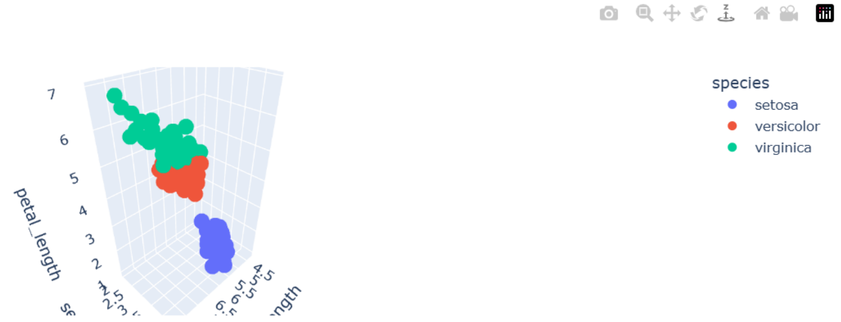
## **Explanation:**

- **px.imshow():** Creates a heatmap. Here, we use the correlation matrix of the tips dataset to generate the heatmap.

## 6. **3D Scatter Plot using Plotly Express**



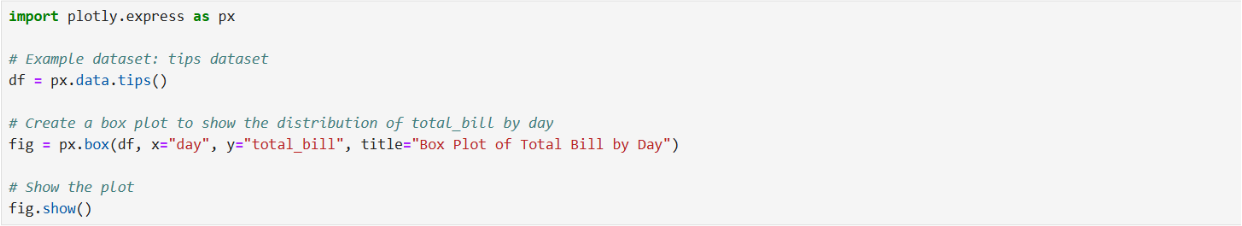
```
import plotly.express as px

# Example dataset: iris dataset
df = px.data.iris()

# Create a 3D scatter plot
fig = px.scatter_3d(df, x='sepal_length', y='sepal_width', z='petal_length', color='species')

# Show the plot
fig.show()
```



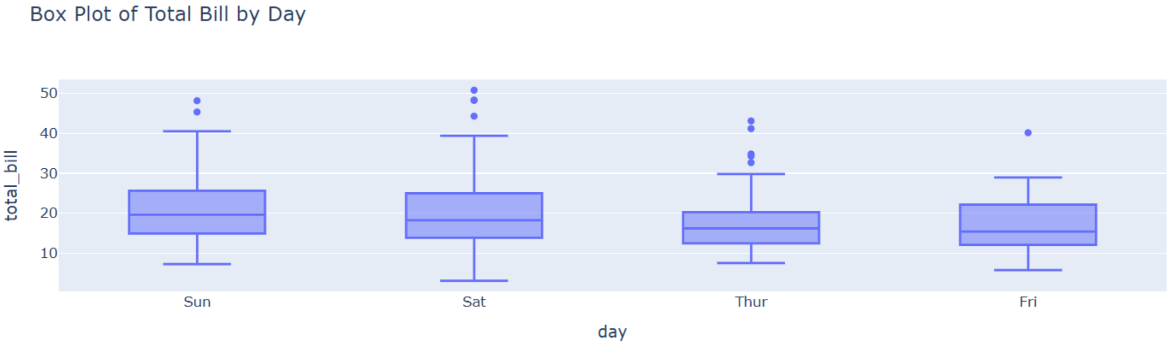
## **Explanation:**

- **px.scatter_3d():** Creates a 3D scatter plot. Here, we plot the columns sepal_length, sepal_width, and petal_length in 3D space, with points colored by species.

## 7. **Box Plot using Plotly Express**



```
import plotly.express as px

# Example dataset: tips dataset
df = px.data.tips()

# Create a box plot to show the distribution of total_bill by day
fig = px.box(df, x="day", y="total_bill", title="Box Plot of Total Bill by Day")

# Show the plot
fig.show()
```



## **Explanation**:

- **px.box()**: Creates a box plot to show the distribution of total_bill across the different days.