Python for Data Science

VirtualEnv

Creating a venv in Conda:

Conda create –name env_name

To activate: activate env_name

Install packages

To deactivate: deactivate env_name

NumPy

NumPy provides efficient handling of large arrays and matrices, along with a vast collection of mathematical functions to operate on them.

```
pip install numpy
```

1. Built-in Generation Methods

- np.array(): Create an array from a Python list or tuple.
- np.zeros(): Create an array filled with zeros.
- np.ones(): Create an array filled with ones.
- np.empty(): Create an array without initializing its elements.
- np.linspace(): Create an array with a specified number of values within a range.
- np.random.rand(): Create an array of random values between 0 and 1.
- np.random.randint(): Create an array of random integers within a specified range.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
zeros_arr = np.zeros((3, 3))  # Creates a 3x3 array filled with zeros
zeros_arr = np.zeros(3) #For singe row
ones_arr = np.ones((2, 2))  # Creates a 2x2 array filled with ones
empty_arr = np.empty((2, 2))  # Creates a 2x2 array with uninitialized elements
np.arange(): Create an array with regularly spaced values.
range_arr = np.arange(1, 10, 2)  # Creates an array with values from 1 to 9 (step of 2)
linspace_arr = np.linspace(0, 1, 5)  # Creates an array with 5 equally spaced values from 0 to 1
random_arr = np.random.rand(3, 3)  # Creates a 3x3 array with random values between 0 and 1
randint_arr = np.random.randint(1, 10, (2, 2))  # Creates a 2x2 array with random integers from 1 to 9
```

2. Attributes and Methods

- ndarray.shape: Dimensions of the array.
- ndarray.ndim: Number of dimensions (axes) of the array.
- ndarray.size: Number of elements in the array.
- ndarray.dtype: Data type of the array elements.
- ndarray.itemsize: Size in bytes of each array element.
- ndarray.reshape(): Change the shape of the array.
- ndarray.resize(): Modify the shape of the array in-place.
- ndarray.max(), ndarray.min(): Maximum and minimum values in the array.
- ndarray.mean(), ndarray.sum(): Mean and sum of array elements.
- ndarray.std(), ndarray.var(): Standard deviation and variance of array elements.

Example:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)  # Output: (2, 3)
print(arr.ndim)  # Output: 2
print(arr.size)  # Output: 6
arr = np.array([1, 2, 3])
print(arr.dtype)  # Output: int64
arr = np.array([1, 2, 3], dtype=np.float64)
print(arr.itemsize)  # Output: 8 (bytes)
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape((2, 3))  # Reshapes the array to a 2x3 matrix
arr.resize((2, 3))  # Resizes the array to a 2x3 matrix in-place
print(arr.max())  # Output: 6
print(arr.min())  # Output: 1
print(arr.mean())  # Output: 3.5
print(arr.sum())   # Output: 21
print(arr.std())  # Output: 1.7078
print(arr.var())  # Output: 2.9167
```

3. Indexing

- Access elements: Use square brackets and indices.
- Slicing: Extract a portion of the array using start:stop:step syntax.
- Fancy Indexing: Pass an array of indices to access specific elements.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr[0])  # Output: 1
print(arr[1:4])  # Output: [2, 3, 4]
indices = np.array([0, 2, 4])
print(arr[indices])  # Output: [1, 3, 5]
arr2 = arr.copy() # To make a copy
```

4. 2D Matrix

- Create a 2D array: np.array([[1, 2, 3],[4, 5, 6]]).
- Shape: Access the shape attribute (arr.shape) to get the dimensions.
- Indexing: Access elements using row and column indices.
- Slicing: Extract a portion of the matrix.
- Matrix operations: NumPy provides functions for matrix operations like np.dot(), np.transpose(), np.linalg.inv(), and more.

Example:

```
import numpy as np
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(matrix.shape)  # Output: (2, 3)
print(matrix[0, 1])  # Output: 2
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(matrix[:, 1:])  # Output: [[2, 3], [5, 6], [8, 9]]
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
product = np.dot(matrix1, matrix2)  # Matrix multiplication
transpose = np.transpose(matrix1)  # Transpose of the matrix
inverse = np.linalg.inv(matrix1)   # Inverse of the matrix
```

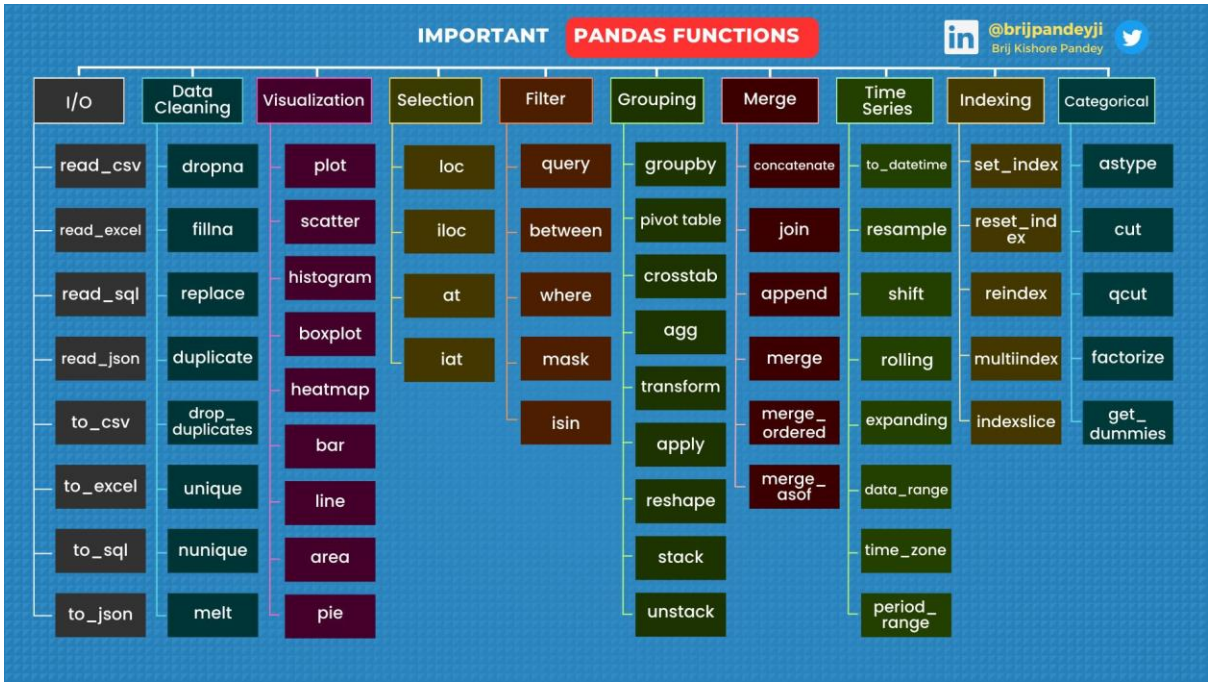
5. NumPy Operations

Element-wise arithmetic operations: +, -, *, /, **.
Aggregation functions: sum(), min(), max(), mean(), std(), etc.

Example:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
addition = arr1 + arr2
subtraction = arr1 - arr2
multiplication = arr1 * arr2
division = arr1 / arr2
exponentiation = arr1 ** arr2

arr = np.array([1, 2, 3, 4, 5])
total_sum = np.sum(arr)
minimum = np.min(arr)
maximum = np.max(arr)
average = np.mean(arr)
standard_deviation = np.std(arr)
```

Pandas



pandas is another popular Python library that is widely used for data manipulation and analysis. It provides data structures and functions to efficiently handle and manipulate structured data, such as tables and time series data.

The primary data structure in pandas is the DataFrame, which is a two-dimensional table with labeled columns and rows. It allows you to perform various operations like indexing, filtering, grouping, and aggregating data.

Series

A Series is a one-dimensional labeled array that can hold any data type. It's similar to a column in a spreadsheet or a dictionary. You can create a Series from a list, NumPy array, or a dictionary.

```
# Create a Series from a list
data = [10, 20, 30, 40, 50]
series = pd.Series(data)
print(series)
Output:
0  10
1  20
2  30
3  40
4  50
dtype: int64
```

DataFrames

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. It's like a table or a spreadsheet. You can create a DataFrame from a dictionary, a NumPy array, or by loading data from a file.

```
import pandas as pd
# Create a DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie', 'Dave'],
        'Age': [25, 30, 35, 40],
        'City': ['New York', 'London', 'Paris', 'Tokyo']}
df = pd.DataFrame(data)
print(df)
     Name  Age      City
0   Alice   25  New York
1     Bob   30    London
2  Charlie   35     Paris
3    Dave   40     Tokyo
```

Indexing

You can select specific rows or columns in a DataFrame using various indexing techniques, such as using column names, row labels, or conditional selection.

```
# Selecting columns
print(df['Name'])
print(df[['Name', 'City']])

# Selecting rows
print(df.loc[1])  # Select row by label. 1,2,3… are default labels
print(df.iloc[2])  # Select row by integer index

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

# Manually add labels using rename()
df = df.rename(index={0: 'Label1', 1: 'Label2', 2: 'Label3'}, columns={'A': 'Column1', 'B':
'Column2'})
print(df)
# Add labels using a list
label_list = ['Label1', 'Label2', 'Label3']
df.index = label_list

column_list = ['Column1', 'Column2']
df.columns = column_list

#Reset index, current index will become new column named index. New index starts from 0
df.reset_index(inplace = True)

#To set a column as index
df.set_index('B', inplace = True)
```

Conditional Selection

```
# Create a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'Dave'],
      'Age': [25, 30, 35, 40],
      'City': ['New York', 'London', 'Paris', 'Tokyo']}
df = pd.DataFrame(data)

# Conditional selection based on a single condition
mask = df['Age'] > 30  # Create a mask for values greater than 30
selected_rows = df[mask]  # Apply the mask to the DataFrame
print(selected_rows)

# Conditional selection with multiple conditions
mask = (df['Age'] > 30) & (df['City'] == 'Paris')
selected_rows = df[mask]
print(selected_rows)

mask = (df['Age'] > 30) & (df['City'].isin(['Paris', 'Tokyo'])) # for or use |
selected_rows = df[mask]
print(selected_rows)

# Select the Name column for rows where Age is greater than 30
selected_column = df.loc[df['Age'] > 30, 'Name']
# Select rows where Name starts with 'A'
mask = df['Name'].str.startswith('A')
selected_rows = df[mask]
print(selected_rows)
# Select rows where Age is greater than 30 using the query() method
selected_rows = df.query('Age > 30')
print(selected_rows)
# Select rows where Age is greater than 30 and City is 'Paris', and retrieve only the 'Name'
and 'Salary' columns
selected_rows = df[df['Age'] > 30].loc[df['City'] == 'Paris', ['Name', 'Salary']]
# Select rows where Salary is greater than or equal to 60000, sort them by Age in descending
order, and retrieve all columns
selected_rows = df[df['Salary'] >= 60000].sort_values(by='Age', ascending=False)
print(selected_rows)
```

Joins and Concatenation

pandas allows you to combine multiple DataFrames through joins and concatenation operations.

```
# Create two DataFrames
df1 = pd.DataFrame({'A': [1, 2, 3],
                'Key': ['K0', 'K1', 'K2']})

df2 = pd.DataFrame({'B': [4, 5, 6],
                'Key': ['K0', 'K1', 'K2']})
```

```
# Join the DataFrames based on the 'Key' column
joined_df = df1.join(df2.set_index('Key'), on='Key')
print(joined_df)

# Joining DataFrames based on a common column

df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                    'B': ['B0', 'B1', 'B2']})
df2 = pd.DataFrame({'A': ['A2', 'A3', 'A4'],
                    'C': ['C2', 'C3', 'C4']})
merged_df = pd.merge(df1, df2, on='A')
print(merged_df)



# Concatenating DataFrames vertically
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                    'B': ['B0', 'B1', 'B2']})
df2 = pd.DataFrame({'A': ['A3', 'A4', 'A5'],
                    'B': ['B3', 'B4', 'B5']})

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

Grouping and Aggregation

pandas allows you to group data based on certain criteria and perform aggregation operations on the grouped data.

```
# Grouping data by a column and computing the mean
grouped_df = df.groupby('City')['Age'].mean()
print(grouped_df)
```

Merging DataFrames

Merging DataFrames is similar to joining, but it allows you to merge based on different criteria (not just a single common column).

```
# Merging DataFrames based on multiple columns
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                    'B': ['B0', 'B1', 'B2'],
                    'Key': ['K0', 'K1', 'K2']})
df2 = pd.DataFrame({'C': ['C0', 'C1', 'C2'],
                    'D': ['D0', 'D1', 'D2'],
                    'Key': ['K0', 'K1', 'K2']})

merged_df = pd.merge(df1, df2, on=['Key'])
print(merged_df)
```

MultiIndex

MultiIndex or hierarchical indexing allows you to have multiple levels of indexing in a DataFrame.

```
# Creating a DataFrame with MultiIndex
arrays = [['A', 'A', 'B', 'B'], [1, 2, 1, 2]]
index = pd.MultiIndex.from_arrays(arrays, names=('Letter', 'Number'))
df = pd.DataFrame({'Value': [10, 20, 30, 40]}, index=index)
print(df)
# Select rows with Letter='A' and Number=1
selected_rows = df.loc[('A', 1)]
# Select the 'Value' column for Letter='A'
selected_column = df['Value', 'A']
# Select the element with Letter='B' and Number=2
selected_element = df.at[('B', 2), 'Value']
```

Handling Missing Data

pandas provides methods to handle missing or null values in a DataFrame, such as dropping or filling missing values.

```
# Handling missing data
df = pd.DataFrame({'A': [1, 2, None, 4, 5],
                   'B': [10, None, 30, 40, 50]})
df.dropna()  # Drop rows with missing values
df.fillna(0)  # Fill missing values with 0
```

Operations on DataFrames

pandas supports various operations on DataFrames, such as arithmetic operations, statistical calculations, and applying functions to the data.

```
# Arithmetic operations
df1 = pd.DataFrame({'A': [1, 2, 3],
                    'B': [4, 5, 6]})
df2 = pd.DataFrame({'A': [7, 8, 9],
                    'B': [10, 11, 12]})
df1 + df2  # Element-wise addition

# Statistical calculations
df.mean()  # Compute column-wise mean
df.sum(axis=1)  # Compute row-wise sum

# Applying functions
df.apply(lambda x: x**2)  # Apply a function to each element
```

These are some of the key concepts and functionalities in pandas. By understanding and practicing these concepts, you'll be equipped to perform various data manipulation and analysis tasks using pandas.

Common DataFrame functions

Certainly! Here are some common DataFrame functions in pandas:

1. Reading and Writing Data:
    - Read data from a CSV file: **df = pd.read_csv('filename.csv')**
    - Read data from an Excel file: **df = pd.read_excel('filename.xlsx')**
    - Write DataFrame to a CSV file: **df.to_csv('filename.csv', index=False)**
    - Write DataFrame to an Excel file: **df.to_excel('filename.xlsx', index=False)**
2. Dropping Rows or Columns:
    - Drop rows by index: **df.drop([index1, index2])**
    - Drop columns by name: **df.drop(['column1', 'column2'], axis=1)**
3. Sorting:
    - Sort DataFrame by column: **df.sort_values(by='column_name')**
    - Sort DataFrame by multiple columns: **df.sort_values(by=['col1', 'col2'])**
    - Sort DataFrame by index: **df.sort_index()**
4. Pivoting:
    - Perform a simple pivot: **df.pivot(index='index_col', columns='column_col', values='value_col')**
    - Perform a pivot table with aggregation: **df.pivot_table(values='value_col', index='index_col', columns='column_col', aggfunc='mean')**
5. Handling Null Values:
    - Check for null values: **df.isnull()**
    - Drop rows with null values: **df.dropna()**
    - Fill null values with a specific value: **df.fillna(value)**
    - Replace null values with the mean of the column: **df.fillna(df.mean())**
6. Aggregation and Descriptive Statistics:
    - Compute the mean of each column: **df.mean()**
    - Compute the sum of each column: **df.sum()**
    - Compute descriptive statistics: **df.describe()**
7. Grouping and Aggregation:
    - Group data based on one or more columns: **grouped = df.groupby(['col1', 'col2'])**
    - Compute the mean within each group: **grouped.mean()**
    - Compute multiple aggregation functions: **grouped.agg(['mean', 'sum'])**
8. Data Visualization:
    - Plot a line chart: **df.plot.line()**
    - Plot a bar chart: **df.plot.bar()**
    - Plot a scatter plot: **df.plot.scatter(x='col1', y='col2')**
    - Plot a histogram: **df.hist()**

Matplotlib

Matplotlib is a popular plotting library in Python that provides a wide range of functions and tools for creating various types of visualizations. It is widely used for data visualization and is often used in conjunction with NumPy and Pandas.

%matplotlib.inline : To view visuals inline in Jupyter

Importing Matplotlib

Importing the Matplotlib library.

```
import matplotlib.pyplot as plt
```

Basic Plotting

Creating line plots, scatter plots, bar plots, and histograms.

```
plt.plot(x, y)
plt.scatter(x, y)
plt.bar(x, y)
plt.hist(data, bins=10)
plt.show() #Always needed to display the plot
```

Customizing Plots

Adding labels, title, legends, custom colors, styles, and setting axis limits.

```
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Plot Title')
plt.legend(['Line 1', 'Line 2'])
plt.plot(x, y, color='red', linestyle='dashed', marker='o')
plt.xlim(0, 10)
plt.ylim(0, 100)
```

Subplots

Creating multiple plots on the same figure.

```
plt.subplot(rows, columns, index)
plt.plot(x1, y1)
plt.subplot(rows, columns, index)
plt.plot(x2, y2)
```

Figure Size and Resolution

Adjusting the figure size and resolution.

```
# Create a figure with a specific size and resolution
plt.figure(figsize=(8, 6), dpi=80)
# Plotting commands
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Plot Title')
plt.show()
```

Saving Plots

Saving plots as image files.

```
plt.savefig('plot.png')
```

Annotations

Adding text annotations to the plot.

```
plt.annotate('Text', xy=(x, y), xytext=(x, y), arrowprops=dict(facecolor='black'))
```

Matplotlib Styles

Using predefined or custom styles for plots.

```
plt.style.use('ggplot')
```

Multiple Plots on the Same Figure

Creating multiple plots on the same figure using subplots.

```
plt.subplot(rows, columns, index)
plt.plot(x1, y1)
plt.subplot(rows, columns, index)
plt.plot(x2, y2)
```

Plot Annotations

Adding annotations with arrows to highlight specific points.

```
plt.annotate('Text', xy=(x, y), xytext=(x, y), arrowprops=dict(facecolor='black'))
```

Adding Gridlines

Displaying gridlines on the plot.

```
plt.grid(True)
```

Legends

Adding legends to distinguish different elements in the plot.

```
plt.legend(['Line 1', 'Line 2'])
```

Colormaps

Applying colormaps to visualize data in scatter plots.

```
plt.scatter(x, y, c=z, cmap='viridis')
```

Logarithmic Scale

Using logarithmic scale for the x-axis and y-axis.

```
plt.xscale('log')
plt.yscale('log')
```

Error Bars

Adding error bars to the plot.

```
plt.errorbar(x, y, yerr=error_values)
```

Subplots with Shared Axes

Creating subplots that share the same x-axis or y-axis.

```
fig, axes = plt.subplots(nrows, ncols, sharex=True, sharey=True)
axes[0].plot(x1, y1)
axes[1].plot(x2, y2)
```

Adding Text

Adding text to the plot.

```
plt.text(x, y, 'Text')
```

3D Plotting

Creating 3D plots for visualizing three-dimensional data.

```
from mpl_toolkits import mplot3d
ax = plt.axes(projection='3d')
ax.plot3D(x, y, z)
```

Working with Dates

Formatting and locating date values on the x-axis.

```
import matplotlib.dates as mdates
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
plt.gca().xaxis.set_major_locator(mdates.DayLocator())
```

Interactive Plots with Widgets

Creating interactive plots with sliders and widgets.

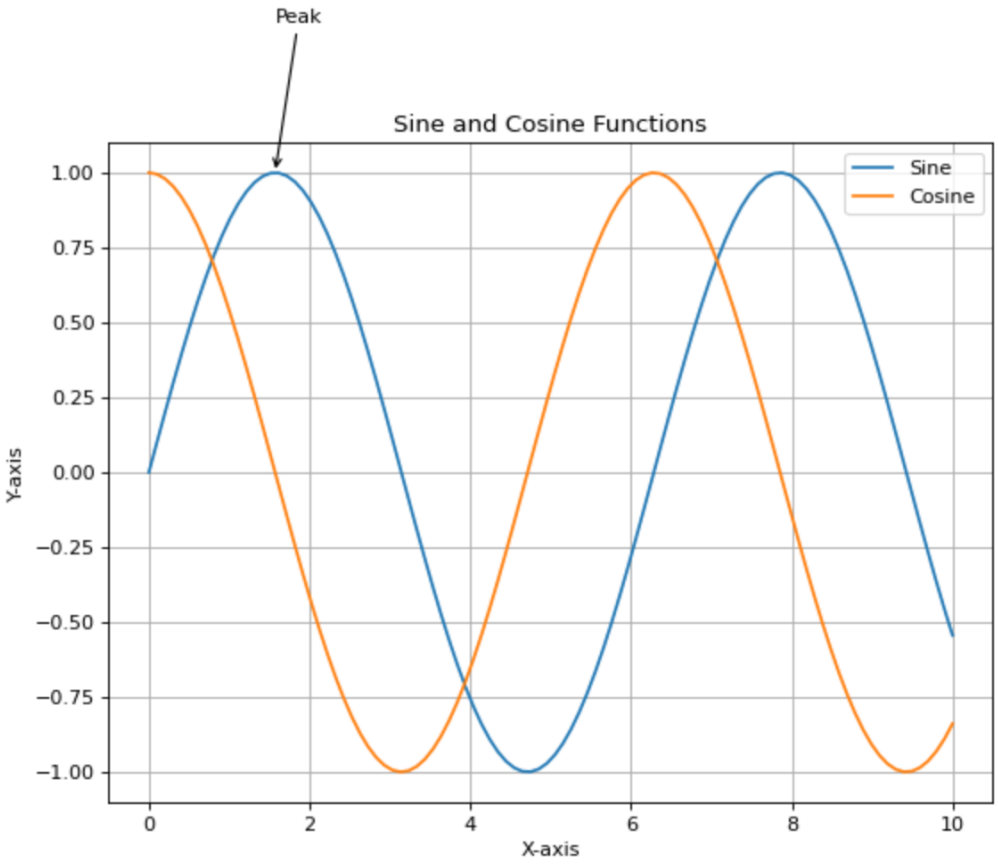
```
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider
fig, ax = plt.subplots()
plt.subplots_adjust(left=0.25, bottom=0.25)

# Create a slider
ax_slider = plt.axes([0.25, 0.1, 0.65, 0.03])
slider = Slider(ax_slider, 'Slider Label', min_value, max_value, initial_value)

def update(val):
   # Update plot based on slider value
   # Example: ax.plot(x, y * slider.val)
   pass

slider.on_changed(update)

plt.show()
```

Example

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

# Generate data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create a figure with a specific size and resolution
plt.figure(figsize=(8, 6), dpi=80)
```

```
# Plotting line plots
plt.plot(x, y1, label='Sine')
plt.plot(x, y2, label='Cosine')

# Customize the plot
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Sine and Cosine Functions')
plt.grid(True)
plt.legend()

# Add annotations
plt.annotate('Peak', xy=(np.pi/2, 1), xytext=(np.pi/2, 1.5),
        arrowprops=dict(facecolor='black', arrowstyle='->'))

# Save the plot as an image file
plt.savefig('plot.png')

# Display the plot
plt.show()
```

Output:



Seaborn

Seaborn is a powerful Python data visualization library built on top of Matplotlib. It provides a high-level interface for creating beautiful and informative statistical graphics.

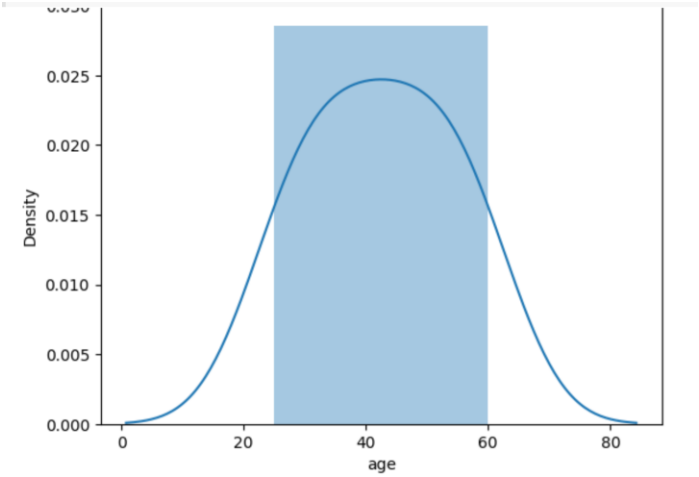
Distribution Plots

distplot

The distplot function combines a histogram with a kernel density estimate (KDE) plot to show the distribution of a dataset. It can also display a rug plot, which shows individual observations along the x-axis.

For example, to create a distplot of a dataset's age column, you could use the following code:

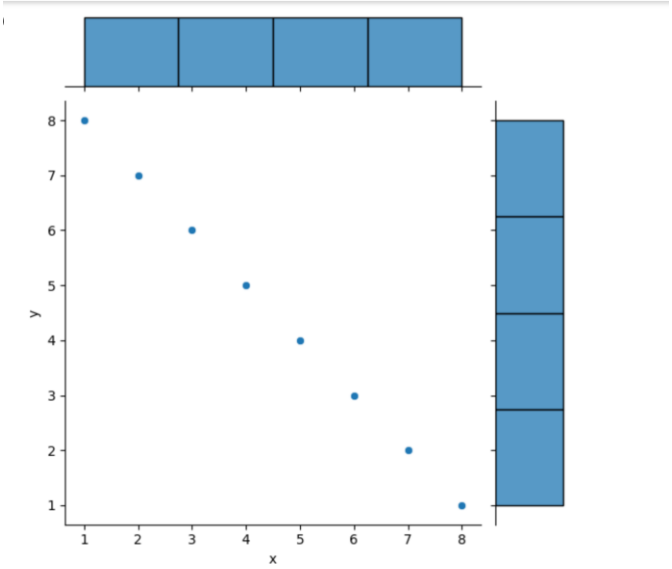
```
import seaborn as sns
sns.distplot(data['age'])
```



Jointplot

The jointplot function creates a multi-panel figure that shows the relationship between two variables, as well as their individual distributions. It can display scatter plots, hexbin plots, KDE plots, and regression plots.

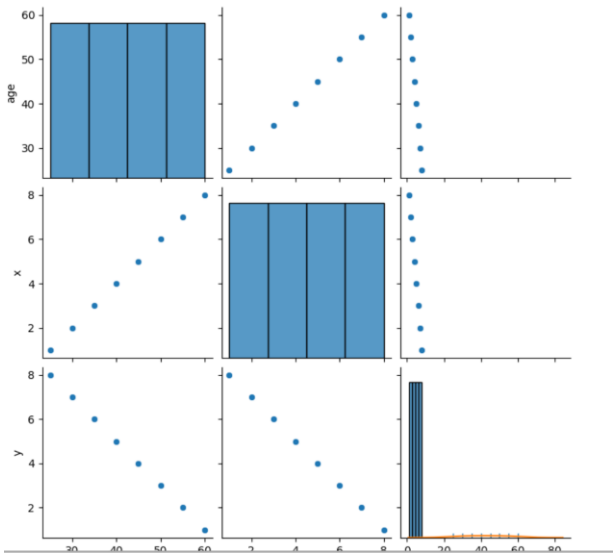
```
sns.jointplot(x='x', y='y', data=data)
```



Pairplot

The pairplot function creates a matrix of plots showing the pairwise relationships between multiple variables in a dataset. It can display scatter plots, KDE plots, and regression plots.

```
sns.pairplot(data)
```



Rugplot

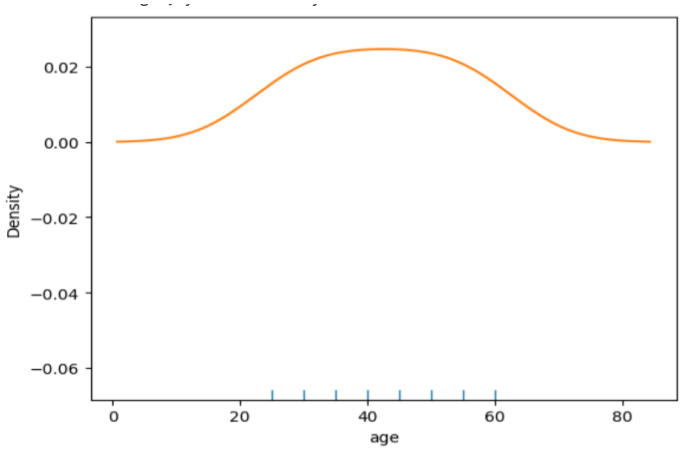
The rugplot function creates a plot that shows individual observations along one axis. It can be useful for visualizing the distribution of a small dataset or for displaying multiple distributions on the same axis.

```
sns.rugplot(data['age'])
```

kdeplot

The kdeplot function creates a plot that shows the kernel density estimate (KDE) of a dataset. It can be useful for visualizing the distribution of a continuous variable or for comparing multiple distributions.

```
sns.kdeplot(data['age'])
```



Categorical Plot

Data:

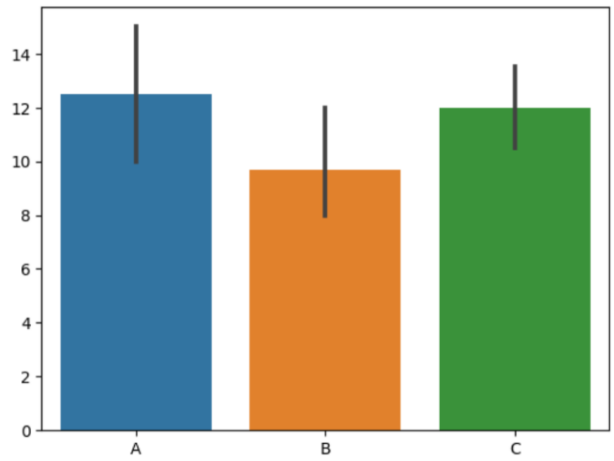
```
data = {
   'category': ['A', 'A', 'B', 'B', 'B', 'C', 'C', 'C', 'C'],
```

```
    'numerical_variable': [10, 15, 8, 12, 9, 11, 13, 14, 10]
}
```

Bar Plot

Displays the average value of a numerical variable for each category.

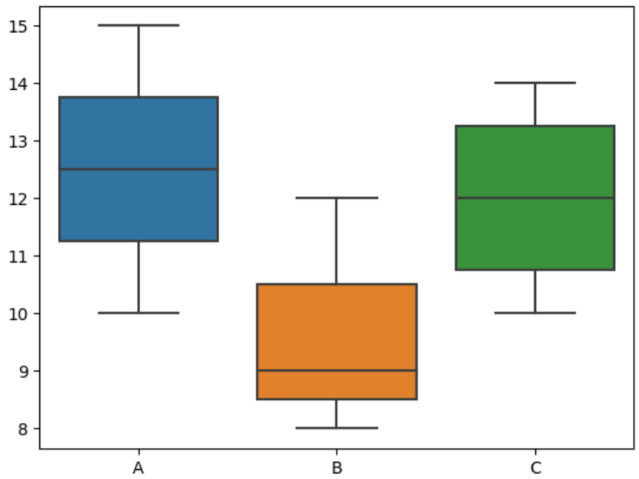
```
sns.barplot(x='category', y='numerical_variable', data=data)
```



Box Plot

Summarizes the distribution of a numerical variable for different categories.

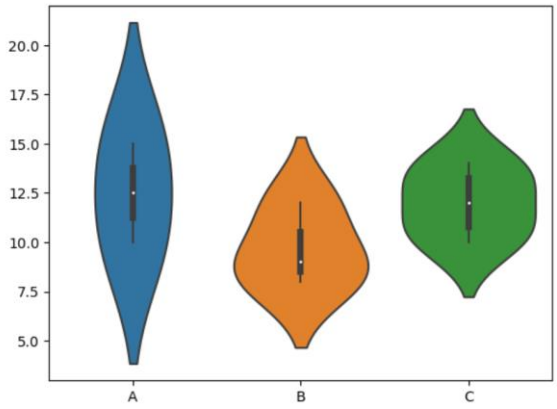
```
sns.boxplot(x='category', y='numerical_variable', data=data)
```



Violin Plot

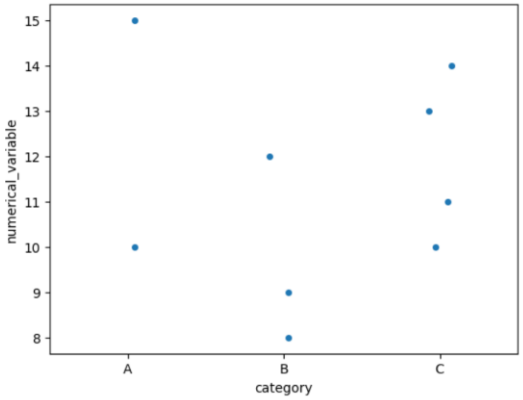
Combines a box plot with a kernel density plot to show the distribution of a numerical variable for each category.

```
sns.violinplot(x='category', y='numerical_variable', data=data)
```



Strip Plot

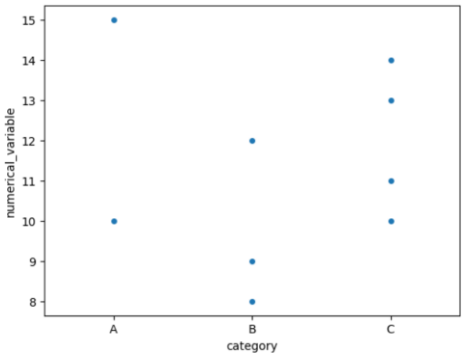
```
sns.stripplot(x='category', y='numerical_variable', data=data)
```



Swarm Plot

Similar to a strip plot but adjusts the positions of data points to prevent overlapping.

```
sns.swarmplot(x='category', y='numerical_variable', data=data)
```



Factor Plot (catplot)

A general plot type that can create various categorical plots based on the specified kind parameter.

```
sns.catplot(x='category', y='numerical_variable', kind='bar', data=data)
```

Matrix Plots

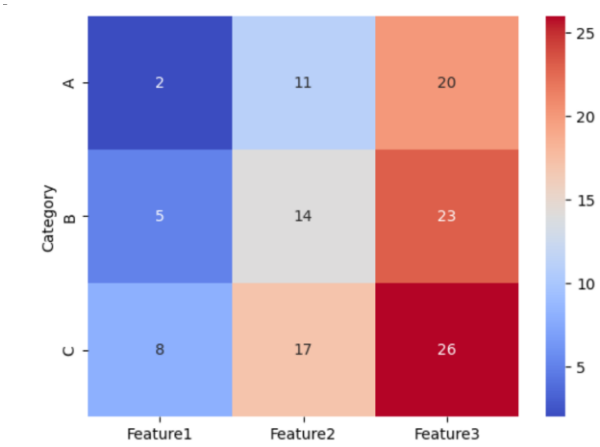
```
data = {
   'Category': ['A', 'A', 'A', 'B', 'B', 'B', 'C', 'C', 'C'],
```

```
    'Feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9],
    'Feature2': [10, 11, 12, 13, 14, 15, 16, 17, 18],
    'Feature3': [19, 20, 21, 22, 23, 24, 25, 26, 27]
}
df = pd.DataFrame(data)
# Create a pivot table
pivot_table = df.pivot_table(index='Category', aggfunc='mean')
```

HeatMaps

A heatmap represents the data as a color-encoded matrix, where the values in the matrix are represented by different colors.

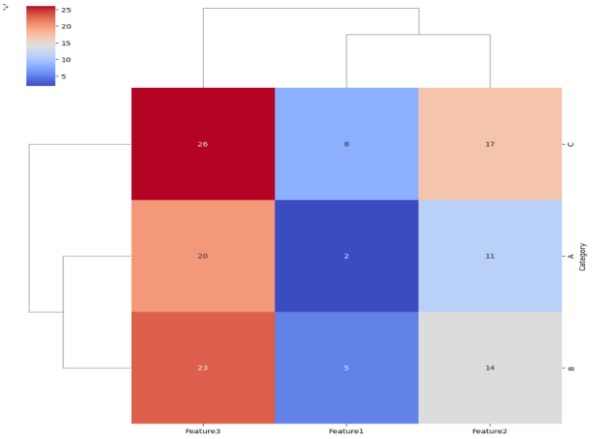
```
sns.heatmap(data, annot=True, cmap='coolwarm')
```



ClusterMaps

A clustermap combines a heatmap with hierarchical clustering, which groups similar variables or samples together.

```
sns.clustermap(data, cmap='coolwarm')
```

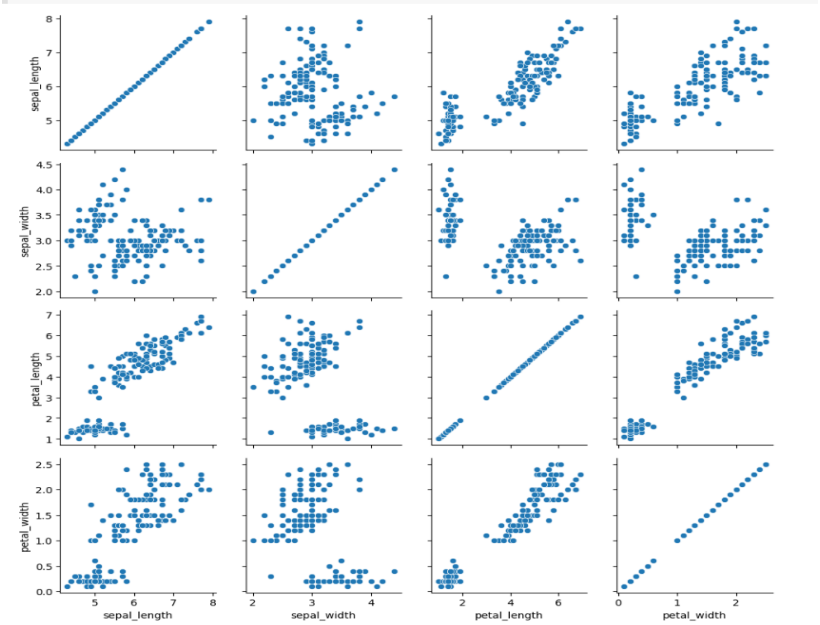


Grid

PairGrid

```
import seaborn as sns
import matplotlib.pyplot as plt
# Load the iris dataset
```

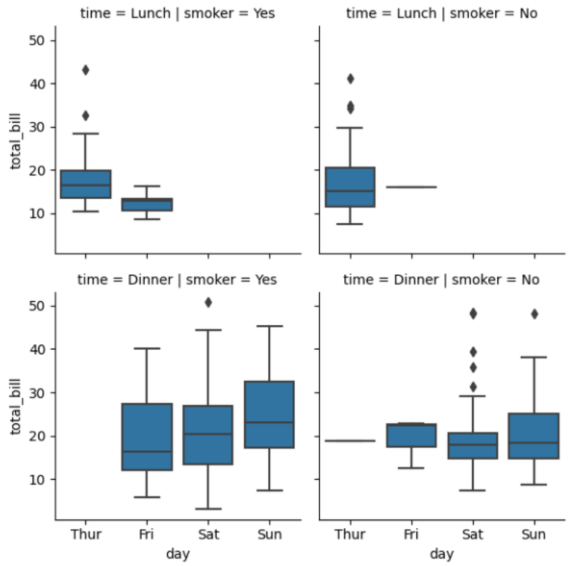
```
iris = sns.load_dataset("iris")
# Create a PairGrid
grid = sns.PairGrid(data=iris)

# Map scatter plots onto the grid
grid.map(sns.scatterplot)
plt.show()
```



Customizing PairGrid: You can customize the plots in a PairGrid using various Seaborn plotting functions. For instance, you can use map_upper() to specify a plot type for the upper triangle of the grid and map_lower() for the lower triangle. Additionally, you can use map_diag() to define the plot type for the diagonal subplots.

FacetGrid

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

# Load the tips dataset
tips = sns.load_dataset("tips")

# Create a FacetGrid
grid = sns.FacetGrid(data=tips, row="time", col="smoker")

# Map box plots onto the grid
grid.map(sns.boxplot, "day", "total_bill")
plt.show()
```



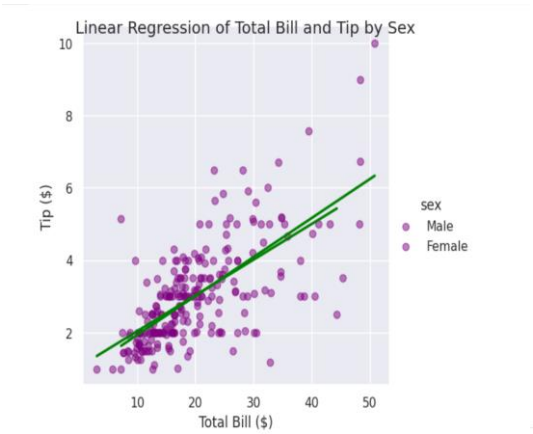
You can customize the plots in a FacetGrid using various Seaborn plotting functions. For instance, you can use map_dataframe() to apply a custom function to each subset of the data. Additionally, you can use set() to modify the aesthetics and style of the grid.

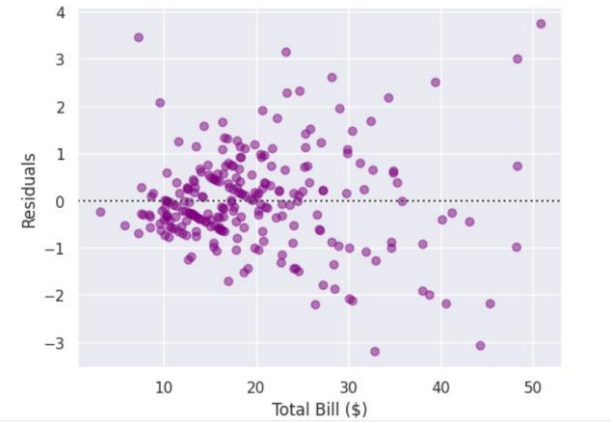
Regression Plots

Linear Model Plot: Seaborn's lmplot() function is commonly used to create linear model plots. It combines a scatter plot of the data points with a regression line, showing the overall trend and any deviations from it.

The lmplot() function creates a linear model plot, while the residplot() function creates a residuals plot. lmplot() provides various customization options to enhance the visual representation. For example, you can add hue, col, or row parameters to create separate linear model plots for different subsets of the data. You can also use the order parameter to fit polynomial regression models.

```
import seaborn as sns
import matplotlib.pyplot as plt
# Load the tips dataset
tips = sns.load_dataset("tips")
# Create a linear model plot with residuals and hue
sns.set(style="darkgrid")
lm_plot = sns.lmplot(x="total_bill", y="tip", hue="sex", data=tips,
                     scatter_kws={"color": "purple", "alpha": 0.5},
                     line_kws={"color": "green"}, ci=None)
# Add residuals plot with hue
res_plot = sns.residplot(x="total_bill", y="tip", hue="sex", data=tips,
                         scatter_kws={"color": "purple", "alpha": 0.5},
                         line_kws={"color": "green"})

# Customize the plots
lm_plot.set_axis_labels("Total Bill ($)", "Tip ($)")
lm_plot.fig.suptitle("Linear Regression of Total Bill and Tip by Sex")
res_plot.set(xlabel="Total Bill ($)", ylabel="Residuals")
plt.show()
```





RegPlot

The regplot() function in Seaborn is used to create a scatter plot with a fitted regression line. It combines the functionality of both scatter plots and regression plots in a single function. Just replace lmplot with regplot in above examples.

Customizations

In Seaborn, you can customize the style, size, and color of your plots using various functions and parameters. Here's a breakdown of how you can achieve these customizations:

1. Style: Seaborn provides different built-in styles to change the overall appearance of your plots. You can set the style using the **set_style()** function. Some commonly used styles include:
    - **"darkgrid"**: A dark background grid with white gridlines (default).
    - **"whitegrid"**: A white background grid with gray gridlines.
    - **"dark"**: A dark background without gridlines.
    - **"white"**: A plain white background without gridlines.
    - **"ticks"**: A white background with ticks along the axes.

    For example, to set the style to **"whitegrid"**, you can use: **sns.set_style("whitegrid")**.
2. Size: You can control the size of your plots by adjusting the figure size and other plot-specific parameters. The figure size can be set using the **plt.figure(figsize=(width, height))** function from Matplotlib. For example, to set the figure size to 10 inches wide and 6 inches high, you can use: **plt.figure(figsize=(10, 6))**.

Additionally, many Seaborn plotting functions provide size-related parameters that allow you to adjust the size of specific plot elements. For instance, **scatter_kws={"s": size}** can be used to specify the size of scatter points in a scatter plot.

3. Color: Seaborn allows you to customize the color palette used in your plots. You can set the color palette using the **set_palette()** function. Seaborn offers several built-in color palettes, such as **"deep"**, **"muted"**, **"bright"**, **"pastel"**, and more. Additionally, you can use custom color palettes or specify individual colors as a list.

For example, to set the color palette to **"deep"**, you can use: **sns.set_palette("deep")**.

Moreover, you can pass a list of colors to specific Seaborn functions using the **palette** parameter to override the default color scheme.

Data Visualizations in Pandas(BuiltIn)

Pandas provides convenient methods to create different types of plots. Here's a summary of the available options for data visualization in Pandas:

1. Histogram (hist): To create a histogram of a column, use the hist method. It plots the frequency distribution of numeric data.

```
df['column'].hist()
```

2. Bar plot (bar): To create a vertical bar plot, use the plot method with kind='bar'.

```
df.plot(kind='bar', x='x_column', y='y_column')
#Plot bars of all columns(stacked optional)
df.plot.bar(stacked=True)
```

3. Line plot (line): To create a line plot, use the plot method with kind='line'.

```
df.plot(kind='line', x='x_column', y='y_column')

df.plot.line( x='x_column', y='y_column')
```

4. Area Plot: Areaplot of all columns of df

```
df.plot.area()
```

5. Scatter plot (scatter): To create a scatter plot, use the plot method with kind='scatter'.

```
df.plot(kind='scatter', x='x_column', y='y_column')

df.plot.scatter(x='x_column', y='y_column')
```

6. Box plot (box): To create a box plot, use the plot method with kind='box'.

```
df.plot(kind='box')

df.plot.box()
```

7. Hexbin plot (hexbin): To create a hexbin plot, use the plot method with kind='hexbin'.

```
df.plot(kind='hexbin', x='x_column', y='y_column', gridsize=10)

df.plot.hexbin(x='x_column', y='y_column', gridsize=10)
```

8. Density plot (density or kde): To create a kernel density estimation plot, use the plot method with kind='density' or kind='kde'.

```
df['column'].plot(kind='density')
# or
df['column'].plot(kind='kde')

df['column'].plot.density()
# or
df['column'].plot.kde()
```