Introduction to NumPy

What is NumPy?

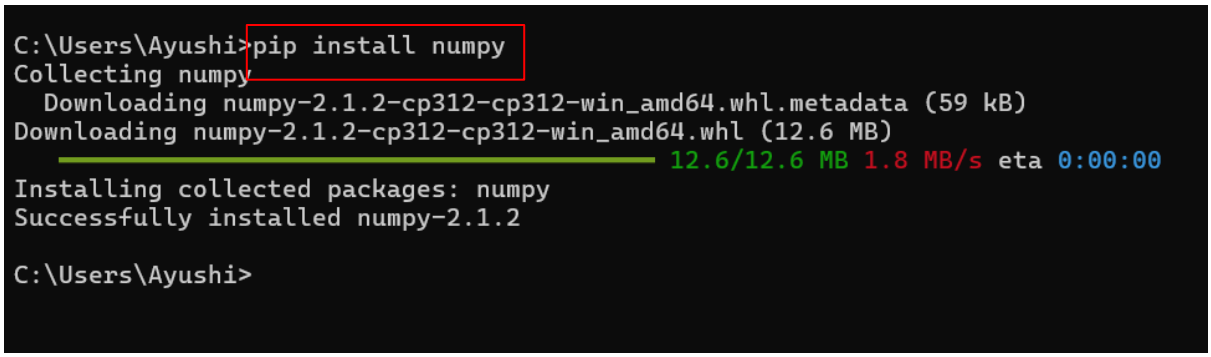
NumPy (Numerical Python) is a powerful library for numerical computing in Python. It provides support for arrays, matrices, and a wide variety of mathematical functions to operate on these data structures. NumPy is particularly useful for handling large datasets, performing mathematical operations efficiently, and is a foundation for many other libraries such as Pandas and SciPy.

Why is NumPy Useful?

- **Efficient Computation**: NumPy uses highly optimized C code, making array operations faster than Python's standard lists.
- **Multi-Dimensional Arrays**: NumPy supports multi-dimensional arrays, which are not natively available in Python.
- **Convenience**: It simplifies performing mathematical operations like linear algebra, random sampling, and Fourier transforms on large datasets.

Installation

To install NumPy, use the following pip command in your terminal or command prompt:



Creating Arrays with Specific Data Types

You can define the data type of the array elements using the dtype parameter:

Importing NumPy

After installation, you can import NumPy into your Python environment:

```
import numpy as np
```

np is a widely used alias for NumPy.

2. Basics of NumPy Arrays

Creating Arrays

NumPy arrays are similar to Python lists but are more efficient for numerical data. Here's how to create different types of arrays:

- **1D Array**:

```
arr_1d = np.array([1, 2, 3, 4, 5])
print(arr_1d)
```

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

- **2D Array**:

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

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

- **Multi-dimensional Array**:

```
arr_multi = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr_multi)
```

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

 [[ 7  8  9]
  [10 11 12]]]
```

- **Creating Arrays with Specific Data Types**

  You can define the data type of the array elements using the `dtype` parameter:

```
arr_float = np.array([1, 2, 3, 4], dtype=float)
print(arr_float)
```

```
[1. 2. 3. 4.]
```

Array Initialization

In addition to creating arrays using np.array(), NumPy provides various functions to initialize arrays with specific values or patterns.

- **Creating an Array of Zeros**: You can create an array filled with zeros using np.zeros(). Specify the shape of the array as an argument:

```
zeros_array = np.zeros((3, 4))  # 3x4 array of zeros
print(zeros_array)
```

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

Creating an Array of Ones: Similarly, np.ones() creates an array filled with ones:

```
ones_array = np.ones((2, 3))  # 2x3 array of ones
print(ones_array)
```

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

Creating an Empty Array: np.empty() initializes an array without setting its values to any particular number. It simply allocates the space, and the contents are whatever values were already in the allocated memory:

```
empty_array = np.empty((2, 2))  # Uninitialized 2x2 array
print(empty_array)
```

```
[[1. 2.]
 [3. 4.]]
```

Creating an Array with a Specific Value: Use np.full() to create an array where all elements are set to a specific value:
Python

```
full_array = np.full((3, 3), 7)  # 3x3 array of sevens
print(full_array)
```

```
[[7 7 7]
 [7 7 7]
 [7 7 7]]
```

Creating Sequences

NumPy provides convenient functions for generating arrays with sequences of numbers.

- **Using np.arange()**: This function works similarly to Python's built-in range(), creating evenly spaced values within a given range:

```
arr_range = np.arange(0, 10, 2)  # Array of values from 0 to 10 with a step of 2
print(arr_range)
```

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

- **Using np.linspace()**: This function generates a specified number of evenly spaced values between two points:

```
arr_linspace = np.linspace(0, 1, 5)  # 5 evenly spaced values between 0 and 1
print(arr_linspace)
```

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

Generating Random Numbers

The np.random module in NumPy allows you to generate arrays with random numbers.

```
random_integers = np.random.randint(1, 10, size=(3, 3))  # 3x3 array of random integ
print(random_integers)
```

```
[[4 5 8]
 [7 4 2]
 [4 4 9]]
```

- **Random Integers**: Generate an array of random integers within a specific range using np.random.randint():

- **Random Floats**: Use np.random.random() to generate random floating-point numbers between 0 and 1:

```
random_floats = np.random.random((2, 2))  # 2x2 array of random floats between 0 and
print(random_floats)
```

```
[[0.5602786  0.49699494]
 [0.32240045 0.31160743]]
```

Random Numbers from a Normal Distribution: Generate random numbers from a standard normal distribution (mean = 0, standard deviation = 1) using np.random.randn()

```
random_normal = np.random.randn(3, 3)  # 3x3 array of normally distributed random nu
print(random_normal)
```

```
[[-0.25985353  0.57734356 -0.21156318]
 [-1.6326923  -0.94040241 -1.47292374]
 [-1.29319263 -0.04929461  0.46873123]]
```

Array Operations

Basic Arithmetic

NumPy allows for element-wise operations, which means operations are performed between corresponding elements of arrays, and it also supports operations with scalars.

- **Element-wise Operations**: These operations apply element by element between two arrays of the same shape:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

addition = arr1 + arr2  # [5, 7, 9]
subtraction = arr1 - arr2  # [-3, -3, -3]
multiplication = arr1 * arr2  # [4, 10, 18]
division = arr1 / arr2  # [0.25, 0.4, 0.5]
print(addition, subtraction, multiplication, division)
```

- **Scalar Operations**: You can perform operations between an array and a single number (scalar):

```
arr = np.array([1, 2, 3])

scalar_add = arr + 5  # [6, 7, 8]
scalar_multiply = arr * 2  # [2, 4, 6]
print(scalar_add, scalar_multiply)
```

```
[6 7 8] [2 4 6]
```

- **Array Broadcasting**

- Broadcasting allows NumPy to perform element-wise operations on arrays of different shapes, extending smaller arrays to match the shape of larger ones under certain rules.

- **Broadcasting Rules**

- If the arrays have different dimensions, NumPy automatically pads the smaller array's shape with ones on the left.
- NumPy compares the arrays element-wise. If dimensions are equal or one of them is 1, the operation proceeds.
- If shapes are not compatible, an error is raised.
- Example:

```
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr2 = np.array([1, 2, 3])

broadcasted_addition = arr1 + arr2  # arr2 is broadcasted across rows of arr1
print(broadcasted_addition)
```

```
[[2 4 6]
 [5 7 9]]
```

Mathematical Functions

NumPy provides a wide range of mathematical functions for arrays:

- **Common Functions**:

```
arr = np.array([1, 2, 3, 4, 5])S

total_sum = np.sum(arr)  # Sum of all elements
mean_value = np.mean(arr)  # Mean of the elements
std_deviation = np.std(arr)  # Standard deviation
minimum_value = np.min(arr)  # Minimum value
maximum_value = np.max(arr)  # Maximum value
square_root = np.sqrt(arr)  # Square root of each element

print(total_sum, mean_value, std_deviation, minimum_value, maximum_value, square_roc
```

```
15 3.0 1.4142135623730951 1 5 [1.         1.41421356 1.73205081 2.         2.236067
98]
```

Trigonometric Functions: NumPy also supports trigonometric functions like sin(), cos(), tan(), and their inverse functions.

```
angles = np.array([0, np.pi/2, np.pi])

sin_values = np.sin(angles)  # Sine of each angle
cos_values = np.cos(angles)  # Cosine of each angle

print(sin_values, cos_values)
```

```
[0.0000000e+00 1.0000000e+00 1.2246468e-16] [ 1.000000e+00  6.123234e-17 -1.000000e
+00]
```

Indexing, Slicing, and Iterating

Indexing

NumPy arrays support indexing similar to Python lists. You can access individual elements by specifying their position.

- **Accessing Elements**:

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

element = arr[2]  # Access the element at index 2 (value: 3)
print(element)
```

```
3
```

Conditional Indexing: You can select elements that satisfy a specific condition.

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

condition = arr[arr > 3]  # Select elements greater than 3
print(condition)
```

```
[4 5]
```

Slicing

Slicing allows you to extract subarrays from a NumPy array. The syntax for slicing is similar to that of lists.

- **Basic Slicing**:

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

sliced_array = arr[1:4]  # Extract elements from index 1 to 3 (excludes index 4)
print(sliced_array)
```

```
[2 3 4]
```

Slicing 2D Arrays: Slicing can be extended to multi-dimensional arrays.

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

sliced_2d = arr_2d[1:, :2]  # Extract rows 1 to the end, and columns 0 to 1
print(sliced_2d)
```

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

Iterating Over Arrays

You can iterate over NumPy arrays using loops.

- **1D Array Iteration**:

```
arr = np.array([1, 2, 3])

for element in arr:
    print(element)
```

```
1
2
3
```

2D Array Iteration: In a multi-dimensional array, you can iterate over each subarray or flatten it to iterate over individual elements

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

for row in arr_2d:
    print(row)  # Iterates over rows

for element in arr_2d.flat:
    print(element)  # Iterates over individual elements
```

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

Reshaping and Manipulating Arrays

Reshaping Arrays

NumPy provides several functions to change the shape and structure of arrays without modifying the data.

- **np.reshape()**: Reshapes an array without changing its data.

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

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

np.ravel(): Flattens a multi-dimensional array into a 1D array. It returns a flattened view (shallow copy) of the original array when possible.

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

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

np.flatten(): Similar to ravel(), but returns a flattened copy (deep copy) of the array

```
flattened_copy = arr_2d.flatten()
print(flattened_copy)
```

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

np.transpose(): Transposes the axes of an array (i.e., switches rows and columns for a 2D array).

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

concatenated_arr = np.concatenate((arr1, arr2))
print(concatenated_arr)
```

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

Array Concatenation and Splitting

NumPy allows you to combine arrays or split them into multiple arrays.

- **np.concatenate()**: Concatenates two or more arrays along an existing axis.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

concatenated_arr = np.concatenate((arr1, arr2))
print(concatenated_arr)
```

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

np.stack(): Stacks arrays along a new axis (like creating a higher dimension).

```
stacked_arr = np.stack((arr1, arr2))  # Stacks arrays vertically
print(stacked_arr)
```

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

np.hstack() and **np.vstack()**: Horizontally and vertically stack arrays, respectively.

```
horizontal_stack = np.hstack((arr1, arr2))  # Horizontally stacks arrays
vertical_stack = np.vstack((arr1, arr2))  # Vertically stacks arrays
print(horizontal_stack, vertical_stack)
```

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

np.split(): Splits an array into multiple sub-arrays.

```
arr = np.array([1, 2, 3, 4, 5, 6])
split_arr = np.split(arr, 3)  # Splits array into 3 equal sub-arrays
print(split_arr)
```

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

Array Copying

There are two types of copying in NumPy: shallow copy (view) and deep copy.

- **Shallow Copy (view())**: A shallow copy creates a new array object, but the data is shared between the original and the copy. Modifying the view will affect the original array.

```
arr = np.array([1, 2, 3])
view_arr = arr.view()  # Create a shallow copy
view_arr[0] = 10  # Modifying view_arr changes arr
print(arr, view_arr)
```

```
[10  2  3] [10  2  3]
```

Deep Copy (copy()): A deep copy creates a completely independent array, with its own data.

```
arr = np.array([1, 2, 3])
deep_copy_arr = arr.copy()  # Create a deep copy
deep_copy_arr[0] = 10  # Does not affect arr
print(arr, deep_copy_arr)
```

```
[1 2 3] [10  2  3]
```

Advanced Array Operations

Linear Algebra with NumPy

NumPy includes a set of functions for performing linear algebra operations.

- **Matrix Multiplication (np.dot() and @ operator)**: Perform matrix multiplication between arrays.

```
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])

result = np.dot(matrix1, matrix2)  # Using np.dot()
result_at = matrix1 @ matrix2  # Using @ operator
print(result, result_at)
```

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

Inverse of a Matrix (np.linalg.inv()): Computes the inverse of a square matrix.

```
inverse_matrix = np.linalg.inv(matrix)
print(inverse_matrix)
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

Determinant (np.linalg.det()): Calculates the determinant of a matrix.

```
matrix = np.array([[1, 2], [3, 4]])
determinant = np.linalg.det(matrix)
print(determinant)
```

```
-2.0000000000000004
```

Eigenvalues and Eigenvectors (np.linalg.eig()): Computes the eigenvalues and eigenvectors of a square matrix.

```
eigenvalues, eigenvectors = np.linalg.eig(matrix)
print(eigenvalues, eigenvectors)
```

```
[-0.37228132  5.37228132] [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

Broadcasting Mechanism

Broadcasting is a powerful tool that allows NumPy to perform operations on arrays of different shapes without copying data. Here's a more complex example:

- **Example**: Suppose you have a 3x3 matrix and want to add a 1D array (of size 3) to each row.

```
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
vector = np.array([1, 2, 3])

broadcast_result = matrix + vector  # Broadcasts the vector to each row
print(broadcast_result)
```

```
[[ 2  4  6]
 [ 5  7  9]
 [ 8 10 12]]
```

Vectorization

Vectorization in NumPy allows for efficient array operations without the need for explicit loops, improving performance. You can vectorize a function using np.vectorize():

- **Vectorizing a Function**:

```
def my_function(x):
    return x * 2

vectorized_function = np.vectorize(my_function)
arr = np.array([1, 2, 3])
result = vectorized_function(arr)
print(result)
```

```
[2 4 6]
```

Vectorized operations are typically faster than traditional loops, making them essential for performance optimization in numerical computing.

Working with Missing Data

Handling Missing Data

Handling missing data in arrays is crucial, especially when working with real-world datasets.

- **np.nan (Not a Number)**: Represents missing or undefined numerical data. It's a special floating-point value.

```
arr = np.array([1, 2, np.nan, 4, 5])
print(arr)
```

```
[ 1.  2. nan  4.  5.]
```

- **np.isnan()**: Checks for NaN values in the array. It returns a boolean array where True indicates the presence of NaN.

```
is_nan = np.isnan(arr)
print(is_nan)  # Output: [False False  True False False]
```

```
[False False  True False False]
```

- **np.nan_to_num()**: Replaces NaN values with a specified value (default is 0). It also handles infinite values.

```
clean_arr = np.nan_to_num(arr, nan=0.0)
print(clean_arr)  # Output: [1. 2. 0. 4. 5.]
```

```
[1. 2. 0. 4. 5.]
```

Performance Optimization

NumPy arrays are optimized for performance and memory efficiency, especially when compared to Python lists.

Efficiency Considerations:

Why NumPy is Faster than Python Lists:

1. **Memory Efficiency**: NumPy arrays are stored in contiguous memory blocks, making them more efficient than lists, which store objects in scattered memory locations

```
import sys
list_size = sys.getsizeof([1, 2, 3, 4, 5])
numpy_size = sys.getsizeof(np.array([1, 2, 3, 4, 5]))
print(list_size, numpy_size)  # Memory comparison
```

```
104 152
```

2. **Vectorization**: NumPy operations are performed on the entire array, avoiding the need for explicit loops.

3. **Memory Layout and np.memmap()**: np.memmap() is a way to access large arrays stored on disk as if they were in memory. It allows efficient access to large datasets without loading everything into memory.

```
fp = np.memmap('datafile.dat', dtype='float32', mode='w+', shape=(1000, 1000))
```

4. **np.nditer() for Optimized Iteration**() : np.nditer() provides an efficient way to iterate through arrays element by element, especially when working with multi-dimensional arrays.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in np.nditer(arr):
    print(x, end=' ')
```

```
1 2 3 4 5 6
```

Practical Applications

NumPy is widely used in various fields such as data science, numerical simulations, and data analysis.

Numerical Simulations:

```
# Example: Random walk simulation
steps = 1000
walk = np.cumsum(np.random.randn(steps))
```

NumPy is a powerful tool for running simple to complex simulations due to its fast computation capabilities.

Data Analysis with NumPy:

NumPy arrays provide the basis for fast and efficient data manipulation in data analysis tasks.

```
# Example: Basic data analysis with NumPy
data = np.array([1, 2, 3, 4, 5, 6])
mean = np.mean(data)
std_dev = np.std(data)
print(f"Mean: {mean}, Standard Deviation: {std_dev}")
```

```
Mean: 3.5, Standard Deviation: 1.707825127659933
```

Integration with Other Libraries:

NumPy works seamlessly with other data science and visualization libraries such as:

- **Pandas**: For advanced data manipulation and analysis.
- **Matplotlib**: For plotting and visualizing data.
- **SciPy**: For scientific computing tasks such as optimization and integration.

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

Advanced Topics

Custom Data Types:

NumPy allows creating custom or structured arrays with different types of fields (similar to a table with columns of different data types)

```
dtype = [('name', 'S10'), ('age', 'i4'), ('height', 'f4')]
data = np.array([('Alice', 25, 5.5), ('Bob', 30, 6.0)], dtype=dtype)
print(data)
```

```
[(b'Alice', 25, 5.5) (b'Bob', 30, 6. )]
```

Masked Arrays:

Masked arrays allow you to handle invalid or missing entries. These arrays mask certain elements and ignore them in operations.

```
masked_array = np.ma.array([1, 2, np.nan, 4], mask=[0, 0, 1, 0])
print(masked_array.mean())  # Ignores masked elements
```

```
2.3333333333333335
```

NumPy Extensions

- **numexpr**: A library that speeds up NumPy operations by evaluating expressions element-wise, especially useful for large arrays.

```
import numexpr as ne
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
result = ne.evaluate('a + b')
print(result)
```

```
[11 22 33 44]
```

- `bottleneck`: A library focused on speeding up NumPy array operations, particularly for dealing with missing data.

```
import bottleneck as bn
data = np.array([1, 2, np.nan, 4])
nanmean = bn.nanmean(data)
print(nanmean)  # Calculates mean, ignoring NaN values
```

```
2.3333333333333335
```

Common Trigonometric Functions

np.sin(x)

```
import numpy as np
x = np.array([0, np.pi/2, np.pi])
sin_values = np.sin(x)
print(sin_values)
```

```
[0.0000000e+00 1.0000000e+00 1.2246468e-16]
```

This function computes the sine of each element in x. The input should be in radians.

np.cos(x)

```
cos_values = np.cos(x)
print(cos_values)
```

```
[ 1.000000e+00  6.123234e-17 -1.000000e+00]
```

This function computes the cosine of each element in x.

np.tan(x)

```
tan_values = np.tan(x)
print(tan_values)
```

```
[ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

This function computes the tangent of each element in x.

Inverse Trigonometric Functions

np.arcsin(x)

```
x = np.array([0, 0.5, 1])
arcsin_values = np.arcsin(x)
print(arcsin_values)
```

```
[0.         0.52359878 1.57079633]
```

This function computes the inverse sine (arc sine) of each element in x, returning the result in radians.

np.arccos(x)

```
arccos_values = np.arccos(x)
print(arccos_values)
```

```
[1.57079633 1.04719755 0.
```

This function computes the inverse cosine (arc cosine) of each element in x.

np.arctan(x)

```
arctan_values = np.arctan(x)
print(arctan_values)
```

```
[0.         0.46364761 0.78539816]
```

This function computes the inverse tangent (arc tangent) of each element in x.

Hyperbolic Trigonometric Functions

np.sinh(x)

```
x = np.array([0, 1, 2])
sinh_values = np.sinh(x)
print(sinh_values)
```

```
[0.         1.17520119 3.62686041]
```

This function computes the hyperbolic sine of each element in x.

np.cosh(x)

```
cosh_values = np.cosh(x)
print(cosh_values)
```

```
[1.         1.54308063 3.76219569]
```

This function computes the hyperbolic cosine of each element in x.

np.tanh(x)

```
tanh_values = np.tanh(x)
print(tanh_values)
```

```
[0.         0.76159416 0.96402758]
```

This function computes the hyperbolic tangent of each element in x

Converting Between Degrees and Radians

np.deg2rad(x)

```
degrees = np.array([0, 90, 180])
radians = np.deg2rad(degrees)
print(radians)
```

```
[0.         1.57079633 3.14159265]
```

This function converts angles from degrees to radians

np.rad2deg(x)

```
radians = np.array([0, np.pi/2, np.pi])
degrees = np.rad2deg(radians)
print(degrees)
```

```
[  0.  90. 180.]
```

This function converts angles from radians to degrees

Introduction to Matplotlib

Matplotlib is a widely used plotting library in Python for creating static, animated, and interactive visualizations. It provides a high degree of control over plots and is often used for data visualization in scientific and engineering fields.

1. Basic Plotting with Matplotlib

1.1 Importing Matplotlib

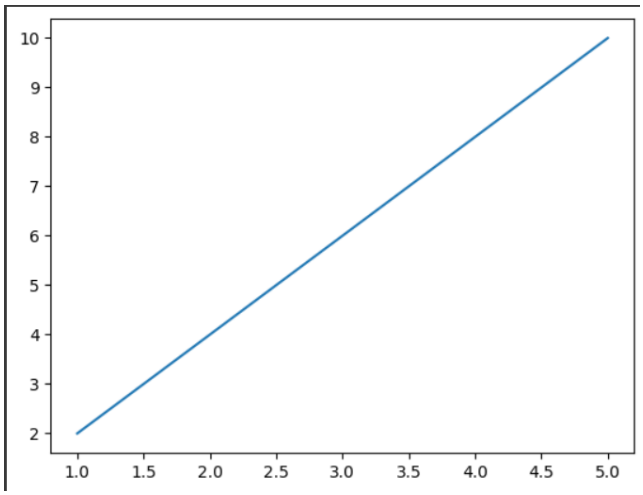
To start using Matplotlib, the library must be imported. It is commonly imported under the alias plt.

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

1.2 Basic Line Plot

A basic line graph can be created using plt.plot(). The x and y values are plotted on the graph.

```
# Sample Data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Plotting the Line Graph
plt.plot(x, y)

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

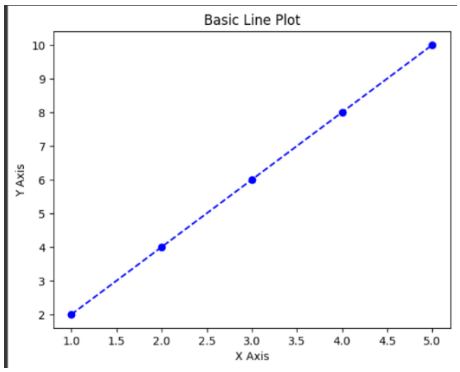


- plt.plot(x, y): Plots the x-values and y-values on a 2D graph.
- plt.show(): Displays the plot on the screen.

1.3 Customizing the Plot

You can add labels, a title, and change the line style for better presentation

```
plt.plot(x, y, color='blue', marker='o', linestyle='--')

# Adding Labels
plt.xlabel('X Axis')
plt.ylabel('Y Axis')

# Adding a Title
plt.title('Basic Line Plot')

plt.show()
```



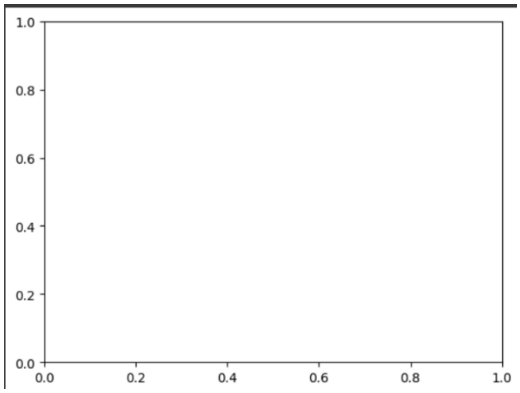
- color, marker, and linestyle allow customization of the line's appearance.
- plt.xlabel() and plt.ylabel() add axis labels.
- plt.title() sets the title of the plot.

2. Figure and Axes

2.1 Creating a Figure and Axes

In Matplotlib, a plot is drawn on a **Figure** (the entire window) and contains one or more **Axes** (individual plots).

```
fig, ax = plt.subplots()
```



- fig: Represents the entire figure or window containing the plot(s).
- ax: Represents a single plot or axes in the figure where the data is plotted.

Example:

```
# Creating a Figure and Axes
fig, ax = plt.subplots()

# Plotting on the Axes
ax.plot(x, y)

# Displaying the Plot
plt.show()
```

2.2 Multiple Subplots

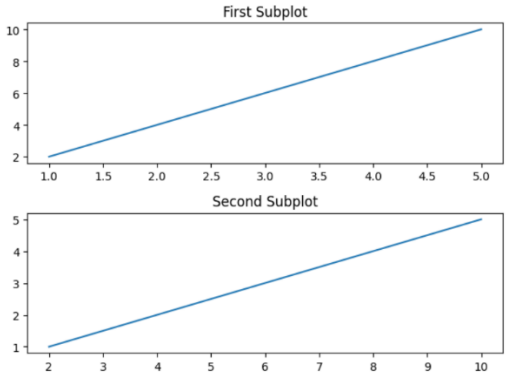
You can create multiple subplots in a single figure using the plt.subplot() function

```
plt.subplot(nrows, ncols, index)
```

- nrows: Number of rows of subplots.
- ncols: Number of columns of subplots.
- index: Position of the current subplot (1-based indexing).

Example:

```
plt.subplot(2, 1, 1)
plt.plot(x, y)
plt.title('First Subplot')

plt.subplot(2, 1, 2)
plt.plot(y, x)
plt.title('Second Subplot')

plt.tight_layout()
plt.show()
```

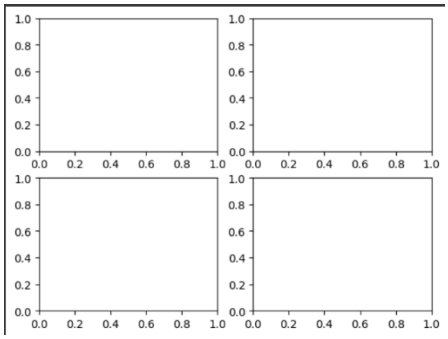


- **plt.subplot(2, 1, 1)**: This creates the first subplot in a 2x1 grid.
- **plt.subplot(2, 1, 2)**: This creates the second subplot.

- **plt.tight_layout()**: Automatically adjusts the spacing to prevent overlapping content.

plt.subplots() : To organize multiple plots in a grid format, we can use plt.subplots() with specified numbers of rows and columns.

```
import matplotlib.pyplot as plt

# Creating a 2x2 grid of subplots
fig, ax = plt.subplots(2, 2)

# Displaying the figure
plt.show()
```



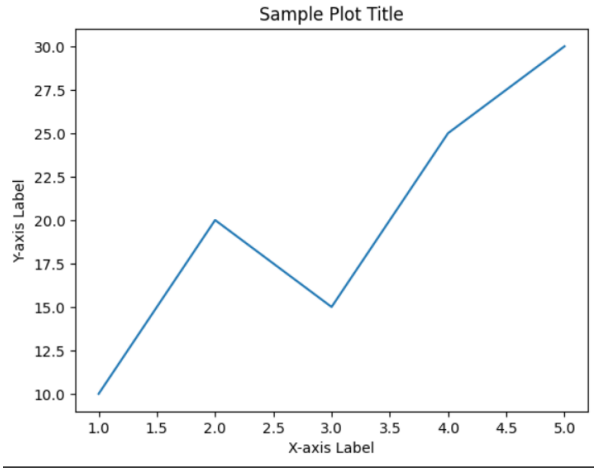
Customizing Plots with Matplotlib

Matplotlib provides several functions to add titles, labels, legends, and grids to make plots more informative.

1. Adding Titles and Labels

Setting titles and labels helps define what the plot represents and clarifies each axis's data.

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 15, 25, 30]


plt.plot(x, y)


plt.title("Sample Plot Title")
plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")


plt.show()
```

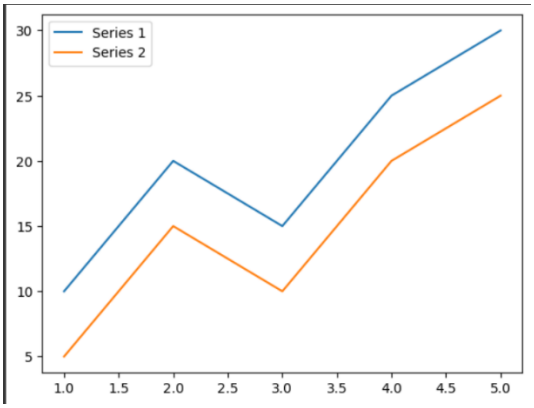


- **plt.title('Title')**: Sets the title of the plot, displayed at the top.
- **plt.xlabel('X-axis Label')**: Sets the label for the x-axis.
- **plt.ylabel('Y-axis Label')**: Sets the label for the y-axis.

2. Adding a Legend

A legend identifies different data series, which is especially helpful when plotting multiple lines.

```
y1 = [10, 20, 15, 25, 30]
y2 = [5, 15, 10, 20, 25]


plt.plot(x, y1, label="Series 1")
plt.plot(x, y2, label="Series 2")


plt.legend()

plt.show()
```



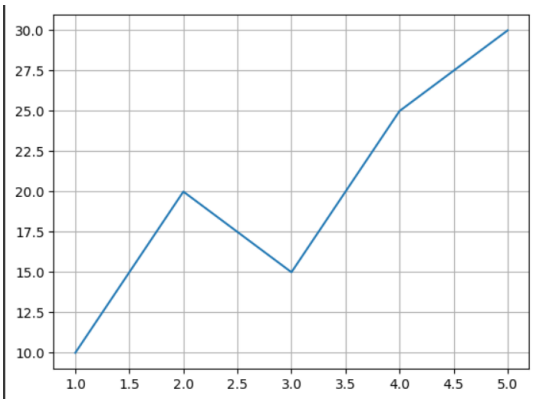
- **plt.legend(['label1', 'label2'])**: Displays a legend with specified labels.

- Alternatively, use label within plt.plot() for each data series, then call plt.legend() to auto-generate it.

3. Enabling or Disabling Gridlines

Gridlines enhance readability by aligning data points visually across axes.

```
# Plotting the data
plt.plot(x, y)

# Adding a grid
plt.grid(True)

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

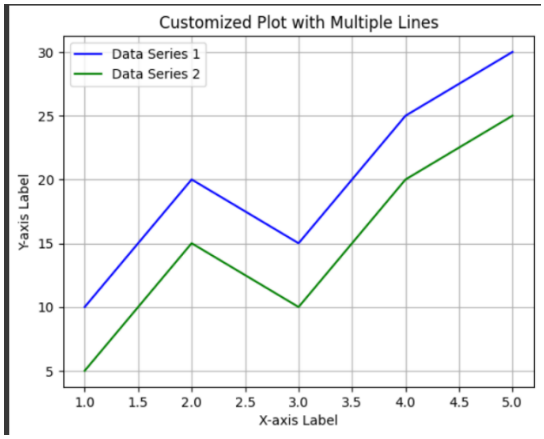


- **plt.grid(True)**: Enables gridlines to make it easier to read and compare data points across axes.
- **plt.grid(False)**: Disables gridlines if they are not needed.

Combining Customizations

Here's a combined example using titles, labels, legends, and gridlines:

```
x = [1, 2, 3, 4, 5]
y1 = [10, 20, 15, 25, 30]
y2 = [5, 15, 10, 20, 25]


plt.plot(x, y1, label="Data Series 1", color="blue")
plt.plot(x, y2, label="Data Series 2", color="green")


plt.title("Customized Plot with Multiple Lines")
plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
plt.legend()
plt.grid(True)


plt.show()
```



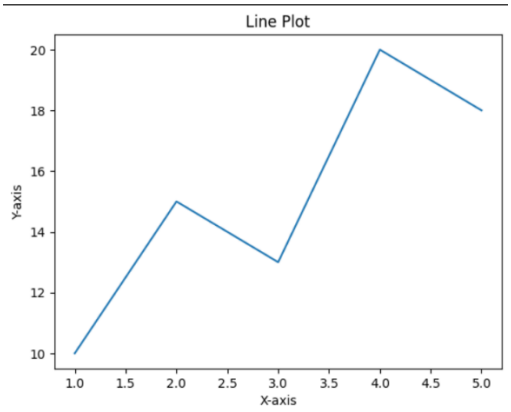
Plotting Different Types of Graphs with Matplotlib

Matplotlib provides various plotting options to visualize data effectively. Below are some of the most commonly used plot types.

1. Line Plot

Line plots are ideal for visualizing trends over time or sequential data.

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 20, 18]

# Creating a line plot
plt.plot(x, y)
plt.title("Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

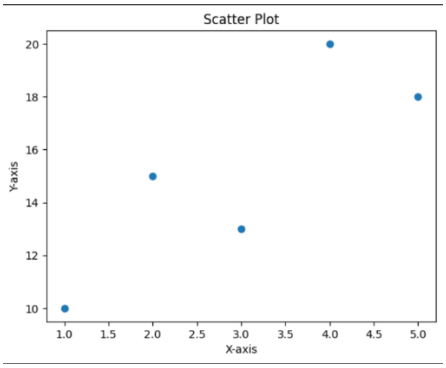


- **plt.plot(x, y)**: Plots y values against x values as a line graph.

2. Scatter Plot

Scatter plots are used to visualize the relationship or correlation between two variables

```
# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 20, 18]

# Creating a scatter plot
plt.scatter(x, y)
plt.title("Scatter Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

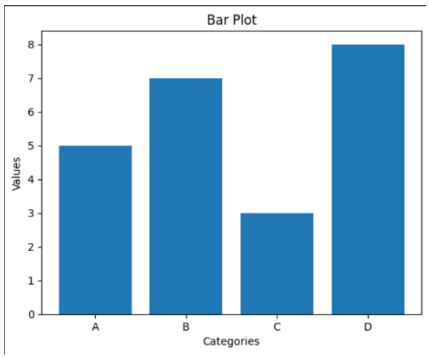


- **plt.scatter(x, y)**: Creates a scatter plot with individual points.

3. Bar Plot

Bar plots display data with rectangular bars and are often used for comparing different categories

```
# Sample data
x = ['A', 'B', 'C', 'D']
heights = [5, 7, 3, 8]

# Vertical bar plot
plt.bar(x, heights)
plt.title("Bar Plot")
plt.xlabel("Categories")
plt.ylabel("Values")
plt.show()
```

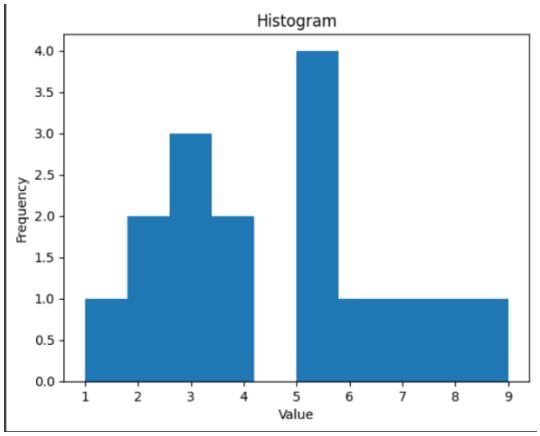


- **plt.bar(x, height)**: Creates a vertical bar plot.
- **plt.barh(y, width)**: Creates a horizontal bar plot.

4. Histogram

Histograms visualize the distribution of data by grouping it into bins.

```
# Sample data
data = [1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 6, 7, 8, 9]

# Creating a histogram
plt.hist(data, bins=10)
plt.title("Histogram")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

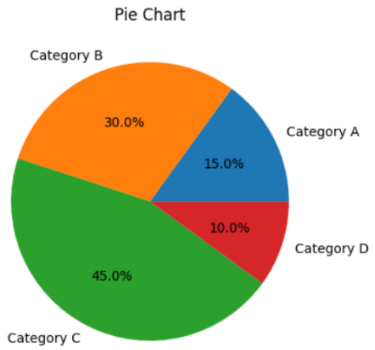


- **`plt.hist(data, bins=10)`**: Creates a histogram with 10 bins, where `data` is grouped to show its frequency distribution.

Pie Chart

Pie charts are used to represent proportions within a whole, showing data as slices of a circle.

```
# Sample data
sizes = [15, 30, 45, 10]
labels = ['Category A', 'Category B', 'Category C', 'Category D']

# Creating a pie chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.title("Pie Chart")
plt.show()
```

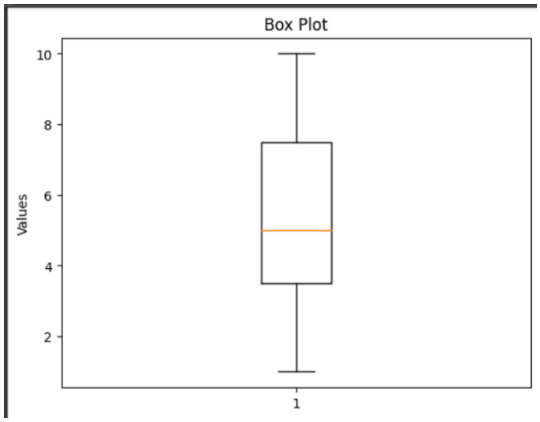


- **plt.pie(sizes, labels=labels)**: Creates a pie chart with labeled slices.
- **autopct='%1.1f%%'**: Adds percentage values to each slice

Box Plot

Box plots summarize the distribution and spread of data through quartiles, showing outliers.

```
# Sample data
data = [1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10]

# Creating a box plot
plt.boxplot(data)
plt.title("Box Plot")
plt.ylabel("Values")
plt.show()
```



- **plt.boxplot(data)**: Creates a box plot to display the spread and skewness of data.

Styling Plots in Matplotlib

Matplotlib offers many ways to customize plots through line styles, colors, and markers. Here's a breakdown of each styling option.

1. Line Styles

Changing line styles can help distinguish different data series and add variety to your plots.

- **Dashed Line**: Use linestyle='--' to create a dashed line.

```
plt.plot(x, y, linestyle='--')
```

- **Solid Line**: The default line style is solid, but you can explicitly set it using `linestyle='-'`

```
plt.plot(x, y, linestyle='-')
```

- **Adding Markers**: Use marker='o' or other symbols to add markers at each data point along the line

```
plt.plot(x, y, marker='o')
```

2. Colors

Matplotlib allows custom colors using standard abbreviations or hex codes.

- **Basic Colors**: Specify colors using a single character. For example, color='r' sets the line to red.

```
plt.plot(x, y, color='r')
```

Hex Color Codes: You can also use hex color codes for more customization

```
plt.plot(x, y, color='#FFA07A')
```

3. Markers

Markers highlight each data point and add emphasis to specific values.

- **Circle Markers**: Set marker='o' to add circular markers to each data point

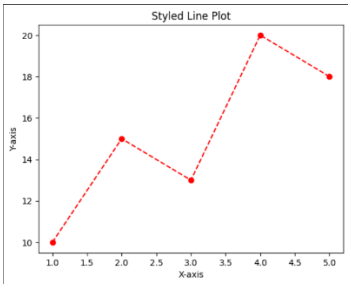
```
plt.plot(x, y, marker='o')
```

- **Square Markers**: Use `marker='s'` to add square markers.

```
plt.plot(x, y, marker='s')
```

- Examples

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 20, 18]


plt.plot(x, y, linestyle='--', color='r', marker='o')
plt.title("Styled Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```



Working with Figures and Axes in Matplotlib

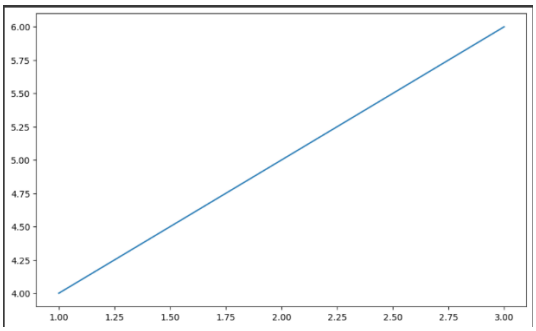
Matplotlib allows you to customize figures with specific sizes, save plots to files, and adjust layouts for readability. Additionally, annotations and text can enhance the clarity of the plot by providing additional context.

1. Figure Size

Adjusting the figure size allows you to control the width and height of your plot, making it more suitable for different screen sizes or print formats.

- **Setting Figure Size**: Use plt.figure(figsize=(width, height)) to specify the figure size in inches.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
```

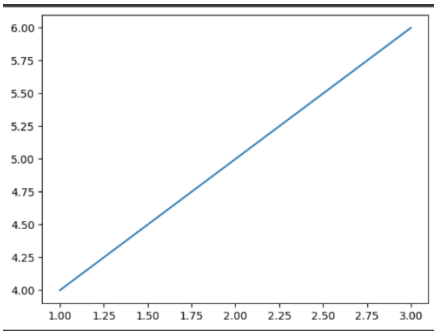


2. Saving Plots

To save a plot, use plt.savefig(). This method can save plots in various formats, such as PNG, PDF, or SVG.

- **Saving the Plot**:

```
plt.plot([1, 2, 3], [4, 5, 6])
plt.savefig('my_plot.png')
```

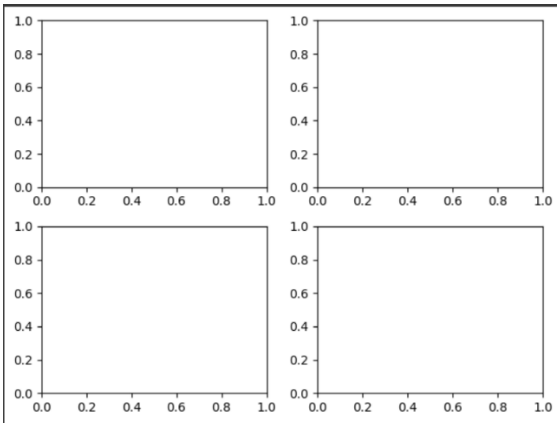


3. Tight Layout

When you have multiple subplots, plt.tight_layout() helps ensure that plots don't overlap and are spaced appropriately. This is particularly useful for avoiding cut-off labels.

- **Adjusting Layout**:

```
fig, axs = plt.subplots(2, 2)
plt.tight_layout()
plt.show()
```



Annotations and Text in Matplotlib

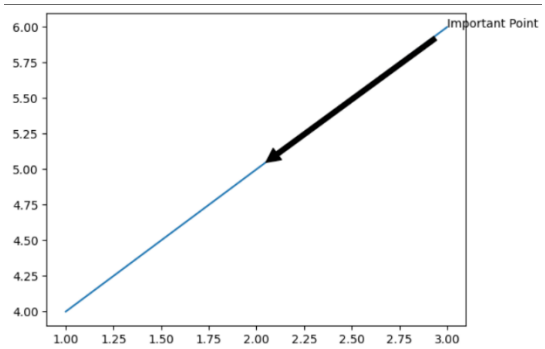
Annotations and custom text can highlight specific data points or areas in a plot, making it easier for viewers to understand important features.

1. Adding Annotations

Annotations combine text with an optional arrow that points to a specific location in the plot.

- **Adding Annotations**:
    - plt.annotate('text', xy=(x, y), xytext=(x+offset, y+offset), arrowprops=dict(facecolor='color')) adds text with an arrow.

```
plt.plot([1, 2, 3], [4, 5, 6])
plt.annotate('Important Point', xy=(2, 5), xytext=(3, 6),
             arrowprops=dict(facecolor='black', shrink=0.05))
plt.show()
```

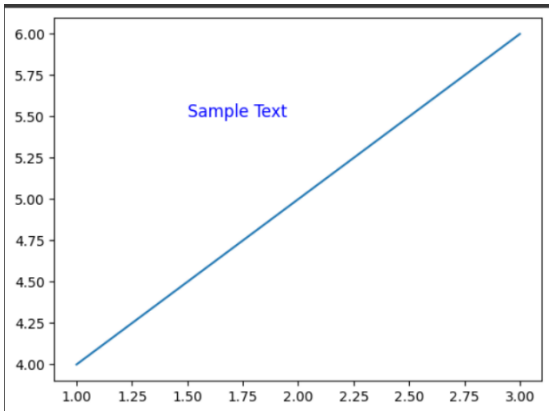


2. Adding Text

Custom text can be placed at any position in the plot, independent of specific data points.

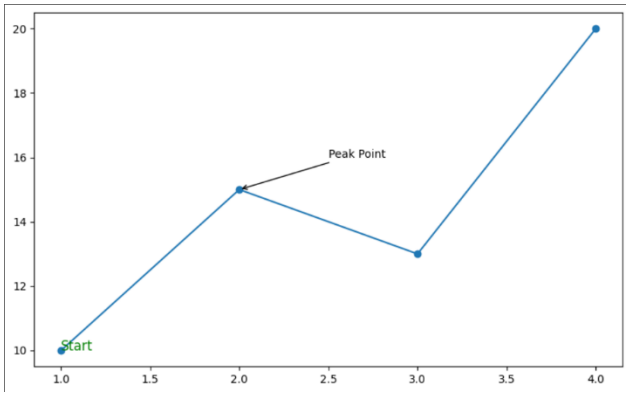
- **Adding**

```
plt.plot([1, 2, 3], [4, 5, 6])
plt.text(1.5, 5.5, 'Sample Text', fontsize=12, color='blue')
plt.show()
```



Examples of Working with Figures, Axes, and Text Annotations

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))
x = [1, 2, 3, 4]
y = [10, 15, 13, 20]
plt.plot(x, y, marker='o')

plt.annotate('Peak Point', xy=(2, 15), xytext=(2.5, 16),
             arrowprops=dict(facecolor='red', arrowstyle='->'))

plt.text(1, 10, 'Start', fontsize=12, color='green')

plt.savefig('styled_plot.png')

plt.tight_layout()
plt.show()
```



Advanced Plot Types in Matplotlib

These advanced plot types can help visualize complex data relationships and distributions, providing additional insight compared to traditional 2D plots.

1. Heatmap

A heatmap is useful for visualizing matrix-like data. Each value in the matrix is represented by a color, often using gradients to represent magnitude

.

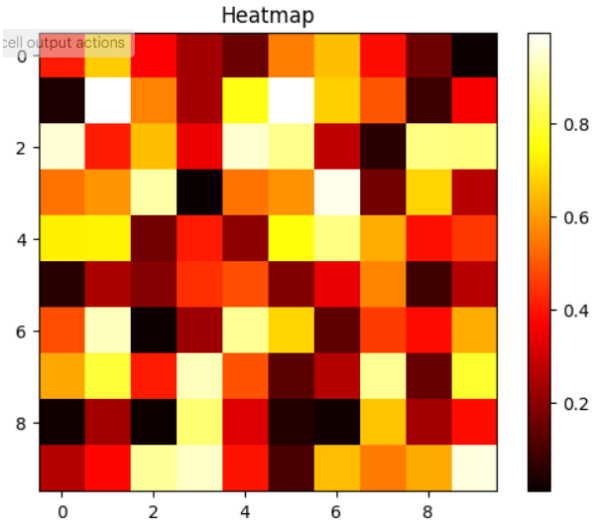
- **Syntax**:

```
plt.imshow(data, cmap='hot', interpolation='nearest')
```

- **Parameters**:
  - data: 2D array or matrix to visualize.
  - cmap: Color map to apply (e.g., 'hot', 'cool', 'viridis').
  - interpolation: Determines how data points are interpolated (e.g., 'nearest' for blocky images).

Example:

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

data = np.random.rand(10, 10)
plt.imshow(data, cmap='hot', interpolation='nearest')
plt.colorbar()
plt.title("Heatmap")
plt.show()
```



2. 3D Plot

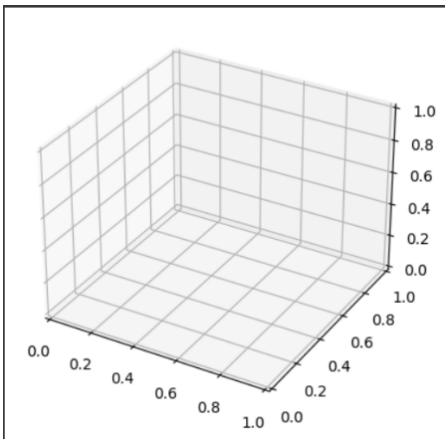
3D plots enable the visualization of data with three dimensions, making them ideal for showing spatial data or relationships between multiple variables.

- **Setup**:
  - First, import Axes3D to enable 3D plotting.

- Then, create a 3D plot using the projection='3d' option when setting up the axes.

- **Syntax**:

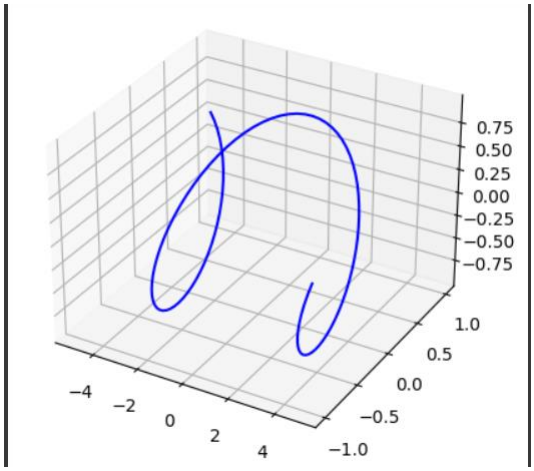
```
from mpl_toolkits.mplot3d import Axes3D
ax = plt.axes(projection='3d')
ax.plot3D(x, y, z)
```



- **Parameters**:
  - x, y, z: Coordinates of the data points to plot.

Example:

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

x = np.linspace(-5, 5, 100)
y = np.sin(x)
z = np.cos(x)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot3D(x, y, z, 'blue')
ax.set_title("3D Line Plot")
plt.show()
```

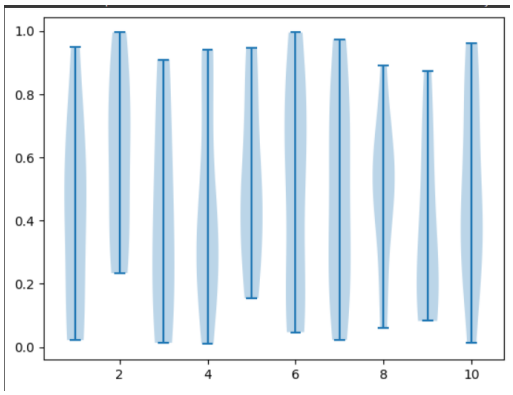


3. Violin Plot

A violin plot visualizes the distribution of data, similar to a boxplot, but also shows the probability density of the data at different values. It combines aspects of boxplots and KDE (Kernel Density Estimation).

- **Syntax**:

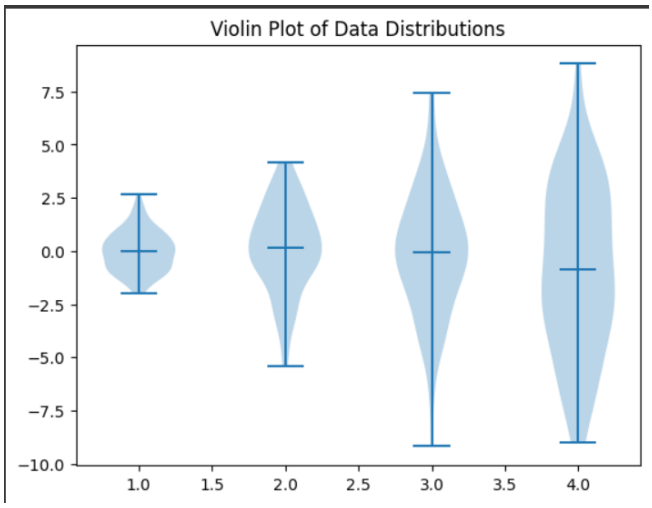
```
plt.violinplot(data)
```



- - **Parameters**:
    - `data`: Array-like dataset to visualize. Typically multiple groups to compare distributions.

- **Example**:

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

data = [np.random.normal(0, std, 100) for std in range(1, 5)]

plt.violinplot(data, showmeans=True)
plt.title("Violin Plot of Data Distributions")
plt.show()
```



Customizing Ticks and Axes in Matplotlib

In Matplotlib, customizing ticks and axis properties can significantly enhance the readability and presentation of a plot. Here, we'll explore various methods for setting ticks, defining axis limits, and adjusting the axis scale.

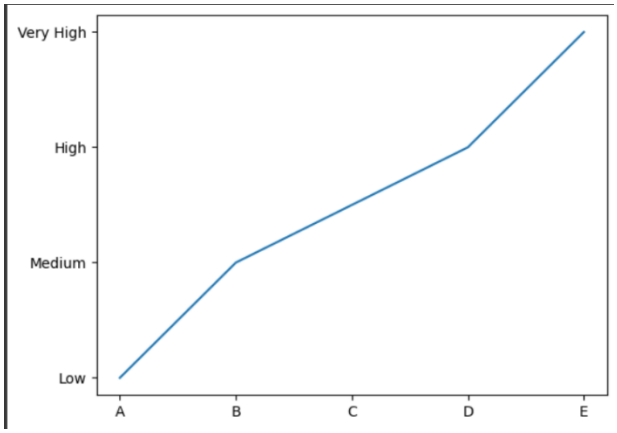
1. Setting Ticks

Ticks are the values that represent data points on the axes. You can modify the ticks on both the X and Y axes for better alignment with your data.

- **plt.xticks(ticks, labels)**: Customize the ticks on the X-axis.
  - ticks: List of positions where you want to place ticks on the X-axis.
  - labels: List of labels corresponding to each tick position.
- **plt.yticks(ticks, labels)**: Customize the ticks on the Y-axis.
  - ticks: List of positions where you want to place ticks on the Y-axis.
  - labels: List of labels for the specified tick positions.

Example:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
plt.plot(x, y)
plt.xticks([1, 2, 3, 4, 5], ['A', 'B', 'C', 'D', 'E'])
plt.yticks([10, 20, 30, 40], ['Low', 'Medium', 'High', 'Very High'])
plt.show()
```



In this example, custom labels are assigned to the tick values on both axes, making the data more descriptive.

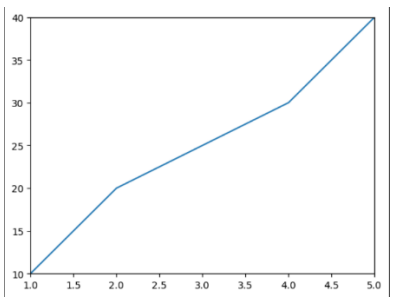
2. Setting Axis Limits

Adjusting axis limits allows you to control the range of values shown on the X and Y axes. This can help focus on specific data ranges or ensure that the axis limits align with external requirements.

- **plt.xlim(min, max)**: Set the limits of the X-axis.
- **plt.ylim(min, max)**: Set the limits of the Y-axis.

Example:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
plt.plot(x, y)
plt.xlim(1, 5)  # Set X-axis from 1 to 5
plt.ylim(10, 40)  # Set Y-axis from 10 to 40
plt.show()
```



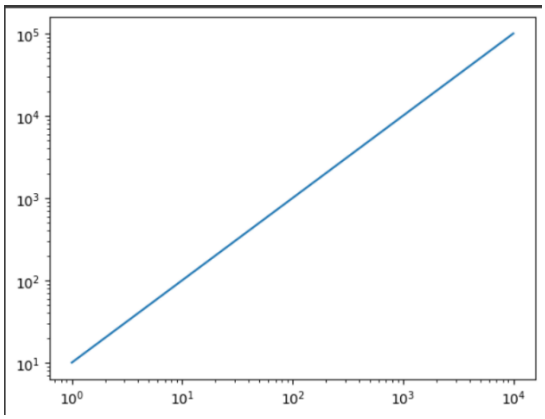
3. Logarithmic Scale

A logarithmic scale is useful when dealing with data that spans multiple orders of magnitude. It helps to display large values more compactly and can make exponential relationships appear linear.

- **plt.xscale('log')**: Sets the X-axis to a logarithmic scale.
- **plt.yscale('log')**: Sets the Y-axis to a logarithmic scale.

Example:

```
import matplotlib.pyplot as plt

x = [1, 10, 100, 1000, 10000]
y = [10, 100, 1000, 10000, 100000]
plt.plot(x, y)
plt.xscale('log')
plt.yscale('log')
plt.show()
```



Combining Plots and Working with Dates in Matplotlib

Combining multiple plots and working with date data in Matplotlib can help in creating informative, multi-dimensional visuals. This section covers how to plot multiple lines, create twin axes for data on different scales, and handle date formats in your plots.

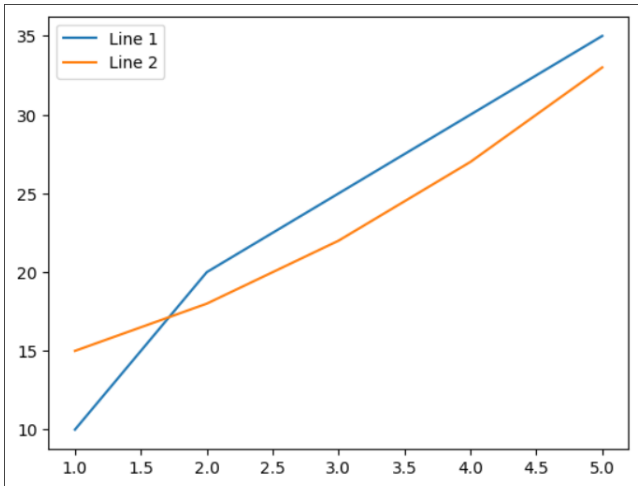
1. Multiple Lines in a Plot

Adding multiple lines to a single plot is useful for comparing datasets on the same axes. A legend helps differentiate between the lines.

- **plt.plot(x, y1, label='Line 1')**: Plot the first line with y1 values.
- **plt.plot(x, y2, label='Line 2')**: Plot the second line with y2 values.
- **plt.legend()**: Display a legend to label each line.

Example:

```python
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y1 = [10, 20, 25, 30, 35]
y2 = [15, 18, 22, 27, 33]

plt.plot(x, y1, label='Line 1')
plt.plot(x, y2, label='Line 2')
plt.legend()  # Display legend to differentiate lines
plt.show()
```



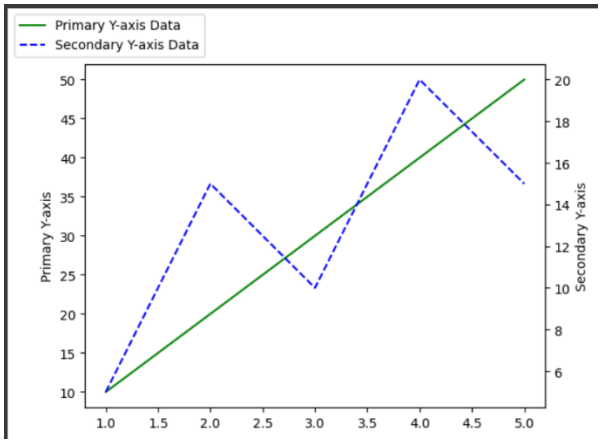
2. Twin Axes

Using twin axes is helpful when you want to display two datasets with different scales on the same plot. For instance, temperature and humidity data can be displayed on two different Y-axes for clarity.

- **ax1 = plt.gca()**: Get the current axis object (ax1).
- **ax2 = ax1.twinx()**: Create a secondary Y-axis (ax2) on the same plot.

Example:

```python
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y1 = [10, 20, 30, 40, 50]  # Data for primary Y-axis
y2 = [5, 15, 10, 20, 15]   # Data for secondary Y-axis

fig, ax1 = plt.subplots()
ax1.plot(x, y1, 'g-', label='Primary Y-axis Data')
ax1.set_ylabel('Primary Y-axis')

ax2 = ax1.twinx()
ax2.plot(x, y2, 'b--', label='Secondary Y-axis Data')
ax2.set_ylabel('Secondary Y-axis')

fig.legend(loc='upper left')
plt.show()
```



Working with Dates

Matplotlib can plot time series data efficiently, with built-in functionality for handling dates and automatically formatting date labels.

- **Plotting with Dates**
  - **import matplotlib.dates as mdates**: Import the dates module for date handling.
  - **plt.plot(dates, y)**: Plot time series data.
  - **plt.gcf().autofmt_xdate()**: Automatically formats the date labels to prevent overlap and improve readability.

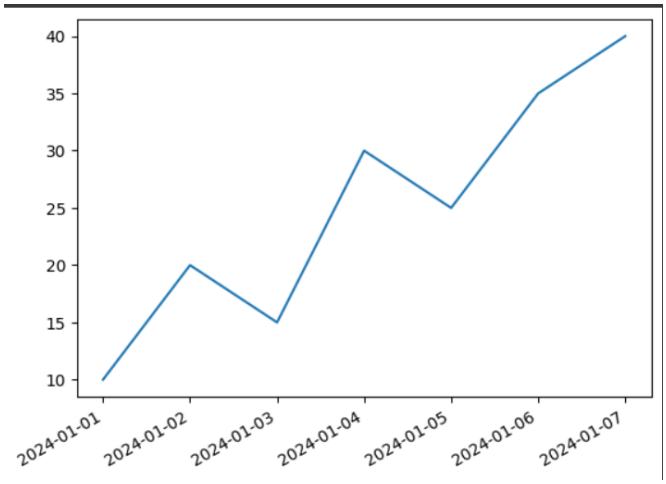
Example

In this example, dates is a list of datetime objects, and plt.gcf().autofmt_xdate() automatically rotates the labels for readability.

- **Formatting Dates**
  - **ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))**: Customizes the date format. For example, %Y-%m-%d would display dates in YYYY-MM-DD format.

Example:

```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime

dates = [datetime.date(2024, 1, i) for i in range(1, 8)]
y = [10, 20, 15, 30, 25, 35, 40]

plt.plot(dates, y)
plt.gcf().autofmt_xdate()  # Rotate date labels for readability
plt.show()
```



Interactive Plots in Matplotlib

Matplotlib's interactive mode enables dynamic plot updates, allowing for real-time visualization of data. This can be especially useful for applications like monitoring live data, animations, or interactive dashboards.

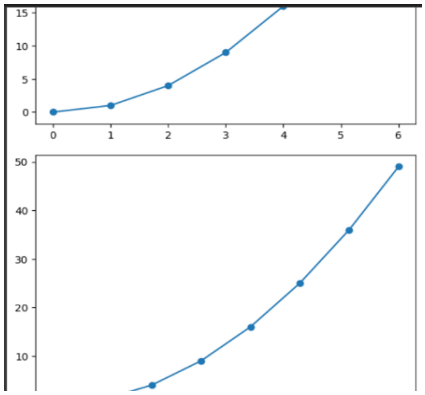
1. Enabling Interactive Mode

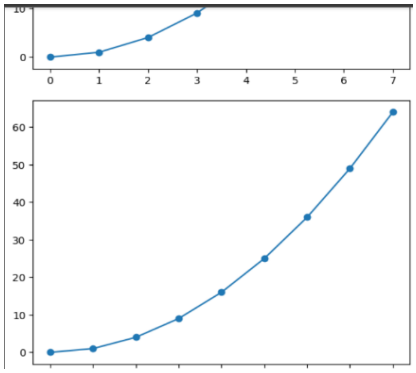
Interactive mode allows you to see updates in the plot immediately without needing to call plt.show() repeatedly. It's useful for continuous updates and animations in live data plotting.

- **plt.ion()**: Enables interactive mode, making the plot update dynamically.
- **plt.ioff()**: Disables interactive mode, so the plot only displays once with plt.show().

Example:

```
import matplotlib.pyplot as plt

plt.ion()
x = [0]
y = [0]

# Initial plot
plt.plot(x, y, marker='o')
plt.xlim(0, 10)
plt.ylim(0, 10)

for i in range(1, 11):
    x.append(i)
    y.append(i**2)
    plt.plot(x, y, marker='o')
    plt.pause(0.5)

plt.ioff()
plt.show()
```





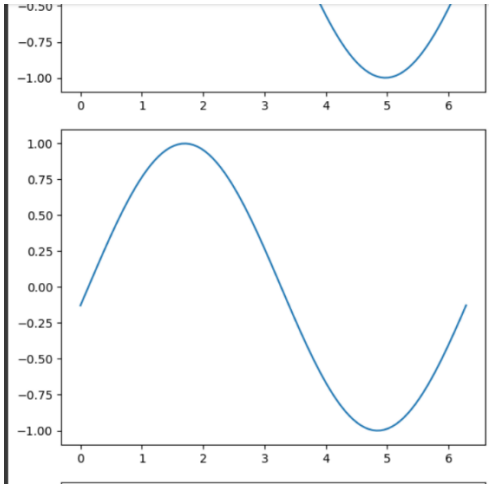
2. Updating Plots Dynamically

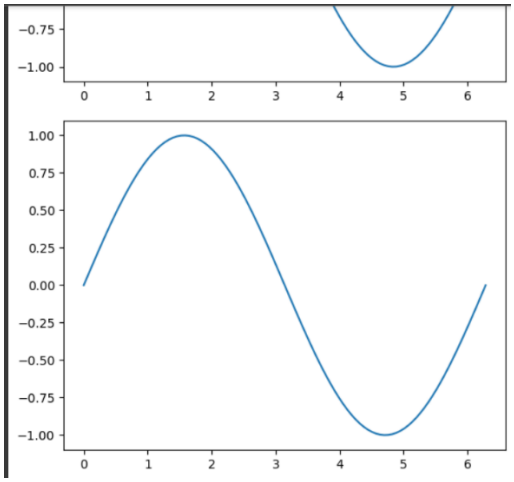
The plt.pause(interval) function can be used to create real-time updates in a plot, making it ideal for displaying live data. Each call to plt.pause() allows the figure to update before the next line of code executes, effectively creating an animation.

- **plt.pause(interval)**: Pauses the plot for a given interval in seconds, allowing the plot to refresh.

Example:

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

x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

plt.ion()  # Enable interactive mode
for phase in np.linspace(0, 2 * np.pi, 50):
    plt.clf()  # Clear the figure to refresh the plot
    plt.plot(x, np.sin(x + phase))  # Update the plot with a phase shift
    plt.pause(0.1)  # Pause briefly to create animation effect

plt.ioff()  # Turn off interactive mode
plt.show()
```


−0.75
−1.00
0
1
2
3
4
5
6
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


−0.75
−1.00
0
1
2
3
4
5
6
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

Introduction to Seaborn

Seaborn is a powerful visualization library based on Matplotlib, offering a high-level interface for drawing attractive and informative statistical graphics. Seaborn works seamlessly with Pandas, making it a great choice for data analysis and exploration.

1. Importing Seaborn

To start using Seaborn, it needs to be imported into your workspace. Since Seaborn is built on top of Matplotlib, it can work alongside Matplotlib functions and complements Pandas well, as it directly takes in Pandas DataFrames and understands column names for easy plotting.

- **import seaborn as sns**: Import Seaborn into your environment.

Example:

```
import seaborn as sns
import pandas as pd

# Sample DataFrame
data = pd.DataFrame({
    'category': ['A', 'B', 'C', 'D'],
    'values': [10, 20, 15, 25]
})

# Simple Seaborn bar plot
sns.barplot(x='category', y='values', data=data)
```

```
<Axes: xlabel='category', ylabel='values'>
```

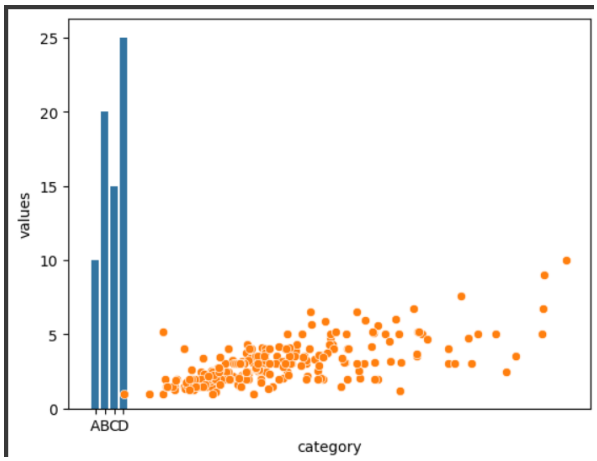
Seaborn Styling

Seaborn comes with a set of built-in styles that can enhance the aesthetics of your plots. Setting the style gives the plot a polished look with minimal code. Seaborn offers various style options:

- **sns.set(style='whitegrid')**: Sets the default style to whitegrid, which includes a background grid to aid readability.
- Other styles include:
  - **dark**
  - **white**
  - **ticks**

Example:

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

# Set style for the plot
sns.set(style='whitegrid')  # Setting style to 'whitegrid'
data = sns.load_dataset('tips')

# Example plot with Seaborn style
sns.scatterplot(x='total_bill', y='tip', data=data)
plt.show()
```



Basic Plotting Functions in Seaborn

Seaborn provides a range of functions to create different types of plots. These functions integrate seamlessly with Pandas DataFrames, making it easy to visualize data relationships and distributions. Here's a guide to some of the most commonly used plot types in Seaborn.

1. Line Plot

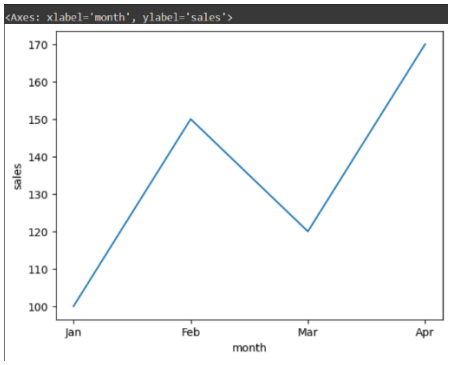
A line plot is useful for visualizing trends over time or ordered categories. Seaborn's lineplot handles multiple lines effortlessly and integrates with DataFrames, which makes plotting time series or continuous data simple.

- **sns.lineplot(x, y, data)**: Plots a line graph with x and y values from the given DataFrame.

Example:

```python
import seaborn as sns
import pandas as pd

# Sample DataFrame
data = pd.DataFrame({
    'month': ['Jan', 'Feb', 'Mar', 'Apr'],
    'sales': [100, 150, 120, 170]
})

sns.lineplot(x='month', y='sales', data=data)
```

```
<Axes: xlabel='total_bill', ylabel='tip'>
```



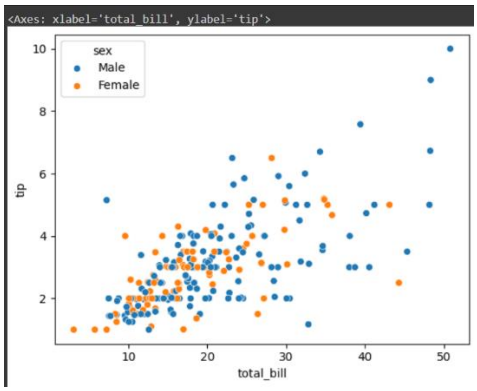
2. Scatter Plot

A scatter plot displays data points for two continuous variables and can show relationships or clusters. Seaborn allows adding a third dimension using the hue parameter for color-coding by a categorical variable.

- **sns.scatterplot(x, y, data)**: Plots a scatter plot with x and y values.
- **hue='category'**: Optional, adds color coding based on a categorical column.

Example:

```
import seaborn as sns

# Load sample data
data = sns.load_dataset('tips')
sns.scatterplot(x='total_bill', y='tip', data=data, hue='sex')
```



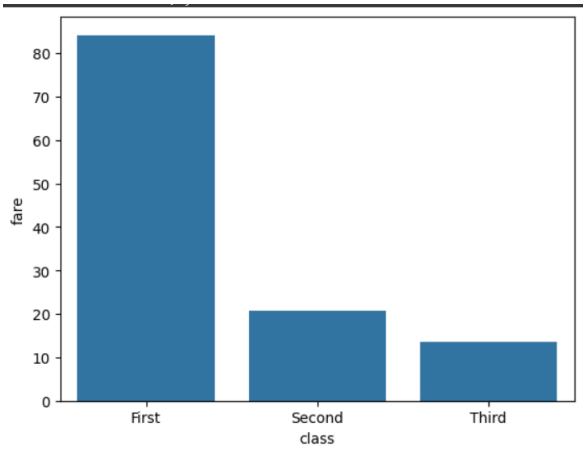
Bar Plot

Bar plots display the relationship between a categorical variable and a numerical variable. This is useful for comparing averages or totals across categories. The ci=None parameter removes confidence intervals for simplicity.

- **sns.barplot(x, y, data, ci=None)**: Creates a bar plot with x (categorical) and y (numerical) values.

Example:

```
import seaborn as sns

data = sns.load_dataset('titanic')
sns.barplot(x='class', y='fare', data=data, ci=None)
```



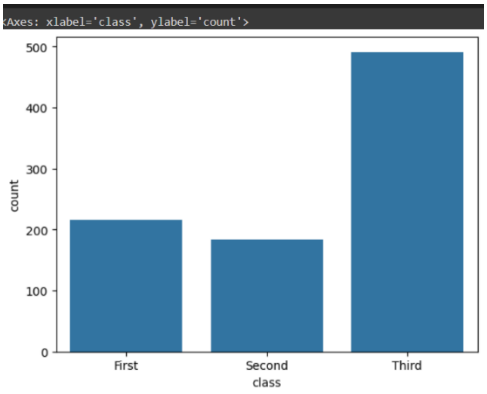
4. Count Plot

A count plot shows the count of occurrences of each category in a categorical variable. This is especially useful for categorical data, such as demographics or survey responses.

- **sns.countplot(x, data)**: Plots the counts of each unique value in the categorical x column.

Example:

```
import seaborn as sns

data = sns.load_dataset('titanic')
sns.countplot(x='class', data=data)
```



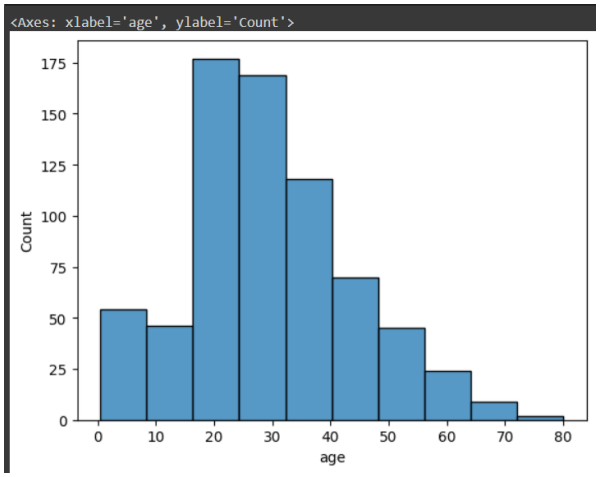
5.Histogram

Histograms visualize the distribution of a single numerical variable by grouping data into bins. This is useful for examining the spread, central tendency, and variability of a dataset.

- **sns.histplot(data, bins=10)**: Plots a histogram with a specified number of bins.

Example:

```
import seaborn as sns

data = sns.load_dataset('titanic')
sns.histplot(data['age'], bins=10)
```



Categorical Plots in Seaborn

Categorical plots help visualize the distribution and relationship of categorical data. Seaborn provides several types of categorical plots, each with its unique way of representing data. Here are four popular categorical plots: Box Plot, Violin Plot, Strip Plot, and Swarm Plot.

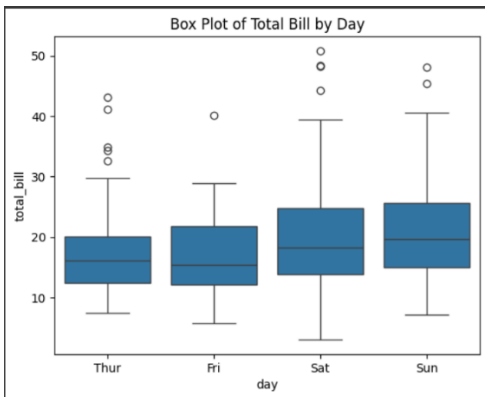
1. Box Plot

Description: The Box Plot is used to display the distribution of quantitative data based on a five-number summary: minimum, first quartile, median, third quartile, and maximum.

- **Function:** sns.boxplot(x, y, data)
- **Usage:** Use Box Plots when you want to get an overview of data distribution and easily identify outliers.

Code Example:

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

# Example data
data = sns.load_dataset("tips")

# Box Plot
sns.boxplot(x="day", y="total_bill", data=data)
plt.title("Box Plot of Total Bill by Day")
plt.show()
```



Key Features:

- Displays median as a central line.
- Shows quartiles and outliers.
- Helps identify data spread and symmetry**.**

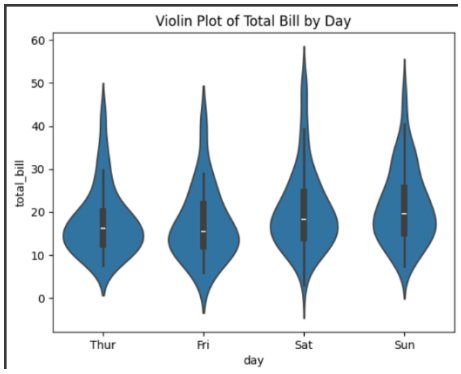
2. Violin Plot

Description: Violin Plots combine the features of a Box Plot and a Kernel Density Plot, giving more information on data distribution by showing density on each side.

- **Function:** sns.violinplot(x, y, data)
- **Usage:** Violin Plots are useful when you want to visualize both the distribution and the probability density of the data.

Code Example:

```python
# Violin Plot
sns.violinplot(x="day", y="total_bill", data=data)
plt.title("Violin Plot of Total Bill by Day")
plt.show()
```



Key Features:

- Shows density of the data at different values.
- Displays quartiles and median within the plot.
- Combines KDE and Box Plot elements for enhanced visualization

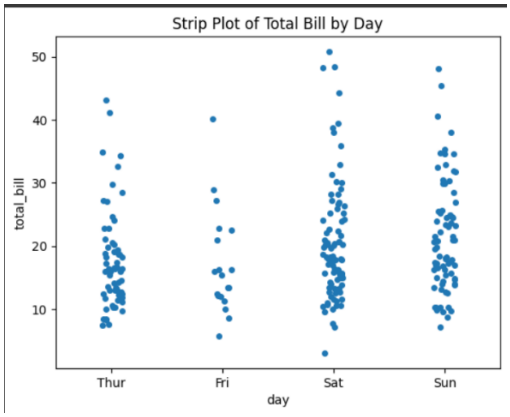
3. Strip Plot

Description: The Strip Plot displays individual data points over a categorical variable, often layered over a Box or Violin Plot.

- **Function:** sns.stripplot(x, y, data)
- **Usage**: Use Strip Plots to visualize the distribution and spread of individual data points, especially if you want to see each value distinctly.

Code Example

```
# Strip Plot
sns.stripplot(x="day", y="total_bill", data=data)
plt.title("Strip Plot of Total Bill by Day")
plt.show()
```



Key Features:

- Points are plotted directly above each category.

- Best for visualizing each individual data point.
- Often combined with Box or Violin Plots for added context.

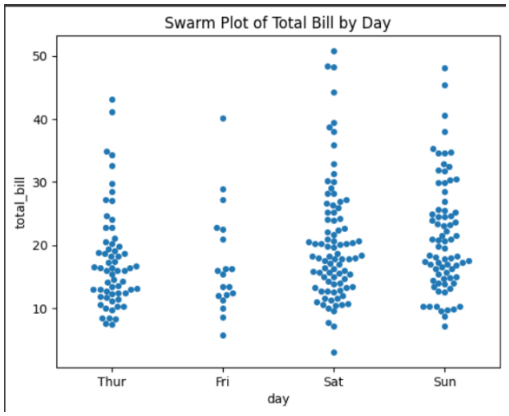
4. Swarm Plot

Description: The Swarm Plot is similar to the Strip Plot but arranges points to avoid overlap, making each point distinct.

- **Function**: sns.swarmplot(x, y, data)
- **Usage**: Swarm Plots are ideal when you want to show individual points without overlap, providing a clear view of data distribution**.**

Code Example:

```
# Swarm Plot
sns.swarmplot(x="day", y="total_bill", data=data)
plt.title("Swarm Plot of Total Bill by Day")
plt.show()
```



Key Features:

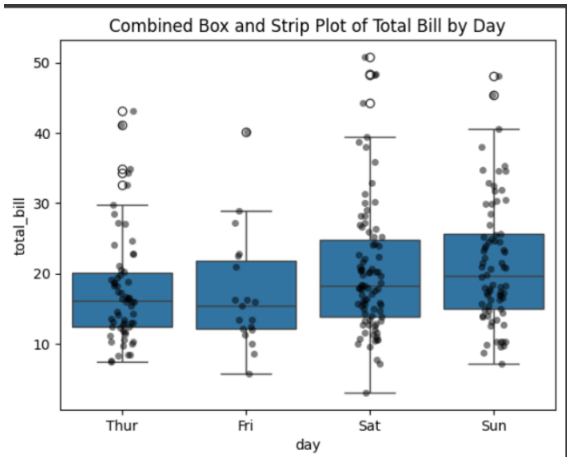
- Adjusts data points to prevent overlap.
- Maintains clarity by spreading out points horizontally.
- Useful for examining individual points in dense data sets.

Combining Categorical Plots

For a more detailed visualization, you can combine these plots. For instance, layering a **Strip Plot** on top of a **Box Plot** provides both summary statistics and individual data points.

Code Example:

```
# Combination of Box Plot and Strip Plot
sns.boxplot(x="day", y="total_bill", data=data)
sns.stripplot(x="day", y="total_bill", data=data, color="black", alpha=0.5)
plt.title("Combined Box and Strip Plot of Total Bill by Day")
plt.show()
```



Distribution Plots in Seaborn

Distribution plots are essential for visualizing the distribution of data. Seaborn provides various types of distribution plots, including **KDE Plot**, **Pair Plot**, **Joint Plot**, and **Dist Plot** (deprecated). These plots are helpful to understand the shape, spread, and relationships between variables.

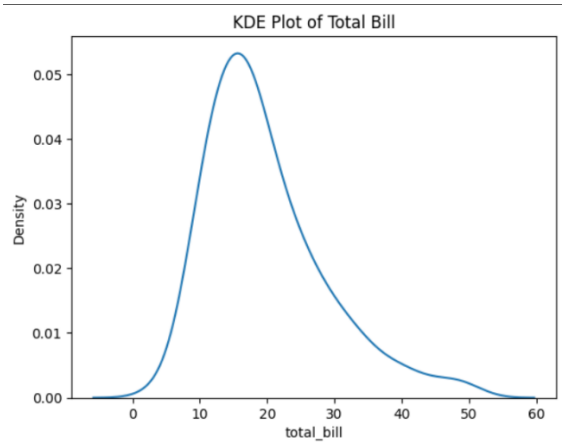
1. KDE Plot (Kernel Density Estimation)

Description: KDE Plot is a method for estimating the probability density function of a continuous variable. It provides a smooth curve representing the data distribution, which is particularly useful for visualizing single-variable distributions.

- **Function**: sns.kdeplot(data)
- **Usage**: Use KDE Plot to understand the distribution of data without relying on a histogram's bin-based approach.

Code Example:

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

# Example data
data = sns.load_dataset("tips")["total_bill"]

# KDE Plot
sns.kdeplot(data=data)
plt.title("KDE Plot of Total Bill")
plt.show()
```



Key Features:

- Displays the probability density function.
- Smooth and continuous curve representation.
- Useful for analyzing continuous data distribution.

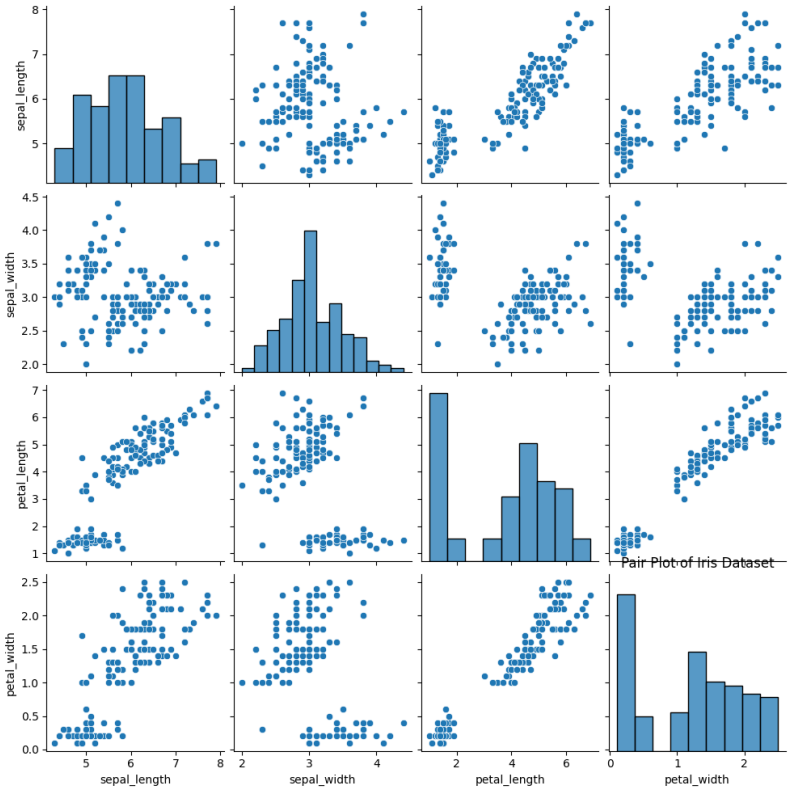
2. Pair Plot

Description: The Pair Plot creates a matrix of scatter plots for each pair of numerical variables in a DataFrame. It also shows KDE plots or histograms for each variable on the diagonal, providing insights into relationships between pairs of variables.

- **Function**: sns.pairplot(data)
- **Usage**: Use Pair Plots to explore relationships between multiple variables, especially when examining data correlation and patterns.

Code Example:

```
# Pair Plot
sns.pairplot(data=sns.load_dataset("iris"))
plt.title("Pair Plot of Iris Dataset")
plt.show()
```



Key Features:

- Matrix of scatter plots for each variable pair.
- Diagonal plots show distribution for each individual variable.
- Ideal for multi-variable exploration and correlation analysis.

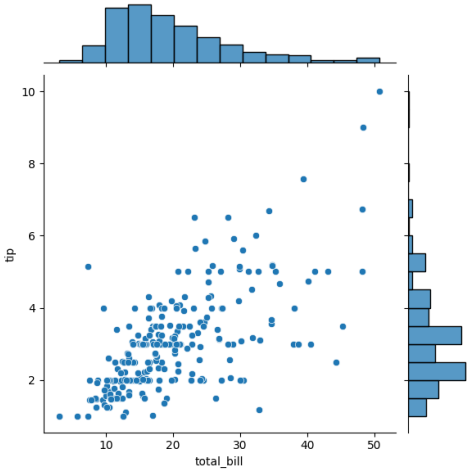
3. Joint Plot

Description: The Joint Plot shows the relationship between two variables, typically as a scatter plot, with histograms or KDE plots on the x and y axes. This plot helps visualize the relationship and distribution of two variables together.

- **Function**: sns.jointplot(x, y, data)
- **Usage**: Use Joint Plots to analyze the relationship between two continuous variables and to examine both marginal and joint distributions.

Code Example:

```
# Joint Plot
sns.jointplot(x="total_bill", y="tip", data=sns.load_dataset("tips"), kind="scatter")
plt.show()
```



Key Features:

- Combines scatter plot with histograms or KDE plots.
- Useful for examining bivariate relationships and distributions.
- Supports different kinds (scatter, kde, hex) for varied visualization.

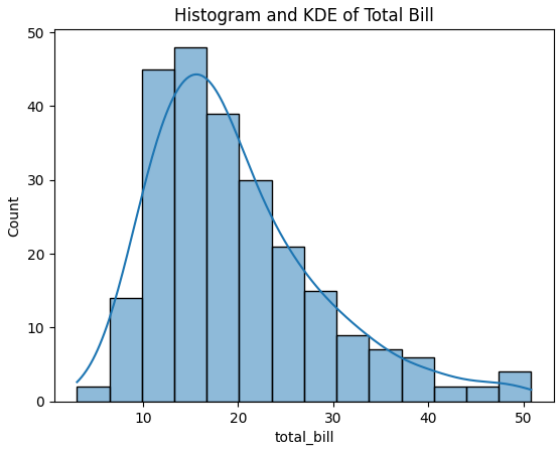
4. Dist Plot (Deprecated)

Description: The Dist Plot was previously used to show a histogram with an optional KDE overlay to visualize the distribution of a variable. In newer Seaborn versions, **sns.distplot** is deprecated and replaced with sns.histplot or sns.kdeplot for more flexible plotting.

- **Function**: sns.distplot(data) (Deprecated in favor of sns.histplot and sns.kdeplot)
- **Usage**: Use sns.histplot for histograms and sns.kdeplot for KDE plots in newer Seaborn versions.

Code Example:

```
# Replacing sns.distplot with sns.histplot
sns.histplot(data=data, kde=True)
plt.title("Histogram and KDE of Total Bill")
plt.show()
```



Key Features:

- Shows a histogram and KDE plot in one view (deprecated).
- Useful for understanding the frequency distribution of data.
- Now better implemented with sns.histplot for flexibility.

Regression Plots and Heatmaps in Seaborn

Regression plots are useful for visualizing relationships and trends between variables, while heatmaps provide a way to visualize matrices, such as correlation matrices. Seaborn makes it easy to create both types of plots with a few simple functions.

Regression Plots

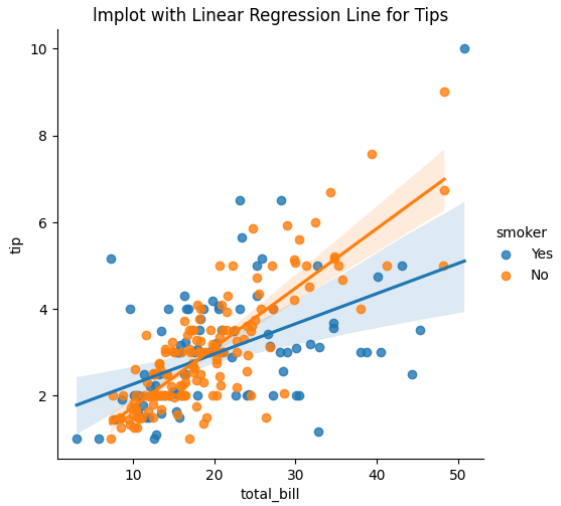
1. lmplot

Description: The lmplot is a high-level function for creating a scatter plot with an overlaid linear regression line. It can be used for different subsets of the data by using the hue parameter to color-code based on a category.

- **Function**: sns.lmplot(x, y, data)
- **Usage**: Use lmplot for quick visualization of relationships with a linear regression line and category-based color differentiation.

Code Example:

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

# Example data
data = sns.load_dataset("tips")

# lmplot
sns.lmplot(x="total_bill", y="tip", data=data, hue="smoker")
plt.title("lmplot with Linear Regression Line for Tips")
plt.show()
```



Key Features:

- Displays linear regression line by default.
- Supports hue to separate data by category.
- Ideal for exploring relationships and trends.

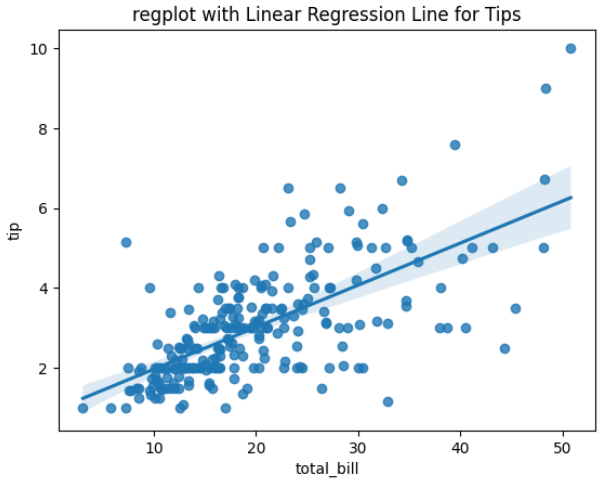
2. regplot

Description: The regplot is an Axes-level function, which means it can be embedded into larger plots with more customization. Like lmplot, it also plots data points with a linear regression line.

- **Function**: sns.regplot(x, y, data)
- **Usage**: Use regplot when you need more flexibility in plotting and when embedding regression plots in larger figure setups.

Code Example:

```
# regplot
sns.regplot(x="total_bill", y="tip", data=data)
plt.title("regplot with Linear Regression Line for Tips")
plt.show()
```



Key Features:

- Allows for embedding within a subplot or larger figure.
- More customizable for detailed visualizations.
- Useful for plotting trends alongside other plots.

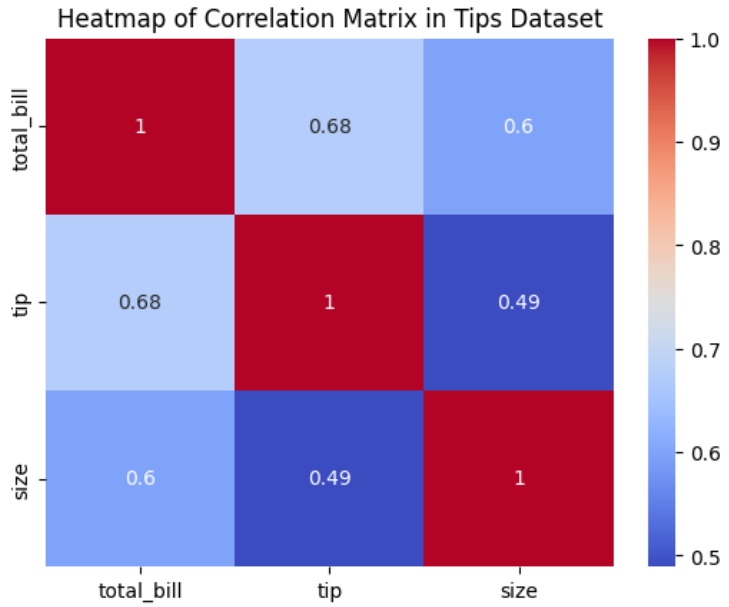
Heatmap

Description: The heatmap function is used to visualize matrices, particularly correlation matrices, in a grid format where each cell's color represents the magnitude of the data point. This is particularly helpful for understanding relationships between variables in a dataset.

- **Function**: sns.heatmap(data, annot=True, cmap='coolwarm')
- **Usage**: Use Heatmaps to analyze correlation matrices, frequency matrices, or any other data that can be represented in a matrix form.

Code Example:

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

# Example data
data = sns.load_dataset("tips")

# Select only numeric columns for correlation
numeric_data = data.select_dtypes(include=np.number)

# Correlation matrix of the numeric columns
correlation_matrix = numeric_data.corr()

# Heatmap
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title("Heatmap of Correlation Matrix in Tips Dataset")
plt.show()
```



Key Features:

- Shows magnitude through color intensity.
- annot=True displays values in each cell for clarity.
- Customizable color palettes with cmap to enhance readability.

Plots in Seaborn

Seaborn provides several tools for exploring pairwise and multivariate relationships within data, especially through the **PairGrid**, **FacetGrid**, and **JointGrid**. These plots allow you to explore relationships, create customized grids, and observe trends across multiple variables.

Pairwise Relationships

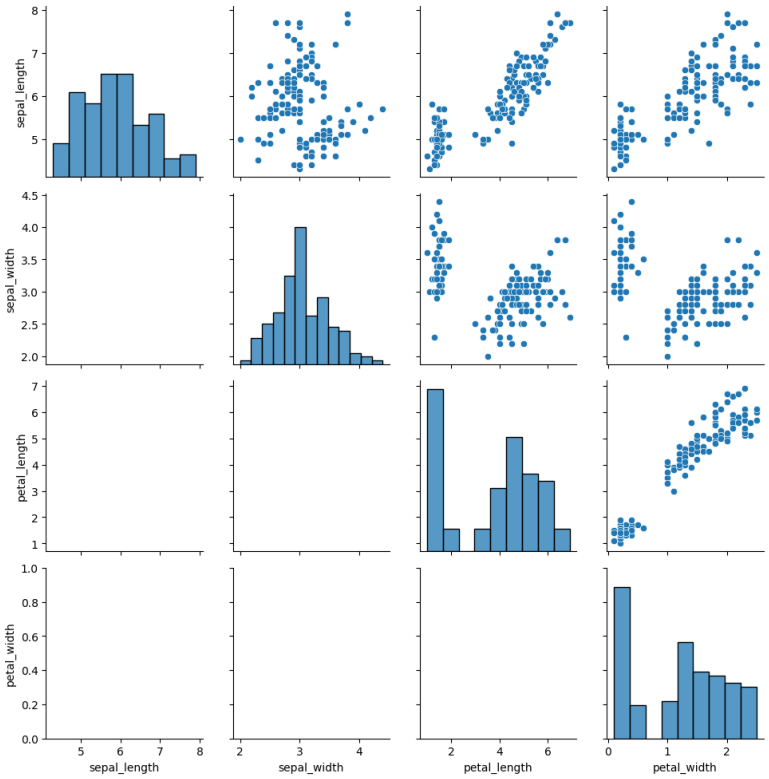
1. PairGrid

Description: PairGrid is a customizable grid that lets you plot pairwise relationships between variables in a dataset. Unlike pairplot, PairGrid offers more control over each plot in the grid, allowing for complex visualizations.

- **Function**: g = sns.PairGrid(data)
- **Usage**: Use PairGrid to create a grid of customized plots between each pair of variables, especially when you want different plot types on the diagonals and off-diagonals.

Code Example:

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

data = sns.load_dataset("iris")

# Create PairGrid
g = sns.PairGrid(data)
g.map_upper(sns.scatterplot)
g.map_diag(sns.histplot)
plt.show()
```



Key Features:

- Customizable plot types for upper, lower, and diagonal sections.
- Allows for detailed exploration of relationships between multiple variables.
- Ideal for datasets with multiple numerical variables.

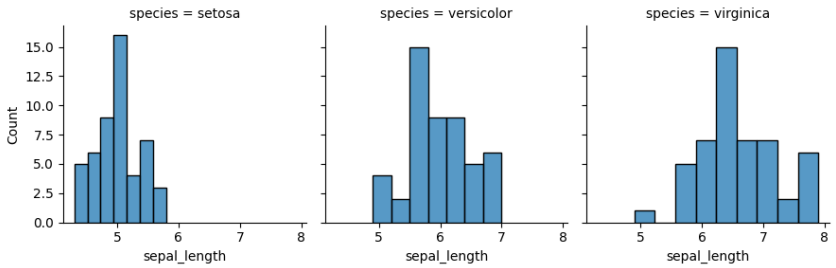
2. FacetGrid

Description: FacetGrid is a powerful tool for plotting subsets of data across multiple categories or values of a variable. It allows you to create a grid of plots, each representing a different subset of the data.

- **Function**: g = sns.FacetGrid(data, col='variable')
- **Usage**: Use FacetGrid when you want to visualize the same plot across different categories or subsets of the data, allowing easy faceting.

Code Example:

```
g = sns.FacetGrid(data, col="species")
g.map(sns.histplot, "sepal_length")
plt.show()
```



Key Features:

- Enables facetting on a variable, with plots displayed across columns or rows.
- Useful for categorical exploration of distributions or patterns.
- Highly customizable, allowing different plot types in each facet.
-

Multivariate Plots

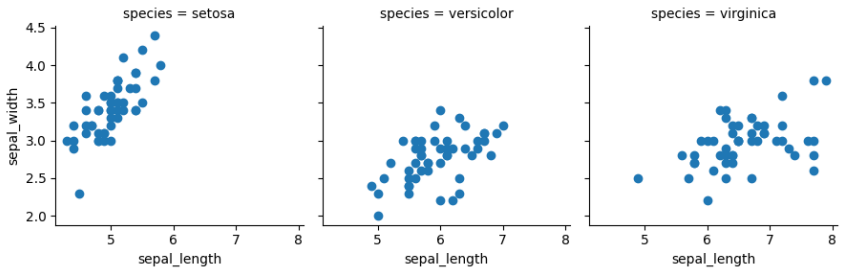
1. FacetGrid with Mapping

Description: Using FacetGrid with mapping allows you to map specific plots, like scatter plots, across grid facets. This is ideal for observing multivariate relationships by specifying plots within each subset.

- **Function**: g.map(plt.scatter, 'x', 'y')
- **Usage**: Use this approach for custom plotting within each subset in a FacetGrid, which can be particularly helpful when exploring interactions between two variables across different categories.

Code Example:

```
g = sns.FacetGrid(data, col="species")
g.map(plt.scatter, "sepal_length", "sepal_width")
plt.show()
```



Key Features:

- Maps a specific plot type, such as scatter, to each subset of data.
- Enables deeper analysis of relationships across different categories.
- Flexible for combining with other plot types.

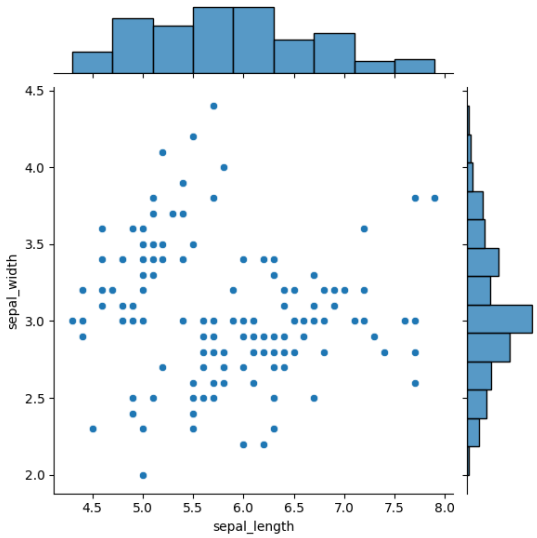
2. JointGrid

Description: JointGrid sets up a grid to visualize the relationship between two variables, displaying a bivariate plot in the center (scatter, hexbin, etc.) and univariate plots (histograms or KDE) on the margins.

- **Function**: g = sns.JointGrid(x='x', y='y', data=data)
- **Usage**: Use JointGrid to analyze bivariate relationships while showing distributions of each variable, allowing for a comprehensive view of variable interactions.

Code Example:

```
g = sns.JointGrid(x="sepal_length", y="sepal_width", data=data)
g.plot(sns.scatterplot, sns.histplot)
plt.show()
```



Key Features:

- Displays bivariate and univariate plots simultaneously.
- Supports various plot types for center (scatter, kde, hex) and margins.
- Ideal for focused analysis of two-variable relationships.

Customizing Visualizations, Saving Plots, and Working with Time Series Data in Seaborn

Seaborn offers numerous options for customizing visualizations, adding details, and saving your work. It also integrates seamlessly with time series data, making it versatile for a wide range of data visualization needs.

Customizing Visualizations

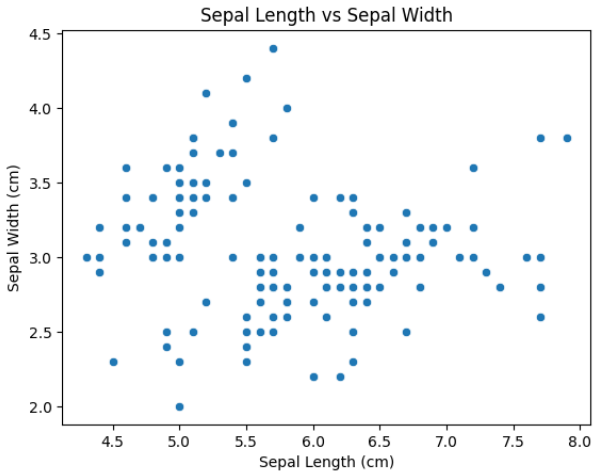
Adding Titles and Labels

To make plots more informative, it's essential to add titles and labels.

- **Title**: plt.title('Title') - Adds a title to your plot.
- **X-axis Label**: plt.xlabel('X-axis Label') - Adds a label to the X-axis.
- **Y-axis Label**: plt.ylabel('Y-axis Label') - Adds a label to the Y-axis.

Code Example:

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


data = sns.load_dataset("iris")

# Plot
sns.scatterplot(x="sepal_length", y="sepal_width", data=data)
plt.title("Sepal Length vs Sepal Width")
plt.xlabel("Sepal Length (cm)")
plt.ylabel("Sepal Width (cm)")
plt.show()
```



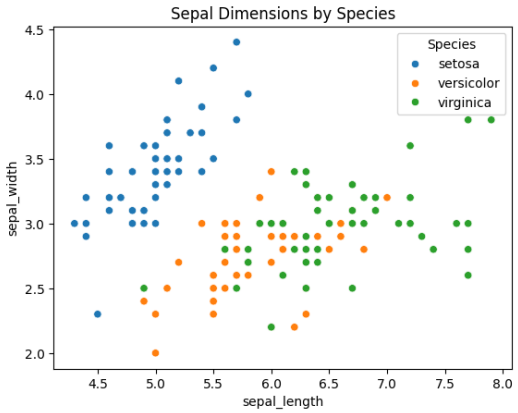
Legends

Legends enhance plot interpretability by labeling colors or symbols associated with categories.

- **Function**: sns.scatterplot(x, y, hue='category')
- **Usage**: Use hue to differentiate categories with colors, automatically generating a legend.

Code Example:

```
sns.scatterplot(x="sepal_length", y="sepal_width", hue="species", data=data)
plt.title("Sepal Dimensions by Species")
plt.legend(title="Species")
plt.show()
```



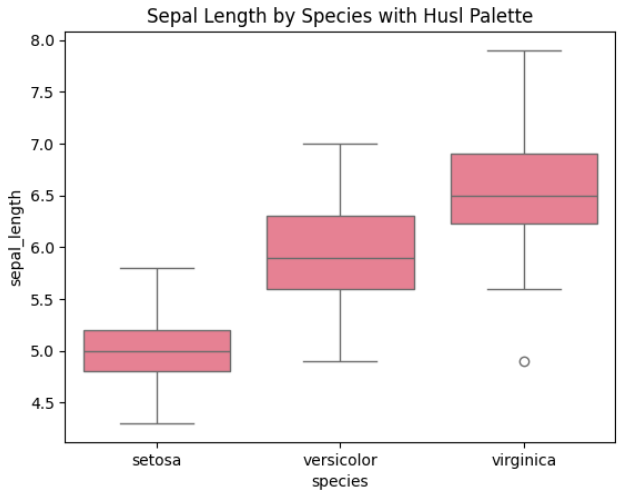
Palette

Seaborn offers several color palettes, which are customizable to improve visual appeal.

- **Function**: sns.set_palette('palette_name')
- **Common Palettes**: 'deep', 'muted', 'bright', 'husl', 'dark'.

Code Example:

```
# Set palette and plot
sns.set_palette("husl")
sns.boxplot(x="species", y="sepal_length", data=data)
plt.title("Sepal Length by Species with Husl Palette")
plt.show()
```



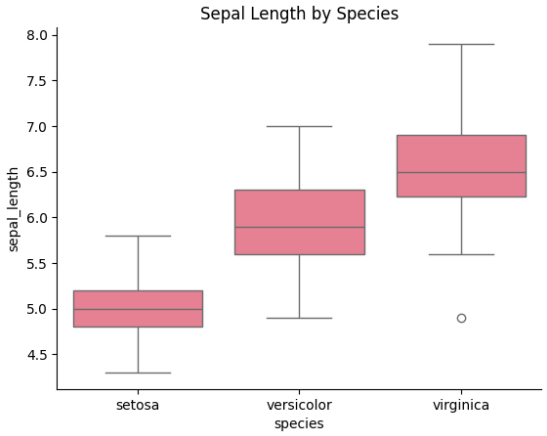
Axis Ticks

Remove or modify axis spines for a cleaner, minimalistic look.

- **Function**: sns.despine()
- **Usage**: Call after plotting to remove top and right spines by default.

Code Example:

```
sns.boxplot(x="species", y="sepal_length", data=data)
plt.title("Sepal Length by Species")
sns.despine()
plt.show()
```



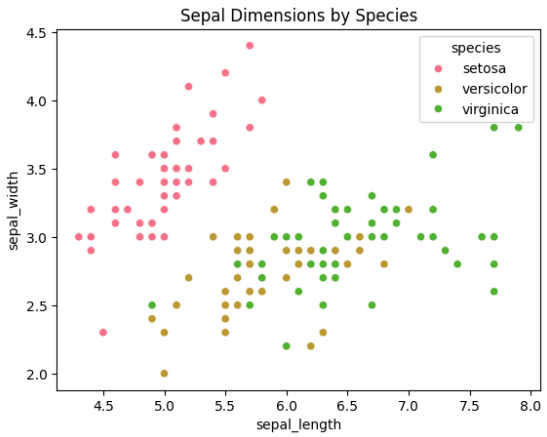
Saving Plots

Seaborn allows you to save plots as image files, making it easy to include visualizations in reports or presentations.

- **Function**: plt.savefig('filename.png')
- **File Types**: You can save files as .png, .jpg, .svg, etc.

Code Example

```
sns.scatterplot(x="sepal_length", y="sepal_width", hue="species", data=data)
plt.title("Sepal Dimensions by Species")
plt.savefig("sepal_scatterplot.png")  # Save plot as PNG
plt.show()
```



Working with Time Series Data

Seaborn can handle time series data, particularly when combined with Pandas DateTime indices. This is especially useful for visualizing trends over time.

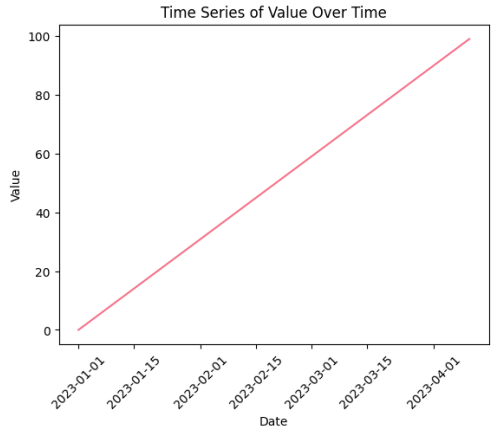
Line Plot with Dates

Line plots are commonly used for time series data to display trends.

- **Function**: sns.lineplot(x='date', y='value', data)
- **Usage**: Load or format data with a DateTime index to display trends effectively.

Code Example:

```
import pandas as pd

# Sample time series data
time_data = pd.DataFrame({
    'date': pd.date_range(start='2023-01-01', periods=100),
    'value': range(100)
})

# Plot
sns.lineplot(x="date", y="value", data=time_data)
plt.title("Time Series of Value Over Time")
plt.xlabel("Date")
plt.ylabel("Value")
plt.xticks(rotation=45)
plt.show()
```



Key Features:

- **Date Formatting**: Use Pandas' pd.to_datetime() to ensure date formatting.
- **Date Rotation**: Rotate x-tick labels if dates overlap.

Statistical Plotting and Seaborn with Matplotlib Integration

Seaborn provides robust tools for statistical visualization, allowing users to explore distributions and relationships in data. Additionally, Seaborn's integration with Matplotlib

enables further customization, making it ideal for building more informative and visually appealing plots.

Statistical Plotting

Statistical plots allow for the detailed examination of distributions and insights, particularly for large datasets.

Boxen Plot

Boxen plots extend the traditional boxplot by providing more details for large datasets, especially at the tails of the distribution.

- **Function**: sns.boxenplot(x, y, data)
- **Usage**: Recommended for datasets where you want to observe data distribution at multiple quantiles.

Code Example:

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


data = sns.load_dataset("diamonds")

# Plot
sns.boxenplot(x="cut", y="price", data=data)
plt.title("Price Distribution Across Diamond Cuts")
plt.xlabel("Cut")
plt.ylabel("Price")
plt.show()
```

Violin Plot

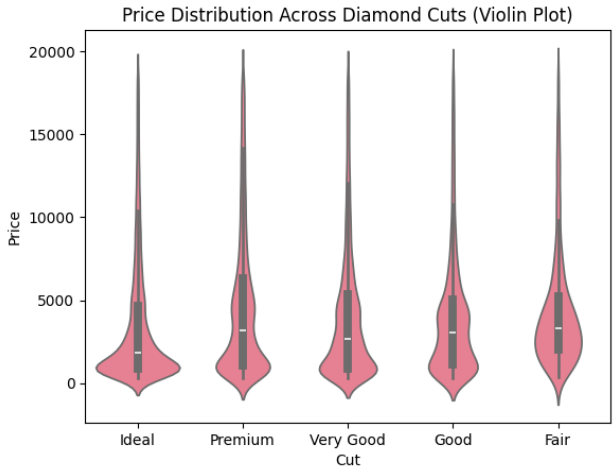
A violin plot combines aspects of a boxplot and a kernel density plot, showing the distribution of the data across several levels.

- **Function**: sns.violinplot(x, y, data)

- **Usage**: Useful for visualizing the distribution of data with an estimation of density.

Code Example:

```
# Plot
sns.violinplot(x="cut", y="price", data=data)
plt.title("Price Distribution Across Diamond Cuts (Violin Plot)")
plt.xlabel("Cut")
plt.ylabel("Price")
plt.show()
```



Seaborn with Matplotlib Integration

Seaborn is built on top of Matplotlib, which means that you can use Matplotlib functions to further customize Seaborn plots. This integration is useful when you need control over elements like titles, labels, legends, and annotations.

Combining Matplotlib and Seaborn

With Seaborn, you can set up plots and then refine them with Matplotlib functions, allowing for a wider range of customizations.

Adding Titles and Labels with Matplotlib

- **Functions**: plt.title(), plt.xlabel(), plt.ylabel()
- **Example**: Add informative titles, labels, or subtitles to your plots.

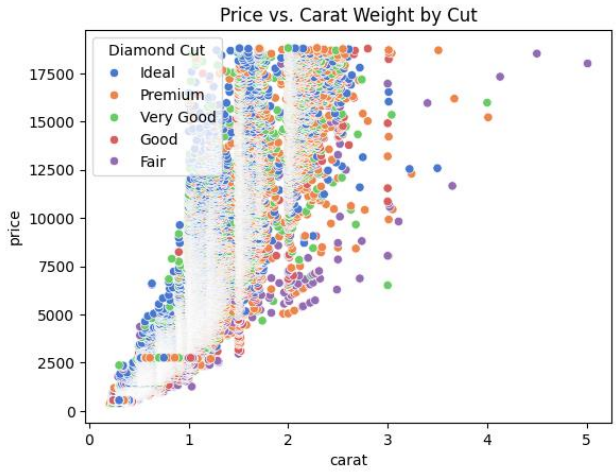
Customizing Legends and Colors

You can use plt.legend() to modify the legend or set a custom color palette.

- **Legend**: Control position, title, or labels.
- **Palette**: Set a color palette with sns.set_palette().

Code Example:

```
sns.set_palette("muted")

sns.scatterplot(x="carat", y="price", hue="cut", data=data)
plt.title("Price vs. Carat Weight by Cut")
plt.legend(title="Diamond Cut", loc='upper left')
plt.show()
```



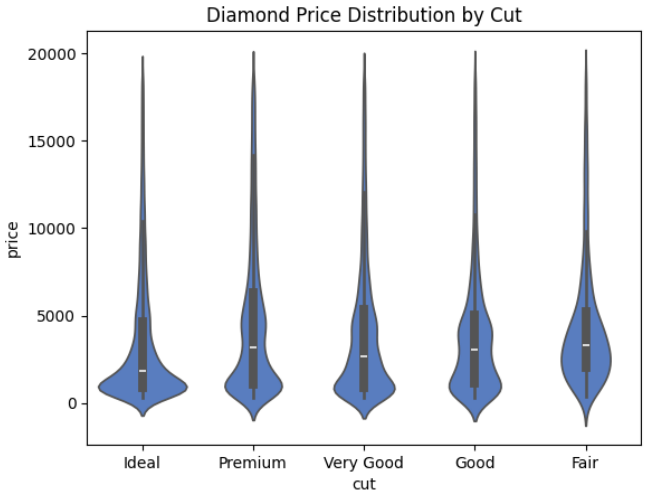
Saving Figures with Matplotlib

To save any Seaborn plot as an image, use Matplotlib's plt.savefig().

- **Function**: plt.savefig('filename.png')
- **Usage**: Specify file type, resolution, and file name.

Code Example

```
# Create plot
sns.violinplot(x="cut", y="price", data=data)
plt.title("Diamond Price Distribution by Cut")

# Save figure
plt.savefig("diamond_price_violinplot.png", dpi=300)
plt.show()
```


Diamond Price Distribution by Cut
price
20000
15000
10000
5000
0
Ideal
Premium
Very Good
Good
Fair
cut