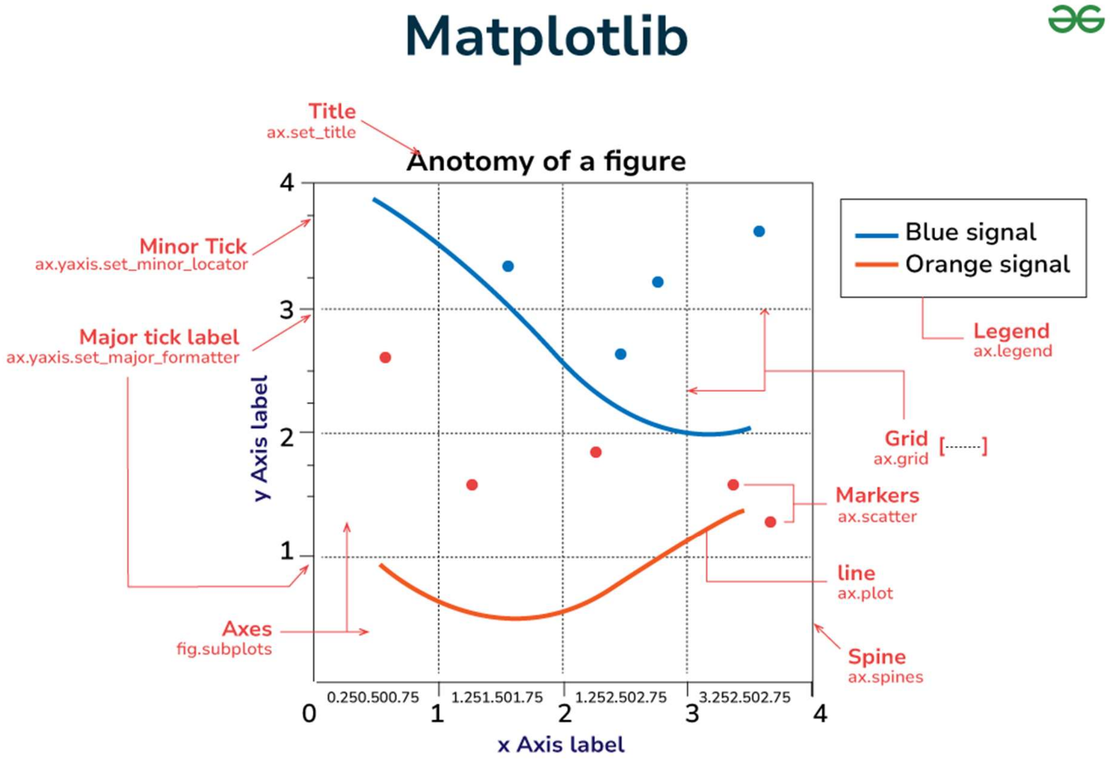
Unit 3

Introduction to Matplotlib:

- Matplotlib is a low-level graph plotting library in python that serves as a visualization utility.
- Matplotlib was created by John D. Hunter.
- Matplotlib is open source and we can use it freely.
- Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.



- **Installation of Matplotlib**

pip install matplotlib

- **Import it in Your Applications**

import matplotlib

- **Check Version**

print(matplotlib.__version__)

- **Pyplot**

import matplotlib.pyplot as plt

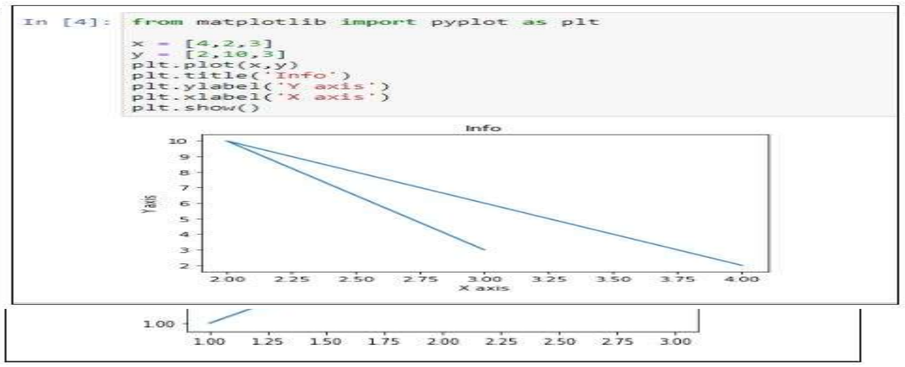
Two ways of creating plots

There are two main ways of creating plots in matplotlib:

1. matplotlib.pyplot.plot() - is recommended for simple plots (e.g. x and y)
2. matplotlib.pyplot.XX (where XX can be one of many methods, this is known as the object-oriented API) - is recommended for more complex plots (for example plt.subplots() can be used to create multiple plots on the same Figure, but we'll get to this later)

Both of these methods are still often created by building off import matplotlib.pyplot as plt as a base.

Getting Started With Pyplot: Pyplot is a Matplotlib module that provides simple functions for adding plot elements, such as lines, images, text, etc. to the axes in the current figure.Let's begin our tutorial with a simple graph that uses fundamental Matplotlib code in Jupyter Notebook.

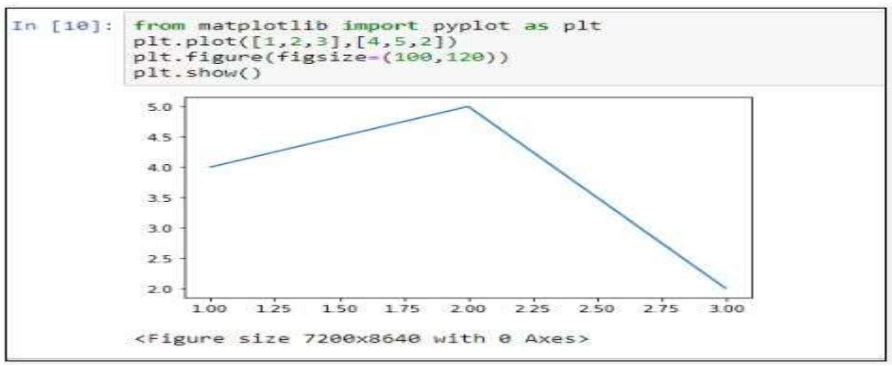


```
In [4]: from matplotlib import pyplot as plt

        x = [4,2,3]
        y = [2,10,3]
        plt.plot(x,y)
        plt.title('Info')
        plt.ylabel('Y axis')
        plt.xlabel('X axis')
        plt.show()
```

Note that the first array appears on the x-axis, and the second array appears on the y-axis of the plot.Let us now see how we can add a title, as well as the x-axis and y-axis names, using the title(), xlabel(), and ylabel() methods, respectively.

Users can also specify the size of the figure using the figure() method. Additionally, users can pass values as tuples, which make up the length of rows

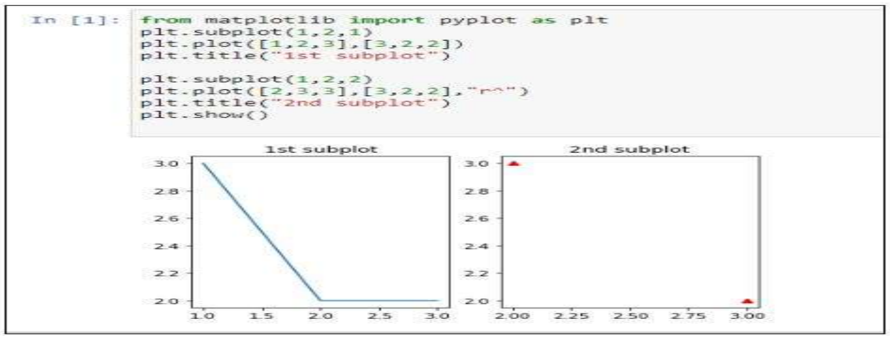


```
In [10]: from matplotlib import pyplot as plt
         plt.plot([1,2,3],[4,5,2])
         plt.figure(figsize=(100,120))
         plt.show()
```

```
<Figure size 7200x8640 with 0 Axes>
```

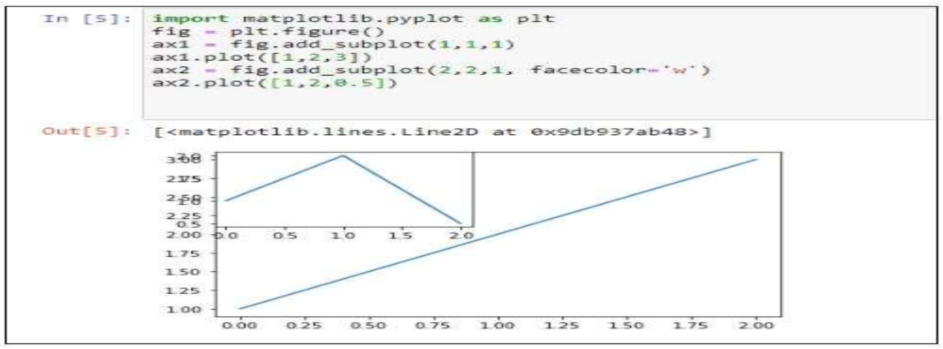
and columns to the argument figsize.

MatplotlibSubplot: You can use the subplot() method to add more than one plot in a figure. Syntax: plt.subplots(nrows, ncols, index)
The three-integer arguments specify the number of rows and columns and the index of the subplot grid.



```
In [1]: from matplotlib import pyplot as plt
        plt.subplot(1,2,1)
        plt.plot([1,2,3],[3,2,2])
        plt.title("1st subplot")

        plt.subplot(1,2,2)
        plt.plot([2,3,3],[3,2,2],"r^")
        plt.title("2nd subplot")
        plt.show()
```



```
In [5]: import matplotlib.pyplot as plt
        fig = plt.figure()
        ax1 = fig.add_subplot(1,1,1)
        ax1.plot([1,2,3])
        ax2 = fig.add_subplot(2,2,1, facecolor='w')
        ax2.plot([1,2,0.5])

Out[5]: [<matplotlib.lines.Line2D at 0x9db937ab48>]
```

The add_subplot() function of the figure class enables us to add a graph inside a graph.
# First create a grid of plots
fig, ax = plt.subplots(2,2,figsize=(10,6)) #this will create the subplots with 2 rows and 2 columns
#and the second argument is size of the plot
# Lets plot all the figures
ax[0][0].plot(x1, np.sin(x1), 'g') #row=0,col=0
ax[0][1].plot(x1, np.cos(x1), 'y') #row=0,col=1
ax[1][0].plot(x1, np.sin(x1), 'b') #row=1,col=0
ax[1][1].plot(x1, np.cos(x1), 'red') #row=1,col=1

plt.tight_layout()
#show the plots
plt.show()

Different categories of plots that Matplotlib provides.

- Line plot
- Histogram
- Bar Chart
- Scatter plot
- Pie charts
- Boxplot

import matplotlib.pyplot

Line Plots:A line plot shows the relationship between the x and y-axis.
The plot() function in the Matplotlib library's Pyplot module creates a 2D hexagonal plot of x and y coordinates. plot() will take various arguments like plot(x, y, scalex, scaley, data, **kwargs).
x, y are the horizontal and vertical axis coordinates where x values are optional, and its default value is range(len(y)).
scalex, scaley parameters are used to autoscale the x-axis or y-axis, and its default value is actual.
**kwargs specify the property like line label, linewidth, marker, color, etc.
this line will create array of numbers between 1 to 10 of length 100

np.linspace(satrt,stop,num)
x1 = np.linspace(0, 10, 100) #line plot
plt.plot(x1, np.sin(x1), '-',color='orange')
plt.plot(x1, np.cos(x1), '--',color='b')
give the name of the x and y axis
plt.xlabel('x label')
plt.ylabel('y label')
also give the title of the plot
plt.title("Title")
plt.show()

Histogram: The most common graph for displaying frequency distributions is a histogram. To create a histogram, the first step is to create a bin of ranges, then distribute the whole range of values into a series of intervals and count the value that will fall in the given interval. We can use plt.Hist () function plots the histograms, taking various arguments like data, bins, color, etc.
x: x-coordinate or sequence of the array

bins: integer value for the number of bins wanted in the graph
range: the lower and upper range of bins
density: optional parameter that contains boolean values
histtype: optional parameter used to create different types of histograms like:-bar bar stacked, step, step filled, and the default is a bar
#draw random samples from random distributions.
x = np.random.normal(170, 10, 250)
#plot histograms
plt.hist(x) plt.show()

Bar Plot: Mainly, the barplot shows the relationship between the numeric and categoric values. In a bar chart, we have one axis representing a particular category of the columns and another axis representing the values or counts of the specific category. Barcharts can be plotted both vertically and horizontally using the following line of code:
plt.bar(x,height,width,bottom,align)
x: representing the coordinates of the x-axis
height: the height of the bars
, width: width of the bars. Its default value is 0.8
bottom: It's optional. It is a y-coordinate of the bar. Its default value is None
align: center, edge its default value is center
#define array
data= [5. , 25. , 50. , 20.]
plt.bar(range(len(data)), data,color='c')
plt.show()

Scatter Plot: Scatter plots show the relationships between variables by using dots to represent the connection between two numeric variables.
The scatter() method in the Matplotlib library is used for plotting.
#create the x and y axis coordinates
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)
plt.legend()
plt.show()

Pie Chart:A pie chart (or circular chart) is used to show the percentage of the whole. Hence, it is used to compare the individual categories with the whole. **Pie()** will take the different parameters such as:
x: Sequence of an array
labels: List of strings which will be the name of each slice in the pie chart

Autopct: It is used to label the wedges with numeric values. The labels will be placed inside the wedges. Its format is %1.2f%

```
#define the figure size
plt.figure(figsize=(7,7))
x = [25,30,45,10]
#labels of the pie chart
labels = ['A','B','C','D']
plt.pie(x, labels=labels)
plt.show()
```

Box Plot:A Box plot in Python Matplotlib showcases the dataset's summary, encompassing all numeric values. It highlights the minimum, first quartile, median, third quartile, and maximum. The median lies between the first and third quartiles. On the x-axis, you'll find the data values, while the y-coordinates represent the frequency distribution.
Parameters used in box plots are as follows:
data: NumPy array
vert: It will take boolean values, i.e., true or false, for the vertical and horizontal plot. The default is True
width: This will take an array and sets of the width of boxes. Optional parameters
Patch_artist: It is used to fill the boxes with color, and its default value is false
labels: Array of strings which is used to set the labels of the dataset

```
#create the random values by using numpy
values= np.random.normal(100, 20, 300)
#creating the plot by boxplot() function which is avilable in matplotlib
plt.boxplot(values,patch_artist=True,vert=True)
plt.show()
```

Area Chart: An area chart or plot visualizes quantitative data graphically based on the line plot. fill_between() function is used to plot the area chart.
Parameter:
x,y represent the x and y coordinates of the plot. This will take an array of length n.
Interpolate is a boolean value and is optional. If true, interpolate between the two lines to find the precise point of intersection.
****kwargs**: alpha, color, face color, edge color, linewidth.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
y = [2, 7, 14, 17, 20, 27, 30, 38, 25, 18, 6, 1]
#plot the line of the given data
```

```
plt.plot(np.arange(12),y, color="blue", alpha=0.6, linewidth=2)
#decorate thw plot by giving the labels
plt.xlabel('Month', size=12)
plt.ylabel('Turnover(Cr.)', size=12) #set y axis start with zero
plt.ylim(bottom=0)
plt.show()
```

3-D Graphs: Now that you have seen some simple graphs, it's time to check some complex ones, i.e., 3-D graphs. Initially, Matplotlib was built for 2-dimensional graphs, but later, 3-D graphs were added. Let's check how you can plot a 3-D graph in Matplotlib.

```
from mpl_toolkits import mplot3d
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(projection='3d')
```

The above code is used to create the 3-dimensional axes.
Each plot that we have seen in 2-D plotting through Matplotlib can also be drawn as 3-D graphs. For instance, let's check a line plot in a 3-D plane.

```
ax = plt.axes(projection='3d')
# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')
# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

all other types of graphs can be drawn in the same way. One particular graph that Matplotlib 3-D provides is the Contour Plot. You can draw a contour plot using the following link:

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe');
```

Matplotlibworkflow:Saving plots

The following workflow is a standard practice when creating a matplotlib plot:

1. **Import** `matplotlib` - For example, `import matplotlib.pyplot as plt`

2. **Prepare data -** This may be from an existing dataset (data analysis) or from the outputs of a machine learning model (data science)
3. **Setup the plot -** In other words, create the Figure and various Axes
4. **Plot data to the Axes -** Send the relevant data to the target Axes
5. **Customize the plot -** Add a title, decorate the colors, label each Axis
6. **Save and show -** See what your masterpiece looks like and save it to file if necessary

```
# A matplotlib workflow
# 0. Import and get matplotlib ready
# %matplotlib inline # Not necessary in newer versions of Jupyter (e.g. 2022
onwards)
import matplotlib.pyplot as plt
# 1. Prepare data
x = [1, 2, 3, 4]
y = [11, 22, 33, 44]

# 2. Setup plot (Figure and Axes)
fig, ax = plt.subplots(figsize=(10,10))
# 3. Plot data
ax.plot(x, y)
# 4. Customize plot
ax.set(title="Sample Simple Plot", xlabel="x-axis", ylabel="y-axis")
# 5. Save & show
fig.savefig("../images/simple-plot.png")
```

Controlling Axes :

Truncating or expanding some plot boundaries is an essential feature in matplotlib, allowing us to be more creative and generate various inferences.
Axes can be positioned for the plot at any location in the figure. One figure can have several axes, although only one can include a certain axis object.
We can scale our plots more accurately by raising or lowering the scales by setting the axis range in our plots.

How to Set Axis Range in Matplotlib?

a) Set Axis Range in a plot

Using xlim() and ylim()

To demonstrate this example, we will put a custom range for a sine curve in matplotlib.

```
# import packages
import matplotlib.pyplot as plt
import numpy as np
# return values between 0 and 10, with a space of 0.1
x = np.arange(0, 10, 0.1)
```

```
# generate the value of the cos function for given x values
y = np.sin(x)
# plot graph of the sine function
plt.plot(y, color='blue')
plt.xlim(0, 50)
plt.ylim(0, 1)
# display plot
plt.show()
```

Explanation:In this code, we use the arange() function to return values between 0 and 10 and have a contiguous space of 0.1. After creating the sine curve, we change the range of axes by using the xlim() and ylim() functions.

Using set_xlim() and set_ylim() functions

Another way to set the range of axes is using the set_xlim() and set_ylim() functions.

These functions don't directly utilize the pyplot module like xlim() and ylim(), they are used with the axes of our plots.

```
# import packages
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
# return values between 0 and 10, with a space of 0.1
x = np.arange(0, 10, 0.1)
# generate the value of the cos function for given x values
y = np.sin(x)
# plot graph of the sine function
plt.plot(y, color='blue')
ax.set_xlim(10, 50)
ax.set_ylim(0, 1)
# display plot
plt.show()
```

Using axis() method:Using the axis() method, we can instantiate the x-axis and y-axis simultaneously.To initialize our axes simultaneously, we need to pass our axes as a list in the axis() function. In the example below, we pass our axes ((0, 5) and (1, 15)) as a list.

```
import matplotlib.pyplot as plt
x =[0, 1, 2, 3, 4, 5]
y =[0, 2, 4, 6, 8, 10]
# Plotting the graph
plt.plot(x, y)
# Setting the x-axis to 0-5
# and y-axis to 1-15
plt.axis([0, 5, 1, 15])
# Showing the graph with an updated axis
```

```
plt.show()
```

b) Set Axis Range in a Subplot:A subplot is essentially one of several plots on the same figure. To set custom ranges, we use the functions twice.

```
# Import Libraries
import matplotlib.pyplot as plt
import numpy as np
# plot 1:
plt.subplot(1, 2, 1)
# Data for plot 1
x = np.arange(0 , 8, 0.2)
y = np.sin(x)
# Plotting
plt.plot(x, y)
# plot 2:
plt.subplot(1, 2, 2)
# Data for plot 2
x = np.arange(0 , 10 ,0.2)
y = np.cos(x)
# Set axes for the specific subplot
plt.xlim(2, 8)
plt.ylim(-0.50, 1.5)
# Plotting
plt.plot(x,y)
# Auto adjust
plt.tight_layout()
# Display
plt.show()
```

Explanation:We create two subplots in a single frame, a sine curve, and a cosine curve respectively. After creating the curves, we use the xlim() and ylim() functions to set the ranges of the X and Y axes, respectively. Here, the range for the x-axis is from 2 to 8, and for the y-axis is -0.5 to 1.5.

c) Set Axis Range in a Scatter plot: In this example, we will plot a scatter graph. After plotting, we will alter the original axes using the functions we went through in this article.

```
import numpy as np
import matplotlib.pyplot as plt
#Generating sample data with random samples from the standard normal distribution
A = np.random.standard_normal((100, 2))
A += np.array((-1, -1))
B = np.random.standard_normal((100, 2))
B += np.array((1, 1))
#Scatter plot generated with random values
```

```
plt.scatter(B[:,0], B[:,1], c = 'k', s = 75.)
plt.scatter(A[:,0], A[:,1], c = 'b', s = 25.)
plt.xlim(1, 2)
plt.ylim(0, 1)
plt.show()
```

Explanation:In this example, we have created a scatter plot using random numbers. After this, we use the xlim() and ylim() functions to set the ranges.

d) Set Axis Range in a Datetime plot: A plot in which either the x-axis is in a datetime format, or the y-axis is called a datetime plot.Now, we will understand how to set the axis range of the datetime plot using matplotlib.

```
import datetime
import matplotlib.pyplot as plt
# Create subplot
fig, ax = plt.subplots()
# Define Data
x = [datetime.date(2022, 1, 15)] * 5
y = [2, 4, 1, 6, 8]
# Set axes
ax.set_xlim([datetime.date(2022, 1, 1), datetime.date(2022, 2,  1)])
ax.set_ylim([0, 10])
# Plot date
ax.plot_date(x, y, markerfacecolor='r', markeredgecolor='k',  markersize= 15)
# Auto format
fig.autofmt_xdate()
# Display
plt.show()
```

Explanation:In this example, our x-axis ranges from 1-1-22 to 1-2-2022. We use the set_xlim() and set_ylim() to alter the original range.

e) Set Axis Range using imshow() method: To change our axes using imshow() function, we need to pass an extra argument called extent.By using extent, we define all of our axes in a single line of code. In the example below, the axes are: (-1 to 1 for the x-axis, and -1 to 1 for the y-axis).

```
# Import Library
import numpy as np
import matplotlib.pyplot as plt
# Define Data
x = np.arange(400).reshape((20,20))
# Imshow set axes
plt.imshow(x , extent=[-1,1,-1,1])
# Add Title
plt.title( "Imshow Plot" )
# Display
plt.show()
```

Explanation Using the imshow() method, we'll change our axes from -1 to 1.

Annotation:

Text:You can annotate any point in your chart with text using the annotate() function. The function parameters used in the example below are:

text : The text of the annotation

xy : The point (x,y) to annotate

xytext : The position (x,y) to place the text at (If None, defaults to xy)

arrowprops : The properties used to draw an arrow between the positions xy and xytext

```
# Libraries
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
#Data
df=pd.DataFrame({'x_pos':range(1,101),'y_pos':np.random.randn(100)*15+rang
e(1,101) })
# Basic chart
plt.plot('x_pos', 'y_pos', data=df,  linestyle='none', marker='o')
# Annotate with text + Arrow
plt.annotate('This point is interesting!', xy=(25, 50), xytext=(0, 80),
arrowprops=dict(facecolor='black', shrink=0.05))
# Show the graph
plt.show()
```

Math:You can add a text of mathematical expression to your plot with text() function:

```
# Library
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
# plot
df=pd.DataFrame({'x_pos':range(1,101),'y_pos':np.random.randn(100)*15+rang
e(1,101) })
plt.plot( 'x_pos', 'y_pos', data=df, linestyle='none', marker='o')
# Annotation
plt.text(40, 0, r'equation: $\sum_{i=0}^\infty x_i$', fontsize=20)
# Show the graph
plt.show()
```

Rectangle: You can use the add_patch() function to add a matplotlib patch to the axes. In the example below, you will see how to add a rectangle. You can see the list of patches here.

```
# libraries
import matplotlib.patches as patches
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
df=pd.DataFrame({'x_pos':range(1,101),'y_pos':np.random.randn(100)*15+rang
e(1,101) })
 fig1 = plt.figure()
ax1 = fig1.add_subplot(111)
ax1.plot( 'x_pos', 'y_pos', data=df, linestyle='none', marker='o')
# Add rectangle
ax1.add_patch(patches.Rectangle((20, 25), # (x,y)50, # width50, # height
alpha=0.3, facecolor="red", edgecolor="black", linewidth=3, linestyle='solid'))
# Show the graph
plt.show()
```

<u>**Ticks in Matplotlib**</u>: Ticks are the value on the axis to show the coordinates on the graph. It is the value on the axes by which we can visualize where will a specific coordinate lie on a graph. Whenever we plot a graph, ticks values are adjusted according to the data, which is sufficient in common situations, but it is not ideal whenever we plot data on a graph. There are many ways to customize the tick labels like matplotlib.pyplot.xticks(), ax.set_xticklabels(), matplotlib.pyplot.setp() and ax.tick_params().
<u>How to Add Ticks in Matplotlib</u>? By default, ticks are generated automatically when plotting data in matplotlib. First, we discuss the parameter and syntax of the function related to ticks.
Syntax:

- matplotlib.pyplot.xticks(ticks=None, labels=None, **kwargs)
- Axes.set_xticklabels(labels, *, fontdict=None, minor=False, **kwargs)
- Axes.tick_params(axis='both', **kwargs)

Parameters:
matplotlib.pyplot.xticks()

- ticks (optional parameter): The list of tick location
- label (optional parameter): The labels to place at a given ticks location.
- **kwargs: Text properties can be used to control the appearance of the labels.

Axes.set_xticklabels()

- label (optional parameter): The labels to place at a given ticks location.
- fontdict (optional parameter): A dictionary controls the appearance of the tick labels.
- minor (optional parameter): Whether to set the minor tick labels rather than the major ones.
- kwargs: Text properties can be used to control the appearance of the labels.

Axes.tick_params()

- axis: The axis to which parameters are applied
- kwargs: Text properties can be used to control the appearance of the labels.

How to Customize Axis Tick Labels in Matplotlib?

Sometimes we need to change the properties of the tick labels on the axis to make it more readable and clear, like we can change color, font size, etc.

Font Size of Tick Labels

We can change the font size of the tick labels by using these functions.

```
matplotlib.pyplot.xticks(fontsize= )
ax.set_xticklabels(xlabels, Fontsize= )
matplotlib.pyplot.setp(ax.get_xticklabels(), Fontsize=)
ax.tick_params(axis='x', Labelsize= )
```

Example1: Changing tick labels fontsize using xticks()

```
import matplotlib.pyplot as plt
import random
plt.rcParams['figure.figsize']=(10,6)
x=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
y= [random.randint(1,10) for i in range(15)]
plt.plot(x,y)
plt.xlabel('x-axis')
plt.ylabel('y-axis')
# Changing the fontsize of tick labels using xticks() function
plt.xticks(fontsize=20)
plt.grid()
plt.show()
```

Color of Tick Labels: We can also change the color of the tick labels. There are three ways to do it.

```
matplotlib.pyplot.xticks(color= )
ax.set_xticklabels(xlabels, color= )
ax.tick_params(axis='x', colors= )
```

Example1: Changing color of tick labels using sticks()

```
import matplotlib.pyplot as plt
import random
# Adjusting figure size
plt.rcParams['figure.figsize']=(10,6)
# generating list of random numbers
x=[random.randint(1,10) for i in range(15)]
y= [random.randint(1,10) for i in range(15)]
# Plotting the scatter plot with given coordinates
plt.scatter(x,y)
plt.xlabel('x-axis')
plt.ylabel('y-axis')
# Changing the color of tick labels using xticks() function
plt.xticks( color='blue')
plt.grid()
plt.show()
```

<u>Matplotlib confriguration using different plot styles:</u> Styling in Matplotlib allows you to change the overall appearance of your plots quickly and consistently. Whether you're preparing figures for a publication, presentation, or just want to maintain a uniform style across multiple plots, Matplotlib's styling capabilities make it easy to achieve the desired look.

1. Using Built-in Styles with plt.style.use()

Matplotlib comes with a variety of built-in styles that can be applied to your plots with a single line of code. These styles can dramatically change the look and feel of your plots, making them more suitable for different purposes like presentations, reports, or technical papers.

Applying a Built-in Style

To use a built-in style, simply call plt.style.use() with the name of the style:

```
import matplotlib.pyplot as plt
# Apply a built-in style
plt.style.use('ggplot')
x = [1, 2, 3, 4, 5]
y = [10, 20, 15, 25, 30]
plt.plot(x, y)
plt.title('Sales Over Time')
plt.xlabel('Time (months)')
plt.ylabel('Sales (units)')
plt.show()
```

In this example, we applied the 'ggplot' style, which gives the plot a distinctive appearance inspired by the popular R package ggplot2. Matplotlib includes several other styles, such as seaborn, fivethirtyeight, dark_background, and more.

<u>Listing All Available Styles:</u>You can list all the available styles in Matplotlib using the following code:

```
print(plt.style.available)
```

2. Creating Custom Styles: While built-in styles are convenient, there might be cases where you need a specific look that isn't covered by the default options. In such cases, you can create your own custom style by defining a set of rcParams (runtime configuration parameters) that control various aspects of your plot's appearance.

Defining Custom Styles: You can define a custom style by setting rcParams directly in your script:

```
plt.rcParams['axes.facecolor'] = 'lightgray'
plt.rcParams['axes.edgecolor'] = 'black'
plt.rcParams['axes.grid'] = True
plt.rcParams['grid.color'] = 'white'
plt.rcParams['grid.linestyle'] = '--'
plt.rcParams['font.size'] = 14
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.color'] = 'blue'
plt.plot(x, y)
plt.title('Custom Styled Plot')
plt.xlabel('Time (months)')
plt.ylabel('Sales (units)')
plt.show()
```

In this example, we customized the plot's background color, gridlines, font size, and line properties.

3. Saving and Sharing Styles: Once you've created a custom style that you like, you might want to save it for reuse in other projects or share it with colleagues. Matplotlib allows you to save styles in a .mplstyle file, which can be loaded and applied just like the built-in styles.

Saving a Custom Style: To save a custom style, create a .mplstyle file and write your rcParams into it. For example, create a file named my_custom_style.mplstyle with the following content:

```
axes.facecolor: lightgray
axes.edgecolor: black
axes.grid: True
grid.color: white
grid.linestyle: --
font.size: 14
lines.linewidth: 2
lines.color: blue
```

Loading and Applying a Custom StyleYou can apply the saved custom style by specifying the path to the .mplstyle file:

```
plt.style.use('my_custom_style.mplstyle')
plt.plot(x, y)
plt.title('Plot with Custom Style')
plt.xlabel('Time (months)')
plt.ylabel('Sales (units)')
plt.show()
```

This makes it easy to maintain a consistent look across multiple projects or share your style with others.

Sharing Your Custom Style: If you want to share your custom style with others, simply distribute the .mplstyle file. Others can place the file in their Matplotlib style directory or use the full path when applying it with plt.style.use().

Seaborn: Python Seaborn library is a widely popular data visualization library that is commonly used for data science and machine learning tasks. You build it on top of the matplotlib data visualization library and can perform exploratory analysis. You can create interactive plots to answer questions about your data.

<u>Type of Functions :</u>
In Seaborn, there are two levels at which you can create plots: Figure-level functions and Axis-level functions.
1. Figure-level functions:
— These are high-level functions that create an entire figure, including one or more subplots.
— They are designed for creating complex visualizations with multiple subplots.
— Examples of figure-level functions include sns.relplot() for relational plots (scatter plots, line plots, etc.) and sns.catplot() for categorical plots (bar plots, box plots, etc.).
2. Axis-level functions:
— These are lower-level functions that create a single plot on an existing set of axes.
— They are more focused on the individual plot itself and are useful for customizing specific aspects of a plot.
— Examples of axis-level functions include sns.scatterplot()for scatter plots and sns.lineplot() for line plots.
Main Classification:
- Relational PlotRelational plots in Seaborn are used to visualize the relationship between two or more variables in a dataset. They help us understand how one variable changes with respect to another. Some common types of relational plots include:

1. Scatter Plots(sns.scatterplot)
2. Line Plots(sns.lineplot())
3. Relational Plots with Multiple Variables (sns.relplot()): Seaborn also provides functions like sns.relplot() which can handle multiple variables and create more complex plots, like scatter plots with additional features like color or size to represent additional information.
   # relplot -> figure level

sns.relplot(data=tips, x='total_bill', y='tip',hue='sex', style='time', size='size', kind='scatter')
- Distribution Plot: Distribution plots in Seaborn are used to visualize the distribution of a univariate dataset. They help us understand the spread, central

tendency, and shape of the data.Here are some common types of distribution plots in Seaborn:

1. Histograms (sns.histplot()):A histogram is a graphical representation of the distribution of a dataset. It displays the frequency or count of different values or ranges of values within the dataset, organizing them into discrete bins or intervals along the horizontal axis. The vertical axis represents the frequency or count of occurrences for each bin. This visualization provides a visual summary of the underlying data, allowing one to quickly grasp the central tendency, spread, and shape of the distribution.
   ```
   # figure level -> displot
   # axes level -> histplot -> kdeplot -> rugplot
   sns.histplot(data=tips, x='total_bill')
   plt.show()
   sns.displot(data=tips, x='total_bill', kind='hist')
   plt.show()
   ```
2. Kernel Density Estimation (KDE) plots (sns.kdeplot()):When you use kdeplot, it takes a dataset as input and produces a KDE plot, which is essentially a smoothed version of a histogram. This plot gives you a more continuous view of the data distribution, making it useful for visualizing probability density.
3. Rug plots (sns.rugplot()):In Seaborn, a rug plot is a simple one-dimensional scatter plot that displays individual data points along a single axis. It is often used in conjunction with other visualizations like histograms or density plots to provide a more detailed view of the distribution of data points. In a rug plot, a small vertical tick, or “rug”, is drawn for each data point at its respective location on the axis. This can be particularly helpful for visualizing the density and spacing of data points, especially when dealing with a relatively small dataset.

- Categorical Plot: Categorical Scatter Plot

- Stripplot: A strip plot is a type of categorical scatter plot that displays individual data points along a categorical axis. It is particularly useful for visualizing the distribution of data points within different categories.
  ```
  sns.catplot(data=tips, x='day',y='total_bill',kind='strip',jitter = 0.2)
  sns.stripplot(data=tips,x='day',y='total_bill',jitter = 0.2)
  plt.show()
  ```
- Swarmplot: A swarm plot is a categorical scatter plot that displays individual data points along a category axis. It is particularly useful for visualizing the distribution of data points within different categories, especially when dealing with relatively small datasets.
  ```
  # Axes Level Function
  sns.swarmplot(data=tips, x='day',y='total_bill')
  #Figure level function
  ```

```
sns.catplot(data=tips, x='day',y='total_bill',kind='swarm')
plt.show()
```

Categorical Distribution Plots

- Boxplot: A boxplot is a standardized way of displaying the distribution of data based on a five-number summary (“minimum”, first quartile [Q1], median, third quartile [Q3] and “maximum”). It can tell you about your outliers and what their values are. Boxplots can also tell you if your data is symmetrical, how tightly your data is grouped and if and how your data is skewed.
  ```
  # Axes plot
  sns.boxplot(data=tips,x='day',y='total_bill')
  # Using catplot
  sns.catplot(data=tips,x='day',y='total_bill',kind='box')
  ```
- Violinplot: A violin plot is a data visualization in Seaborn that combines elements of a box plot and a kernel density plot. It provides a comprehensive view of the distribution of numerical data across different categories or groups.
  ```
  # Axes plot
  sns.violinplot(data=tips,x='day',y='total_bill')
  # Using catplot
  sns.catplot(data=tips,x='day',y='total_bill',kind='violin')
  ```

Categorical Estimate Plot -> for central tendency

- Barplot: A bar plot is a visualization that represents categorical data with rectangular bars. The height of each bar corresponds to the value of the category it represents. Bar plots are used to display and compare the values of different categories or groups, making them particularly useful for summarizing and visualizing data distributions.
  ```
  #Axes plot
  sns.barplot(data=tips, x='sex', y='total_bill',ci=None)
  ```
- Pointplot: A pointplot is a statistical visualization that displays the estimated central tendency and uncertainty of a numerical variable for different levels of one or more categorical variables. It is essentially a line plot with markers to represent the data at distinct categories. The central tendency is typically represented by the point at the mean or median of the data, and error bars can be used to indicate the confidence interval or standard deviation.
  ```
  # point plot
  sns.pointplot(data=tips, x='sex', y='total_bill',hue='smoker',ci=None)
  ```
- Countplot: A countplot is a categorical plot that displays the count of observations in each category of a categorical variable. It is essentially a bar plot where the data is represented in a way that shows the number of occurrences of each category. This type of plot is particularly useful for

visualizing the distribution of categorical data and understanding the relative frequencies of different categories within a dataset
# countplot
sns.countplot(data=tips,x='sex',hue='day')

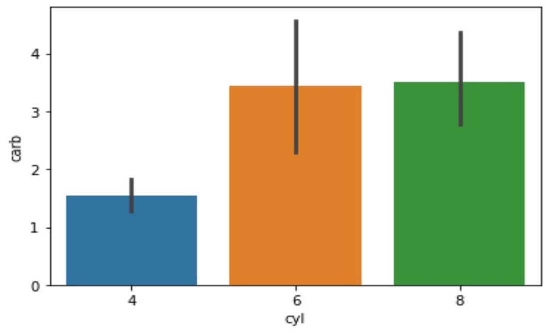
- Regression Plot: A regression plot in Seaborn is a visualization that helps to understand the relationship between two continuous variables. It displays a scatter plot of the data points, where each point represents an observation with values for both variables
sns.regplot(data=tips,x='total_bill',y='tip')
- Matrix Plot
- Multiplots

Python Seaborn Plotting Functions

The Seaborn library provides a range of plotting functions that makes the visualization and analysis of data easier.

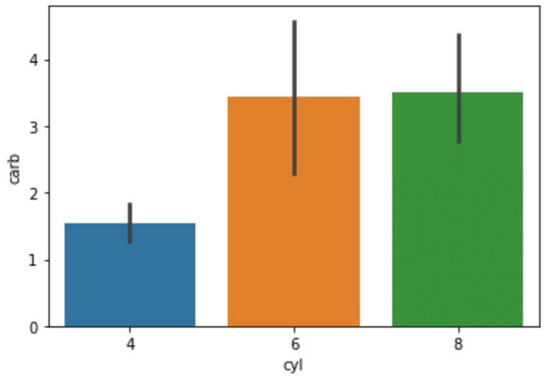
Barplot: A bar plot gives an estimate of the central tendency for a numeric variable with the height of each rectangle. It provides some indication of the uncertainty around that estimate using error bars. To build this plot, you usually choose a categorical column on the x-axis and a numerical column on the y-axis.

```
res = sns.barplot(mtcars['cyl'],mtcars['carb'])
plt.show()
```



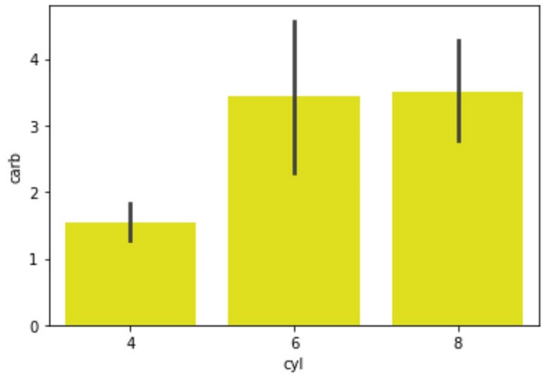
In the above plot, you have used the barplot() function and passed it in the cylinder (cyl) column in the x-axis and carburetors (carb) in the y-axis.
The code depicted below is another way to create the same bar plot.
Here you are exclusively defining the x and y-axis columns and also passing the name of the data frame using the data argument.

```
res = sns.barplot(x='cyl', y ='carb', data = mtcars)
plt.show()
```



Python Seaborn allows the users to assign colors to the bars. The bar chart below will convert all the bars to yellow color.

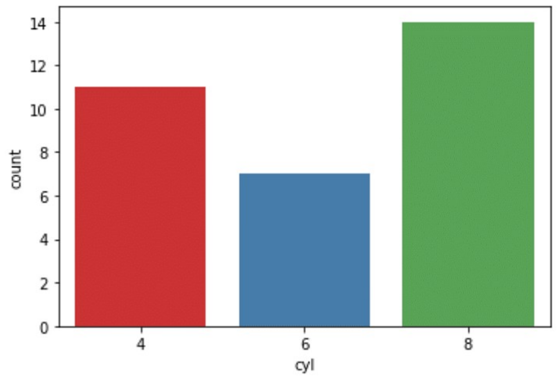
```
res = sns.barplot(mtcars['cyl'],mtcars['carb'], color = 'yellow')
plt.show()
```



Countplot: The countplot() function in the Python Seaborn library returns the count of total values for each category using bars.The below count plot returns the number of vehicles for each category of cylinders.

```
sns.countplot(x='cyl', data=mtcars, palette="Set1")
```

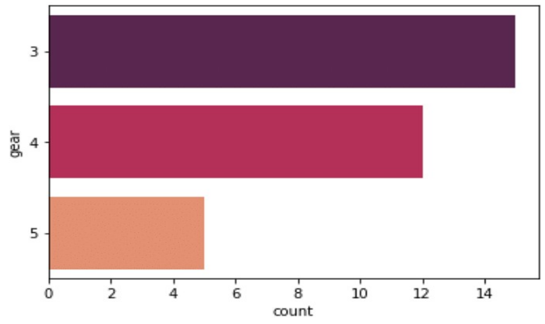
```
<matplotlib.axes._subplots.AxesSubplot at 0x181625ee188>
```



Python Seaborn allows you to create horizontal count plots where the feature column is in the y-axis and the count is on the x-axis.The below visualization shows the count of cars for each category of gear.

```
sns.countplot(y='gear', data = mtcars, palette='rocket')
```

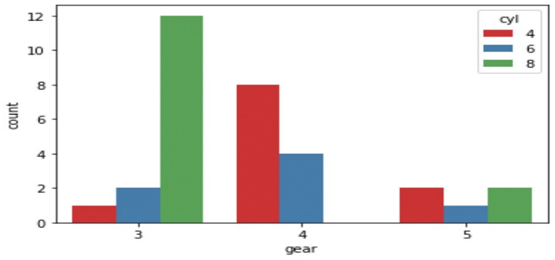
```
<matplotlib.axes._subplots.AxesSubplot at 0x1815b602f48>
```



From the above plot, you can see that we have 15 vehicles with 3 gears, 12 vehicles with 4 gears, and 5 vehicles with 5 gears.Now, you can also create a grouped count plot using the hue parameter. The hue parameter accepts the column name for color encoding.In the below count plot, you have the count of cars for each category of gears that are grouped based on the number of cylinders.

```
sns.countplot(x='gear', hue = 'cyl', data = mtcars, palette = 'Set1')
```

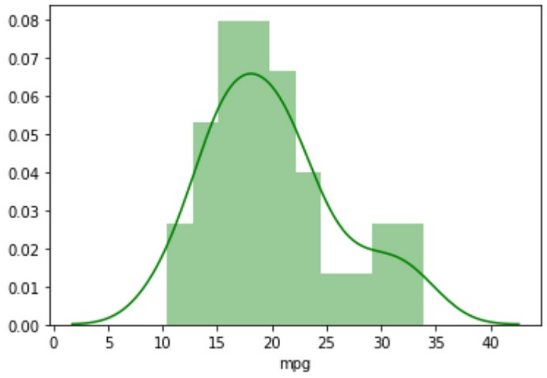
```
<matplotlib.axes._subplots.AxesSubplot at 0x1815b4cb748>
```



Distribution Plot: The Seaborn library supports the distplot() function that creates the distribution of any continuous data.In the below example, you must plot the distribution of miles per gallon of the different vehicles. The mpg metrics measure the total distance the car can travel per gallon of fuel.

```
sns.distplot(mtcars.mpg, bins = 10, color = 'g')
```

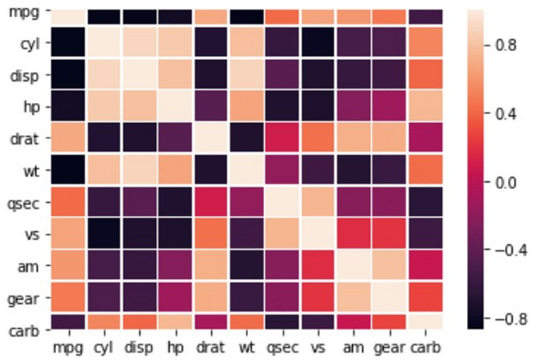
```
<matplotlib.axes._subplots.AxesSubplot at 0x1958252d908>
```



Heatmap: Heatmaps in the Seaborn library lets you visualize matrix-like data. The values of the variables are contained in a matrix and are represented as colors.Below is an example of the heatmap where you are finding the correlation between each variable in the mtcars dataset.

```
sns.heatmap(mtcars.corr(), cbar=True, linewidths = 0.5)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x19589d2ea88>
```



Scatterplot:The Seaborn scatterplot() function helps you create plots that can draw relationships between two continuous variables.Moving ahead, to understand scatter plots and other plotting functions, you must use the IRIS flower dataset.So, go ahead and load the iris dataset.

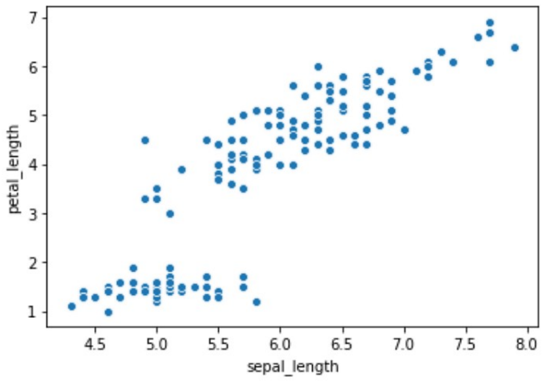
```
iris = sns.load_dataset('iris')
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

The scatter plot below shows the relationship between sepal length and petal length for different species of iris flowers.

```
sns.scatterplot(x='sepal_length', y ='petal_length', data = iris)
```

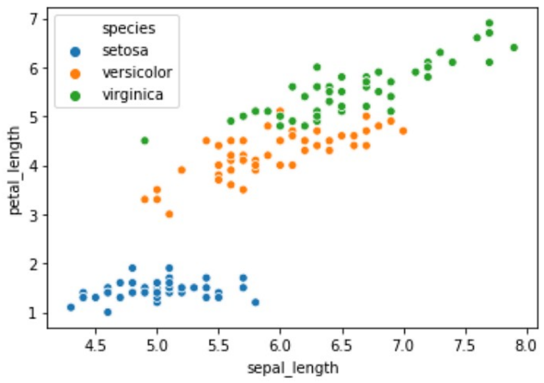
```
<matplotlib.axes._subplots.AxesSubplot at 0x195836b2188>
```



Now, you can classify the different species of flowers using the hue parameter as “species” in the function.From the below plot, you can easily differentiate the three types of iris flowers based on their sepal length and petal length.

```
sns.scatterplot(x='sepal_length', y ='petal_length', data = iris , hue = 'species')
```

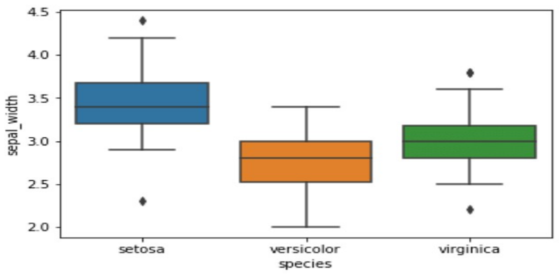
```
<matplotlib.axes._subplots.AxesSubplot at 0x19583799f08>
```



Boxplot: A boxplot, also known as a box and whisker plot, depicts the distribution of quantitative data. The box represents the quartiles of the dataset. The whiskers show the rest of the distribution, except for the outlier points.The boxplot below shows the distribution of the three species of iris flowers based on their sepal width.

```
sns.boxplot(x = 'species', y = 'sepal_width', data = iris)
```

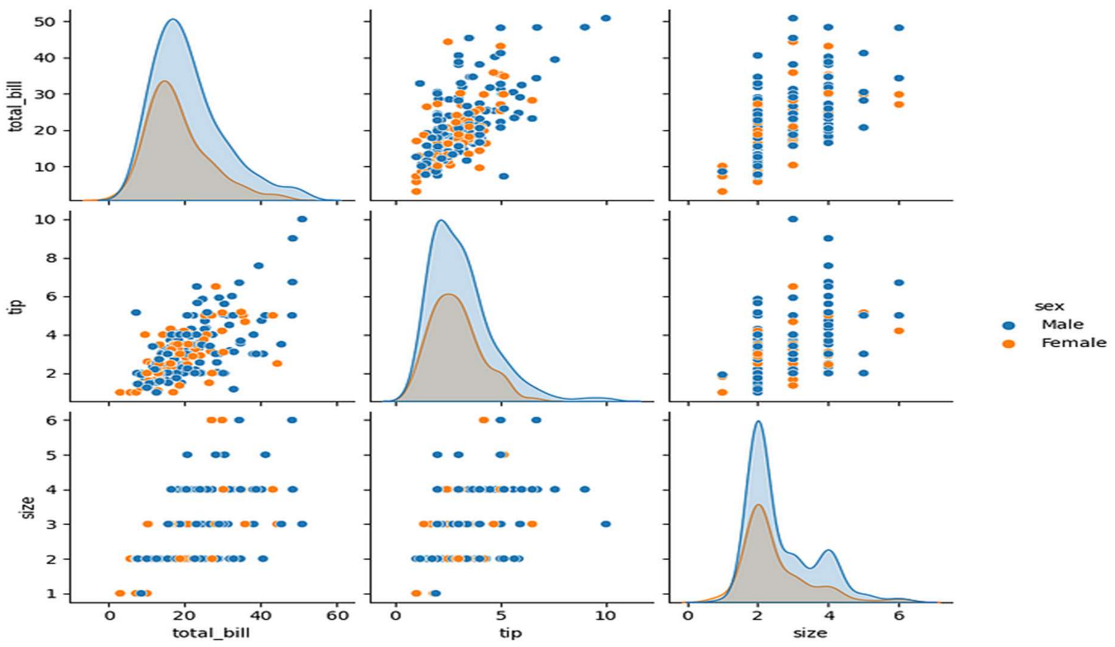
```
<matplotlib.axes._subplots.AxesSubplot at 0x1958a487108>
```



Pair plot: A pair plot, also known as a scatterplot matrix, is a matrix of graphs that enables the visualization of the relationship between each pair of variables in a dataset. It combines both histogram and scatter plots, providing a unique overview of the dataset's distributions and correlations. The primary purpose of a pair plot is to simplify the initial stages of data analysis by offering a comprehensive snapshot of potential relationships within the data.

This function (sns.pairplot()) is used to create a grid of scatter plots where each variable in the dataset is compared to every other variable. It's particularly useful for visualizing relationships in multivariate datasets.

sns.pairplot(tips,hue='sex').



Changing Plot Kinds:You can also set other plots instead of the scatter plots and set other diagonal plots. The parameters kind ('scatter' is its default value) and diag_kind ('auto' is its default value, so its kind is based on the presence of the hue parameter) respectively are used for this purpose.

```
import seaborn as sns
import matplotlib.pyplot as plt
# Loading the dataset with data about three different iris species
iris_df = sns.load_dataset('iris')
# Setting the kind parameter and diag_kind parameters
sns.pairplot(iris_df, hue='species', kind='reg', diag_kind=None, height=2, aspect=0.8)
plt.show()
```

Plotting multiple plots in seaborn: When exploring multi-dimensional data, a useful approach is to draw multiple instances of the same plot on different subsets of your dataset. This technique is sometimes called either “lattice” or “trellis” plotting, and it is related to the idea of “small multiples”. It allows a viewer to quickly extract a large amount of information about a complex dataset.

The FacetGrid class is useful when you want to visualize the distribution of a variable or the relationship between multiple variables separately within subsets of your dataset. A FacetGrid can be drawn with up to three dimensions: row, col, and hue. The first two have obvious correspondence with the resulting array of axes; think of the hue variable as a third dimension along a depth axis, where different levels are plotted with different colors.Each of relplot(), displot(),

catplot(), and lmplot() use this object internally, and they return the object when they are finished so that it can be used for further tweaking.
The class is used by initializing a FacetGrid object with a dataframe and the names of the variables that will form the row, column, or hue dimensions of the grid. These variables should be categorical or discrete, and then the data at each level of the variable will be used for a facet along that axis. For example, say we wanted to examine differences between lunch and dinner in the tips dataset:
Initializing the grid like this sets up the matplotlib figure and axes, but doesn't draw anything on them.
The main approach for visualizing data on this grid is with the FacetGrid.map() method. Provide it with a plotting function and the name(s) of variable(s) in the dataframe to plot. Let's look at the distribution of tips in each of these subsets, using a histogram:

```
tips = sns.load_dataset("tips")
g = sns.FacetGrid(tips, col="time")
g.map(sns.histplot, "tip")
```

This function will draw the figure and annotate the axes, hopefully producing a finished plot in one step. To make a relational plot, just pass multiple variable names. You can also provide keyword arguments, which will be passed to the plotting function:

```
g = sns.FacetGrid(tips, col="sex", hue="smoker")
g.map(sns.scatterplot, "total_bill", "tip", alpha=.7)
g.add_legend()
```

There are several options for controlling the look of the grid that can be passed to the class constructor.

```
g = sns.FacetGrid(tips, row="smoker", col="time", margin_titles=True)
g.map(sns.regplot, "size", "total_bill", color=".3", fit_reg=False, x_jitter=.1)
```

Note that margin_titles isn't formally supported by the matplotlib API, and may not work well in all cases. In particular, it currently can't be used with a legend that lies outside of the plot.
The size of the figure is set by providing the height of each facet, along with the aspect ratio:

```
g = sns.FacetGrid(tips, col="day", height=4, aspect=.5)
g.map(sns.barplot, "sex", "total_bill", order=["Male", "Female"])
```

Styling your plot in seaborn:
Seaborn has five built-in themes to style its plots: darkgrid, whitegrid, dark, white, and ticks. Seaborn defaults to using the darkgrid theme for its plots, but you can change this styling to better suit your presentation needs.
To use any of the preset themes pass the name of it to sns.set_style().

```
sns.set_style("darkgrid")
sns.stripplot(x="day", y="total_bill", data=tips)
```

When thinking about the look of your visualization, one thing to consider is the background color of your plot. The higher the contrast between the color palette of your plot and your figure background, the more legible your data visualization will be.

```
sns.set_style("dark")
sns.stripplot(x="day", y="total_bill", data=tips)
```

The white and tick themes will allow the colors of your dataset to show more visibly and provides higher contrast so your plots are more legible:

```
sns.set_style("ticks")
sns.stripplot(x="day", y="total_bill", data=tips)
```

A grid allows the audience to read your chart and get specific information about certain values. Research papers and reports are a good example of when you would want to include a grid.

```
sns.set_style("whitegrid")
sns.stripplot(x="day", y="total_bill", data=tips)
```

You can also specify how many spines you want to include by calling despine() and passing in the spines you want to get rid of, such as: left, bottom, top, right.

```
sns.set_style("whitegrid")
sns.stripplot(x="day", y="total_bill", data=tips)
sns.despine(left=True, bottom=True)
```

You can set the visual format, or context, using sns.set_context()

Within the usage of sns.set_context(), there are three levels of complexity:

- Pass in one parameter that adjusts the scale of the plot
- Pass in two parameters - one for the scale and the other for the font size
- Pass in three parameters - including the previous two, as well as the rc with the style parameter that you want to override

Scaling Plots: Seaborn has four presets which set the size of the plot and allow you to customize your figure depending on how it will be presented.

In order of relative size they are: paper, notebook, talk, and poster. The notebook style is the default.

```
sns.set_style("ticks")
# Smallest context:
sns.set_context("paper")
sns.stripplot(x="day", y="total_bill", data=tips)
```

3d plot of surface: 3D data visualization is the art of representing these multidimensional datasets in a way that's visually intuitive and insightful. Here's why it's such a big deal:

- Deeper Insights: 3D visualizations can reveal hidden patterns, trends, and correlations that 2D plots might miss.

- Enhanced Communication: They allow you to present complex data in a format that’s easy for others to grasp, making your insights more accessible.
- Real-world Applications: From scientific research to finance, 3D visualization has practical applications across various domains.

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
# Generate 3D data
x = np.random.rand(100)
y = np.random.rand(100)
z = x**2 + y**2
# Create a 3D scatter plot with Seaborn
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, c=z, cmap='viridis', marker='o')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
plt.title('3D Scatter Plot with Seaborn')
plt.show()
```

Just like with 2D plots, you can customize 3D plots to match your style and convey your message effectively. Here’s an example where we add colors, labels, and a legend:

```
# Adding customization
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(x, y, z, c=z, cmap='viridis', marker='o')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
# Adding a colorbar
cbar = plt.colorbar(scatter)
cbar.set_label('Z-values')
plt.title('Customized 3D Scatter Plot with Seaborn')
plt.show()
```

While scatter plots are fantastic, Seaborn can take you even further with 3D surface plots. Here’s an example of visualizing a 3D surface using Seaborn:

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

```
# Generate data for a 3D surface plot
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))
# Create a 3D surface plot with Seaborn
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
plt.title('3D Surface Plot with Seaborn')
plt.show()
```

Advanced Visualizations with Seaborn

Seaborn simplifies the creation of complex visualizations and often requires less code than Matplotlib. Here are a few advanced visualizations you can create with Seaborn:

1. Pair Plot with Regression Lines

Pair plots are useful for visualizing relationships between pairs of variables.

```
import seaborn as sns
import matplotlib.pyplot as plt
# Load the example dataset for Anscombe's quartet
df = sns.load_dataset('iris')
sns.pairplot(df, kind='reg', diag_kind='kde', hue='species', markers=['o', 's', 'D'])
plt.suptitle('Pair Plot with Regression Lines', y=1.02)
plt.show()
```

2. Violin Plot with Multiple Categories

Violin plots are great for visualizing the distribution of data across different categories.

```
import seaborn as sns
import matplotlib.pyplot as plt
# Load the example dataset for Anscombe's quartet
df = sns.load_dataset('tips')
plt.figure(figsize=(10, 6))
sns.violinplot(x='day', y='total_bill', hue='sex', data=df, split=True,
inner='quart', palette='muted')
plt.title('Violin Plot with Multiple Categories')
plt.show()
```