Matplot Library

Matplotlib: Visualization with Python

- Matplotlib is a comprehensive library for creating static, animated, and List iteminteractive visualizations in Python. Matplotlib makes easy things easy and hard things possible.
- We can create bar-plots, scatter-plots, histograms and a lot more with matplotlib.

Lineplot

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

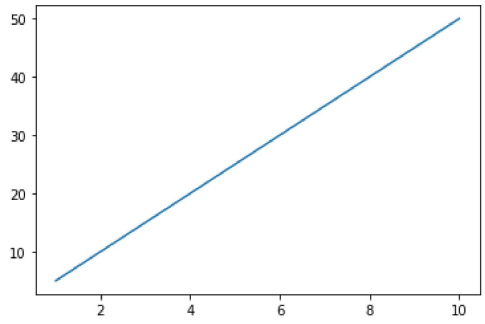
```
x = np.arange(1,11)
x
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
y = 5*x
y
```

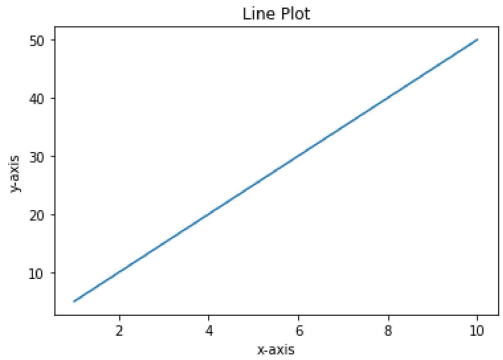
```
array([ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50])
```

```
plt.plot(x,y)
plt.show()
```



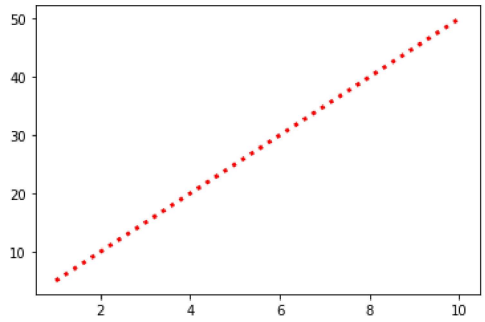
Adding Titles and Labels

```
plt.plot(x,y)
plt.title('Line Plot')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.show()
```



Changing Line Aesthetics

```
plt.plot(x,y,color='r',linestyle=':',linewidth=3)
plt.show()
```



Linestyle Description

'-' or 'solid' solid line '--' or 'dashed' dashed line '-.' or 'dashdot' dash-dotted line ':' or 'dotted' dotted line 'None' draw nothing ' ' draw nothing '' draw nothing

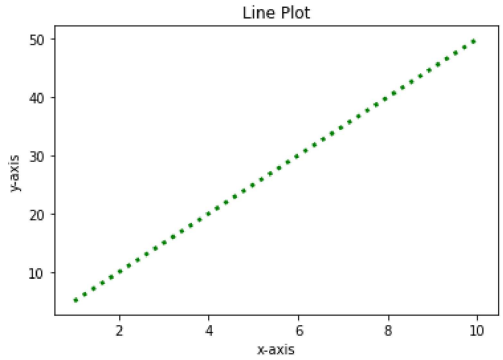
Types of Linestyle

Linestyle and its description :

linestyle	description
'-' or 'solid'	solid line
'--' or 'dashed'	dashed line
'-.' or 'dashdot'	dash-dotted line
':' or 'dotted'	dotted line
'None'	draw nothing
' '	draw nothing
''	draw nothing

Titles

```
plt.plot(x,y,color='g',linestyle='dotted',linewidth=3)
plt.title('Line Plot')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.show()
```




```
x
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

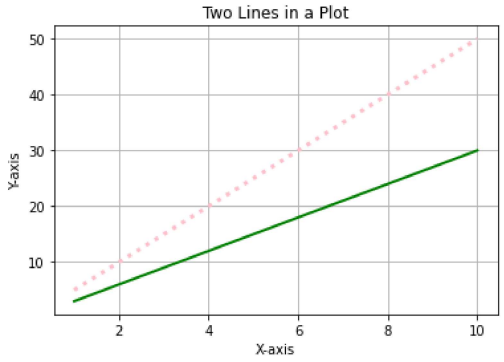
```
y1 = 3*x
y1
```

```
array([ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30])
```

```
y2 = 5*x
y2
```

```
array([ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50])
```

```
# Line Asthetics
plt.plot(x,y1,color='g', linewidth=2)
plt.plot(x,y2,color='pink',linestyle=':',linewidth=3)
plt.grid(True)
# Titles
plt.title('Two Lines in a Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()
```



Adding Sub-Plots

.subplot(row,column,index)

```
x = np.arange(1,11)
y1 = 2*x
y2 = 3*x
```

Plotting in same column

```
[ ] ↳ 1 cell hidden
```

Plotting in same row

```
[ ] ↳ 1 cell hidden
```

Bar Plot

The bar() function takes arguments that describes the layout of the bars. The categories and their values represented by the first and second argument as arrays (dictionaries).

- .bar(first,second) arguments as arrays.
- .bar(keys,values) as dictionaries.

Example

```
student = {'Varun':90,'Nivedita':75,'Prince':45,'Anuj':60,'Aparna':100}
names = list(student.keys())
names
```

```
['Varun', 'Nivedita', 'Prince', 'Anuj', 'Aparna']
```

```
marks = list(student.values())
marks
```

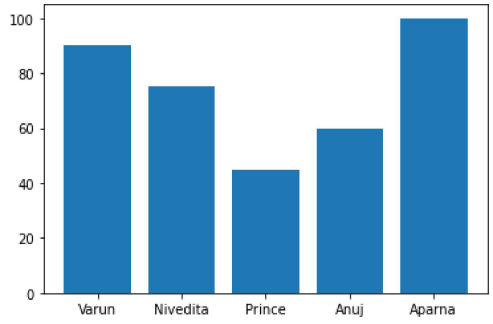
```
[90, 75, 45, 60, 100]
```

Creating Bar Graph

With Pyplot, you can use the bar() function to draw bar graphs:

As default, graph will be plotted vertically.

```
plt.bar(names,marks)
plt.show()
```



.bar(marks,names)

[ ] ↳ *1 cell hidden*

Set Titles to Bar-Plot

[ ] ↳ *1 cell hidden*

Set color of Bar Plot

[ ] ↳ *1 cell hidden*

Set width to Bar Plot

[ ] ↳ *3 cells hidden*

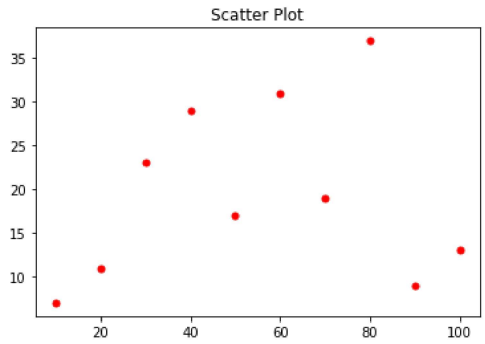
Scatter Plot

A scatter plot is a diagram where each value in the data set is represented by a dot.

Use the scatter() method to draw a scatter plot diagram:

Example

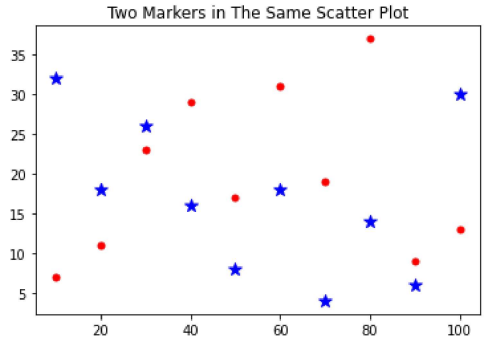
```
x = [10,20,30,40,50,60,70,80,90,100]
a = [7,11,23,29,17,31,19,37,9,13,]
plt.scatter(x,a,marker='.',c='r',s=100)
''' where c => color
          s => size
'''
# Titles
plt.title('Scatter Plot')
plt.show()
```



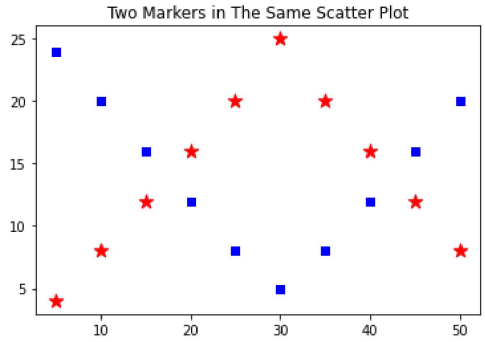
Adding two markers in same plot

```
x = [10,20,30,40,50,60,70,80,90,100]
a = [7,11,23,29,17,31,19,37,9,13,]
```

```
b = [32,18,26,16,8,18,4,14,6,30]
plt.scatter(x,a,marker='.',c='r',s=100)
plt.scatter(x,b,marker='*',c='b',s=100)
plt.title('Two Markers in The Same Scatter Plot')
plt.show()
```




```
x = [5,10,15,20,25,30,35,40,45,50]
a = [4,8,12,16,20,25,20,16,12,8]
b = [24,20,16,12,8,5,8,12,16,20]
plt.scatter(x,a,marker='*',c='r',s=120)
plt.scatter(x,b,marker=',',c='b',s=40)
# Title
plt.title('Two Markers in The Same Scatter Plot')
plt.show()
```



Adding Subplot

```
[ ]  ↳ 1 cell hidden
```

HISTOGRAM

A histogram is a graph showing frequency distributions.

It is a graph showing the number of observations within each given interval.

The hist() function will read the array and produce a histogram.

Difference between Histogram and Barchart

Histogram:

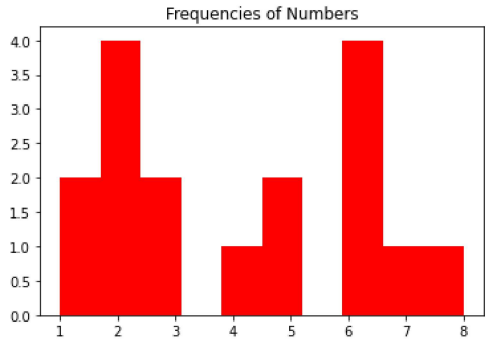
- Histogram is a type of bar chart that is used to represent statistical information by way of bars to display the frequency distribution of continuous data. It indicates the number of observations that lie in-between the range of values, which is known as class or bin.
- A histogram chart helps you to display the distribution of numerical data by rendering vertical bars. You can compare non-discrete values with the help of a histogram chart.

Bar Chart:

- Bar Chart is used to compare the frequency, total count, sum, or an average of data in different categories by using horizontal or vertical bars. It is also known as a column chart.
- With the help of Bar Chart, you can also do various types of category comparison, which is graphically visualized using a bar chart. Generally, the bar chart will have an axis, label, scales, and bars, represent measurable values like percentages or numbers.
- With the help of Bar Chart, you can also do various types of category comparison, which is graphically visualized using a bar chart. Generally, the bar chart will have an axis, label, scales, and bars, represent measurable values like percentages or numbers.

Creating Histogram

```
data = [2,2,2,1,5,5,6,3,2,1,6,6,4,3,6,7,8]
plt.hist(data,color='r')
plt.title('Frequencies of Numbers')
plt.show()
```



Let's understand about size of bins in a Histogram

[ ] ↳ *4 cells hidden*

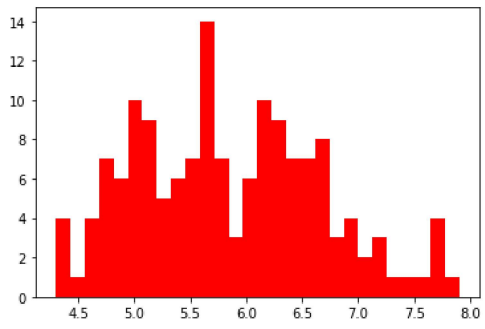
Histogram

Working with a dataset

```
import pandas as pd
iris = pd.read_csv('/content/drive/MyDrive/Notes/Iris.csv')
iris.head()
```

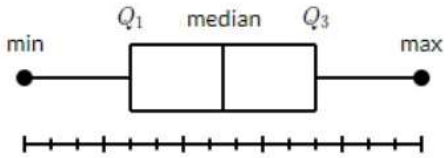
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

```
plt.hist(iris['SepalLengthCm'],color='r',bins=28)
plt.show()
```



Box Plot

A box plot which is also known as a whisker plot displays a summary of a set of data containing the minimum, first quartile, median, third quartile, and maximum. In a box plot, we draw a box from the first quartile to the third quartile. A vertical line goes through the box at the median. The whiskers go from each quartile to the minimum or maximum.



Let us create the data for the boxplots. We use the numpy.random.normal() function to create the fake data. It takes three arguments, mean and standard deviation of the normal distribution, and the number of values desired.

np.random.seed(10) data = np.random.normal(100, 10, 200)

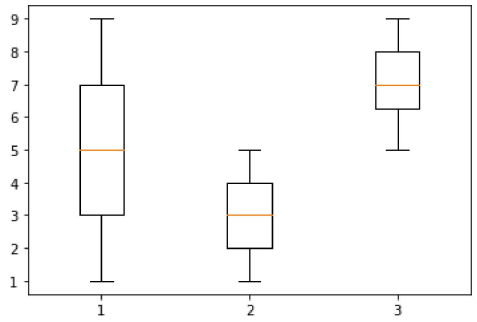
Example:

```
one = [1,2,3,4,5,6,7,8,9]
two = [1,2,3,4,5,4,3,2,1]
three = [6,7,8,9,8,7,6,5,7,8]
# Make list of lists
data = list([one,two,three])
data
```

```
[[1, 2, 3, 4, 5, 6, 7, 8, 9],
 [1, 2, 3, 4, 5, 4, 3, 2, 1],
 [6, 7, 8, 9, 8, 7, 6, 5, 7, 8]]
```

```
# Make Box Plot
plt.boxplot(data)
plt.show()
```

```
/usr/local/lib/python3.8/dist-packages/matplotlib/cbook/__init__.py:1376: VisibleDeprecationWarning: Cr
  X = np.atleast_1d(X.T if isinstance(X, np.ndarray) else np.asarray(X))
```



Violin-Plot

Violin plots are similar to box plots, except that they also show the probability density of the data at different values. These plots include a marker for the median of the data and a box indicating the interquartile range, as in the standard box plots. Overlaid on this box plot is a kernel density estimation. Like box plots, violin plots are used to represent comparison of a variable distribution (or sample distribution) across different "categories".

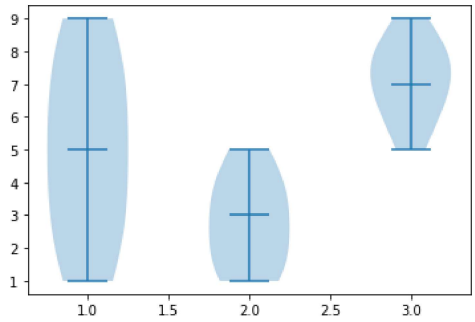
A violin plot is more informative than a plain box plot. In fact while a box plot only shows summary statistics such as mean/median and interquartile ranges, the violin plot shows the full distribution of the data.

Example:

```
one = [1,2,3,4,5,6,7,8,9]
two = [1,2,3,4,5,4,3,2,1]
three = [6,7,8,9,8,7,6,5,7,8]
# Make list of lists
data = list([one,two,three])
# Make Violin Plot
plt.violinplot(data)
plt.show()
```

```
/usr/local/lib/python3.8/dist-packages/matplotlib/cbook/__init__.py:1376: VisibleDeprecationWarning: Creating an ndarray from ragged
  X = np.atleast_1d(X.T if isinstance(X, np.ndarray) else np.asarray(X))
/usr/local/lib/python3.8/dist-packages/numpy/core/fromnumeric.py:1970: VisibleDeprecationWarning: Creating an ndarray from ragged nes
  result = asarray(a).shape
```


```
# Show Median
plt.violinplot(data,showmedians=True)
plt.show()
```



Pie-Chart

A Pie Chart can only display one series of data. Pie charts show the size of items (called wedge) in one data series, proportional to the sum of the items. The data points in a pie chart are shown as a percentage of the whole pie.

Matplotlib API has a pie() function that generates a pie diagram representing data in an array.

Parameters

Following table lists down the parameters foe a pie chart –

x	array-like. The wedge sizes.
labels	list. A sequence of strings providing the labels for each wedge.
Colors	A sequence of matplotlibcolorargs through which the pie chart will cycle. If None, will use the colors in the currently active cycle.
Autopct	string, used to label the wedges with their numeric value. The label will be placed inside the wedge. The format string will be fmt%pct.

Labels:

- Add labels to the pie chart with the label parameter.
- The label parameter must be an array with one label for each wedge:

Creating Data:

```
fruit = ['Apple','Mango','Guava','Orange','Cherry']
quantity = [10,19,5,14,27]

plt.pie(quantity, labels = fruit)
plt.show()
```

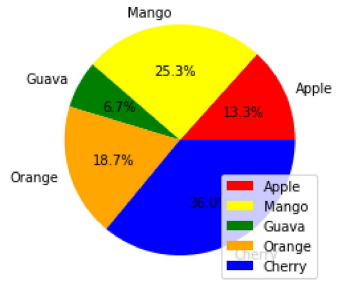
Add Percentage and Change the Color:

The proportionate percentage is displayed inside the respective wedge with the help of autopct parameter which is set to %1.2f%.

```
plt.pie(quantity, labels=fruit, autopct='%0.1f%%', colors = ['red','yellow','green','orange','blue'])

#Add a legend
plt.legend()
# We can also use .figure() function to show the chart.
#plt.show()
plt.figure()
```

```
<Figure size 432x288 with 0 Axes>
```

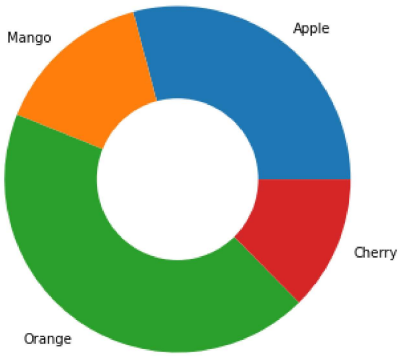



```
<Figure size 432x288 with 0 Axes>
```

DoughNut Chart (Donut-Chart)

Donut charts are the modified version of Pie Charts with the area of center cut out. A donut is more concerned about the use of area of arcs to represent the information in the most effective manner instead of Pie chart which is more focused on comparing the proportion area between the slices. Donut charts are more efficient in terms of space because the blank space inside the donut charts can be used to display some additional information about the donut chart.

```
# Creating Data
fruits = ['Apple','Mango','Orange','Cherry']
quantity = [67,34,100,29]
# Making Plot
plt.pie(quantity, labels = fruits, radius = 1.5)
plt.pie([1],radius = 0.7, colors =['w'])
plt.show()
```



Seaborn Library

Seaborn: statistical data visualization

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

Seaborn is a library that uses Matplotlib underneath to plot graphs. It will be used to visualize random distributions.

Install Seaborn

If you have Python and PIP already installed on a system, install it using this command:

pip install seaborn

If you use Jupyter, install Seaborn using this command:

!pip install seaborn

```
pip install seaborn
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: seaborn in /usr/local/lib/python3.8/dist-packages (0.11.2)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.8/dist-packages (from seaborn) (1.21.6)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.8/dist-packages (from seaborn) (1.7.3)
Requirement already satisfied: matplotlib>=2.2 in /usr/local/lib/python3.8/dist-packages (from seaborn) (3.2.2)
Requirement already satisfied: pandas>=0.23 in /usr/local/lib/python3.8/dist-packages (from seaborn) (1.3.5)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.8/dist-packages (from matplotlib>=2.2->seaborn) (0.11.0)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib>=2.2->seaborn) (2.8.2)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib>=2.2->seaborn) (1.4.4)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.8/dist-packages (from matplotlib>=2
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (from pandas>=0.23->seaborn) (2022.6)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from python-dateutil>=2.1->matplotlib>=2.2->seabor
```

Import Seaborn and Pyplot

```
import seaborn as sb
from matplotlib import pyplot as plt
```

Loading inbuilt dataset 'fmri' of seaborn library.

```
fmri = sb.load_dataset('fmri')
fmri.head()
```

	subject	timepoint	event	region	signal
0	s13	18	stim	parietal	-0.017552
1	s5	14	stim	parietal	-0.080883
2	s12	18	stim	parietal	-0.081033
3	s11	18	stim	parietal	-0.046134
4	s10	18	stim	parietal	-0.037970

Check Rows and Columns of dataset 'fmri'

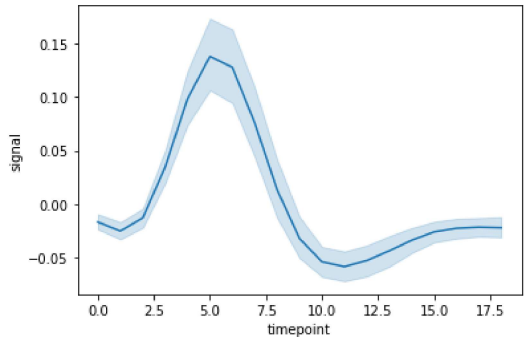
```
fmri.shape
```

```
(1064, 5)
```

Lineplot using Seaborn and Matplotlib

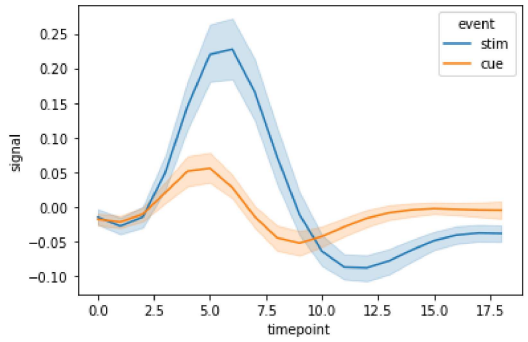
Take 'timepoint' as x-axis and 'signal' as y-axis.

```
sb.lineplot(x='timepoint', y='signal', data = fmri)
plt.show()
```



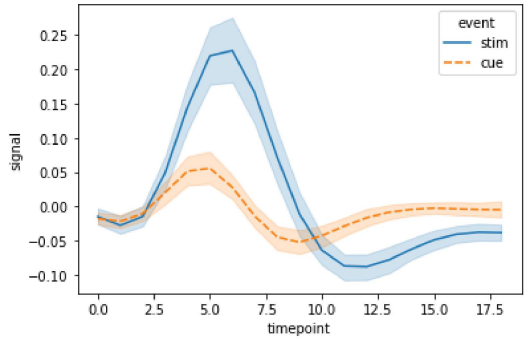
Grouping data with 'hue'.

```
sb.lineplot(x = 'timepoint', y = 'signal', hue = 'event', data = fmri)
plt.show()
```



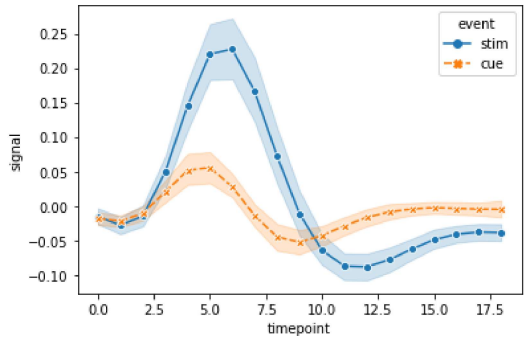
Adding Styles

```
sb.lineplot(x = 'timepoint', y = 'signal', data = fmri, hue = 'event', style = 'event')
plt.show()
```



Adding Markers

```
sb.lineplot(x = 'timepoint', y = 'signal', hue = 'event', style = 'event', markers = True, data = fmri)
plt.show()
```



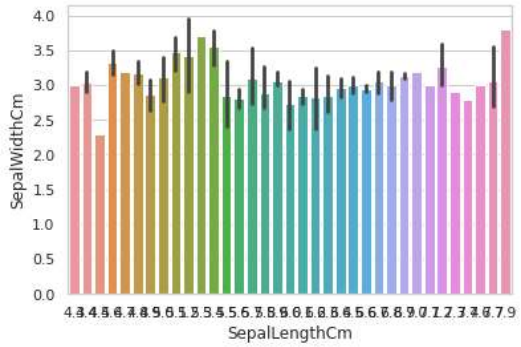
Seaborn Bar Plot

```
import pandas as pd
sb.set(style = 'whitegrid')
iris = pd.read_csv('/content/drive/MyDrive/Notes/Iris.csv')
```

```
iris.head()
```

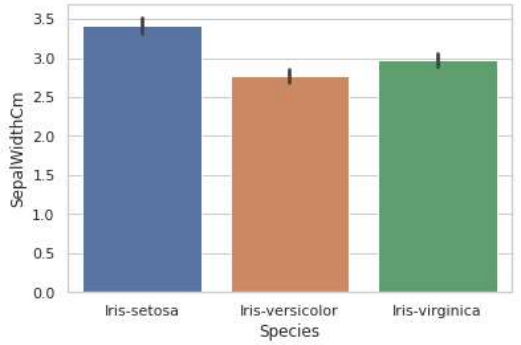
```
Id  SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm      Species
```

```
sb.barplot(x = 'SepalLengthCm', y = 'SepalWidthCm', data = iris)
plt.show()
```



To better understanding check another example

```
sb.barplot(x = 'Species', y = 'SepalWidthCm', data = iris)
plt.show()
```



To better understanding of the concept of Seaborn Library, let's try a new dataset 'Pokemon Dataset'.

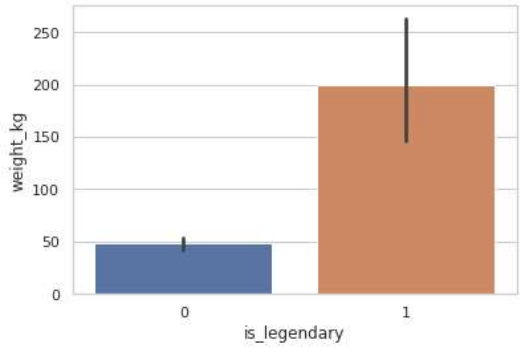
```
pokemon = pd.read_csv('/content/drive/MyDrive/Notes/pokemon.csv')
pokemon.head()
```

	abilities	against_bug	against_dark	against_dragon	against_electric	against_fairy	against_fight	against_fire	against_f
0	['Overgrow', 'Chlorophyll']	1.0	1.0	1.0	0.5	0.5	0.5	2.0	
1	['Overgrow', 'Chlorophyll']	1.0	1.0	1.0	0.5	0.5	0.5	2.0	
2	['Overgrow', 'Chlorophyll']	1.0	1.0	1.0	0.5	0.5	0.5	2.0	
3	['Blaze', 'Solar Power']	0.5	1.0	1.0	1.0	0.5	1.0	0.5	
4	['Blaze', 'Solar Power']	0.5	1.0	1.0	1.0	0.5	1.0	0.5	

5 rows × 41 columns

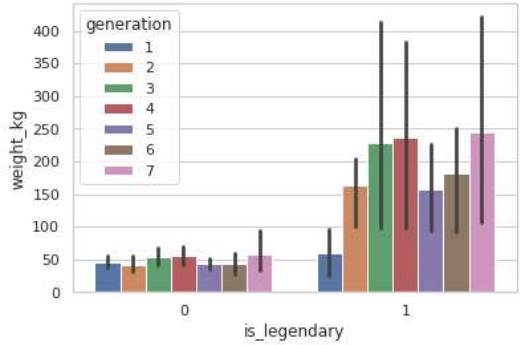
```
sb.barplot(x = 'is_legendary', y = 'speed', data = pokemon)
plt.show()
```


```
sb.barplot(x = 'is_legendary', y = 'weight_kg', data = pokemon)
plt.show()
```

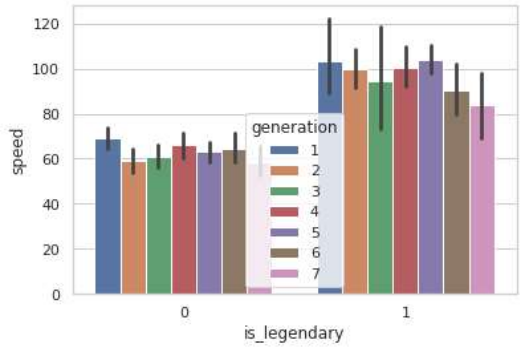


Grouping Bar-plot with 'hue'

```
sb.barplot(x = 'is_legendary', y = 'weight_kg', data = pokemon, hue = 'generation')
plt.show()
```




```
sb.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation')
plt.show()
```




```
import seaborn as sns
```

Seaborn Color Palette

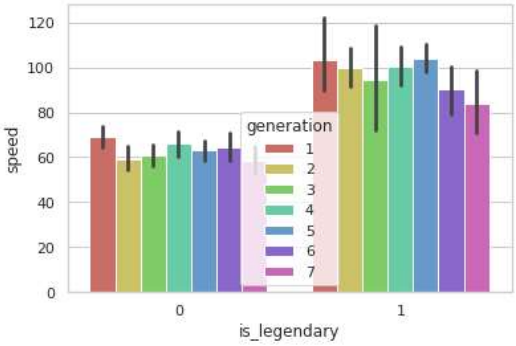
hsl

This is what most seaborn functions default to when they need to use more colors than are currently set in the default color cycle.

The most common way to do this uses the 'hls' color space, which is a simple transformation of RGB values. We saw this color palette before as a counterexample for how to plot a histogram:

```
sns.color_palette('hls')
```

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'hls')
plt.show()
```

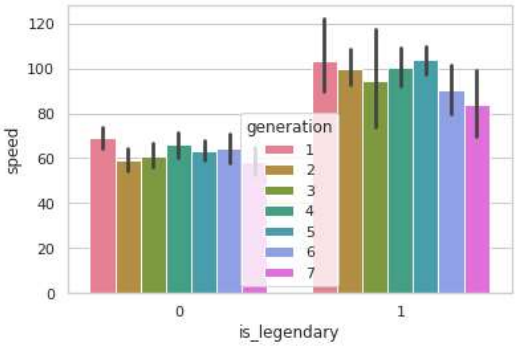


husl

Seaborn provides an interface to the husl system (since renamed to HSLuv), which achieves less intensity variation as you rotate around the color wheel:

```
sns.color_palette('husl',8)
```

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'husl')
plt.show()
```



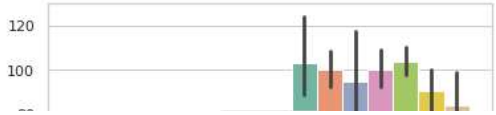
Using categorical Color Brewer palettes

Another source of visually pleasing categorical palettes comes from the Color Brewer tool (which also has sequential and diverging palettes, as we'll see below).

Set2

```
sns.color_palette('Set2',10)
```

```
# Set2
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'Set2')
plt.show()
```

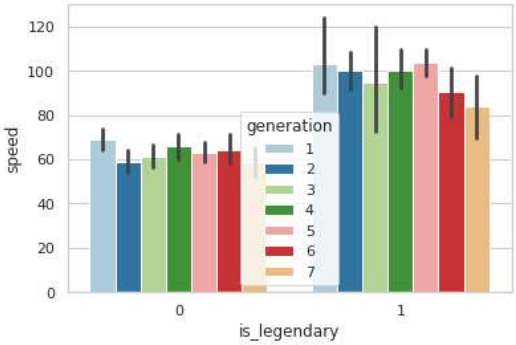


Be aware that the qualitative Color Brewer palettes have different lengths, and the default behavior of color_palette() is to give you the full list:

Paired


```
sns.color_palette('Paired')
```

```
# Paired
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'Paired')
plt.show()
```



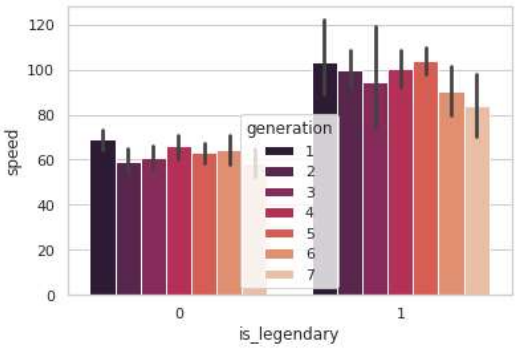
Perceptually uniform palettes

The relative discriminability of two colors is proportional to the difference between the corresponding data values. Seaborn includes four perceptually uniform sequential colormaps: "rocket", "mako", "flare", and "crest". The first two have a very wide luminance range and are well suited for applications such as heatmaps, where colors fill the space they are plotted into:

rocket

```
sns.color_palette('rocket')
```

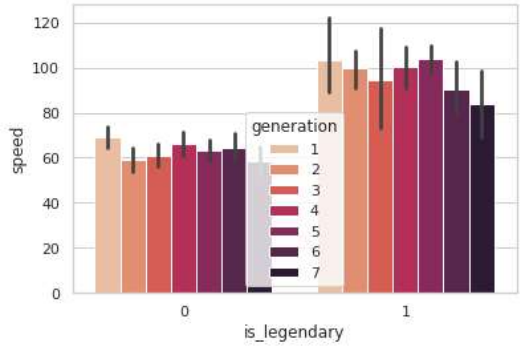
```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'rocket')
plt.show()
```



As with the convention in matplotlib, every continuous colormap has a reversed version, which has the suffix "_r":

```
sns.color_palette('rocket_r')
```

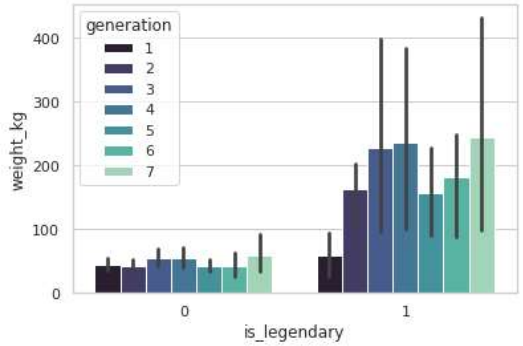
```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'rocket_r')
plt.show()
```



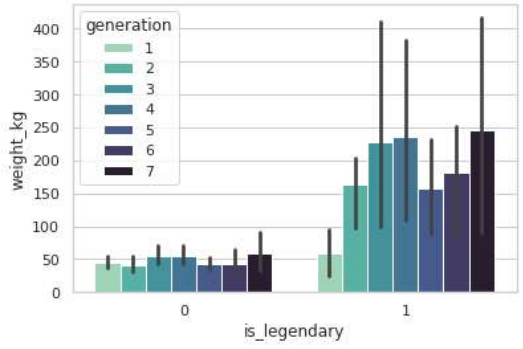
mako

```
sns.color_palette('mako')
```

```
sns.barplot(x = 'is_legendary', y = 'weight_kg', data = pokemon, hue = 'generation', palette = 'mako')
plt.show()
```



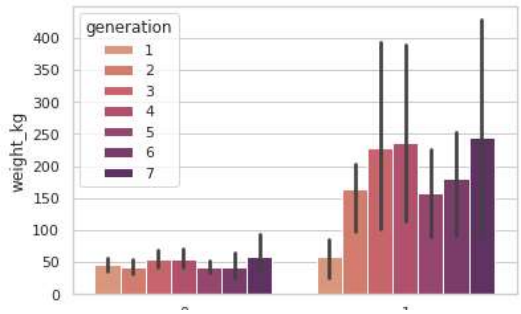

```
# Reverse
sns.barplot(x = 'is_legendary', y = 'weight_kg', data = pokemon, hue = 'generation', palette = 'mako_r')
plt.show()
```



flare

```
sns.color_palette('flare')
```

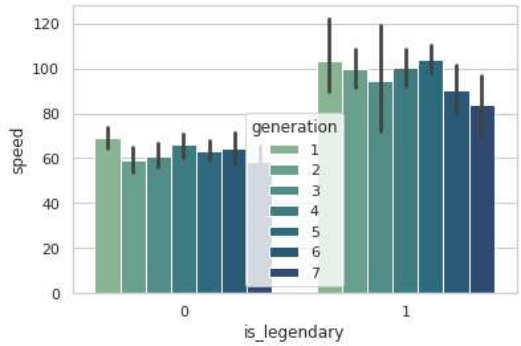
```
sns.barplot(x = 'is_legendary', y = 'weight_kg', data = pokemon, hue = 'generation', palette = 'flare')
plt.show()
```



crest

```
sns.color_palette('crest')
```

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'crest')
plt.show()
```



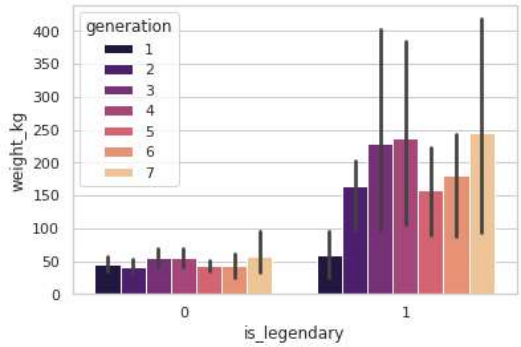
It is also possible to use the perceptually uniform colormaps provided by matplotlib, such as "magma" and "viridis":

magma

```
sns.color_palette('magma')
```

```
# Reversed Palette
sns.color_palette('magma_r')
```

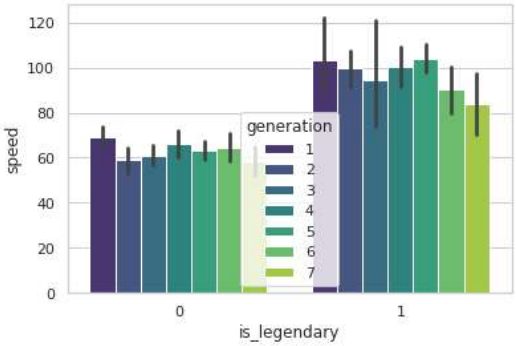
```
sns.barplot(x = 'is_legendary', y = 'weight_kg', data = pokemon, hue = 'generation',palette = 'magma')
plt.show()
```



viridis

```
sns.color_palette('viridis')
```

```
sb.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'viridis')
plt.show()
```



Sequential Color Brewer palettes

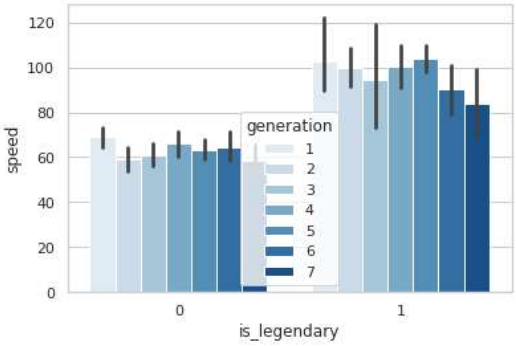
The Color Brewer library also has some good options for sequential palettes. They include palettes with one primary hue:

Blues

```
sns.color_palette('Blues')
```

```
# Reversed Palette
sns.color_palette('Blues_r')
```

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'Blues')
plt.show()
```

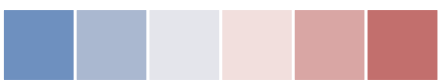


Perceptually uniform diverging palettes

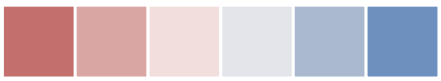
Seaborn includes two perceptually uniform diverging palettes: "vlag" and "icefire". They both use blue and red at their poles, which many intuitively processes as “cold” and “hot”:

vlag

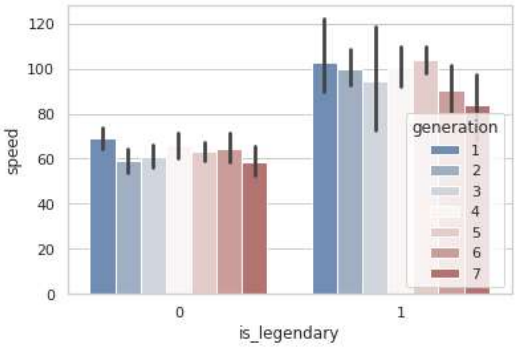
```
sns.color_palette('vlag')
```



```
# Reversed Palette
sns.color_palette('vlag_r')
```



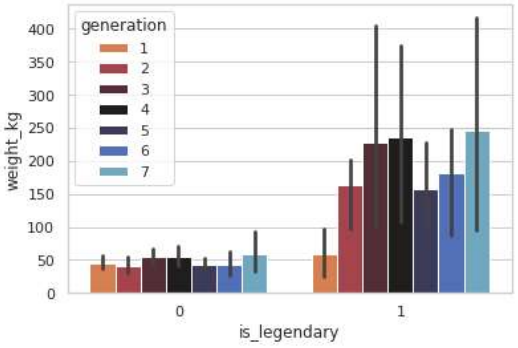
```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'vlag')
plt.show()
```



icefire

```
sns.color_palette('icefire')
```

```
sns.barplot(x = 'is_legendary', y = 'weight_kg', data = pokemon, hue = 'generation', palette = 'icefire_r')
plt.show()
```



Other diverging palettes

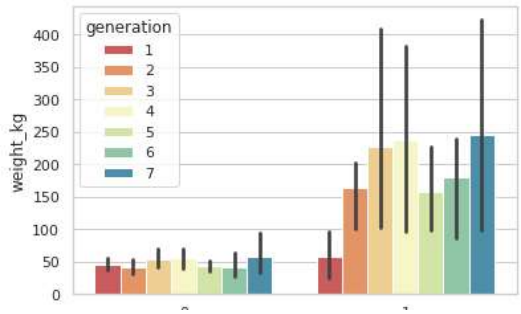
There are a few other good diverging palettes built into matplotlib, including Color Brewer palettes:

Spectral

```
sns.color_palette('Spectral')
```

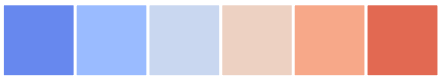
```
# Reversed Palette
sns.color_palette('Spectral_r')
```

```
sns.barplot(x = 'is_legendary', y = 'weight_kg', data = pokemon, hue = 'generation', palette = 'Spectral')
plt.show()
```

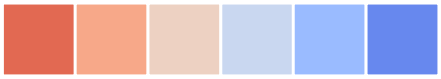


coolwarm

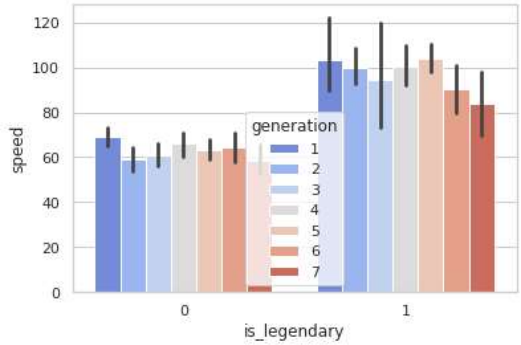
```
sns.color_palette('coolwarm')
```



```
# Reversed Palette
sns.color_palette('coolwarm_r')
```



```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', palette = 'coolwarm')
plt.show()
```



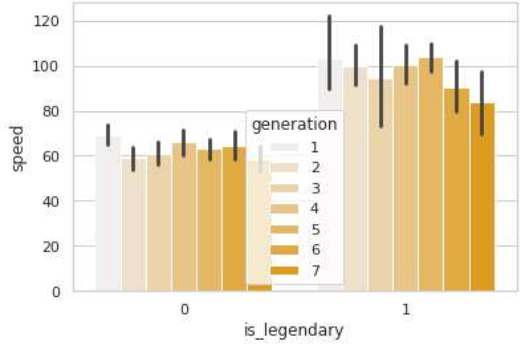
Fixed or Specific Color

For fixed or specific color of barplot use color as place of palette.

Example:

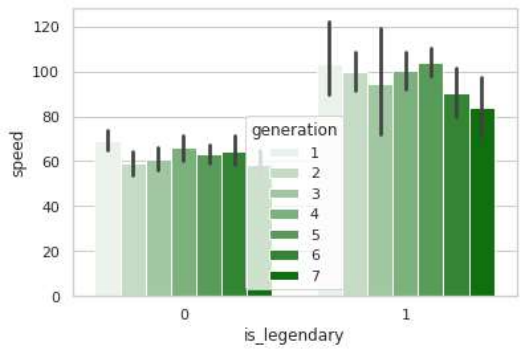
color = 'orange'

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', color = 'Orange')
plt.show()
```



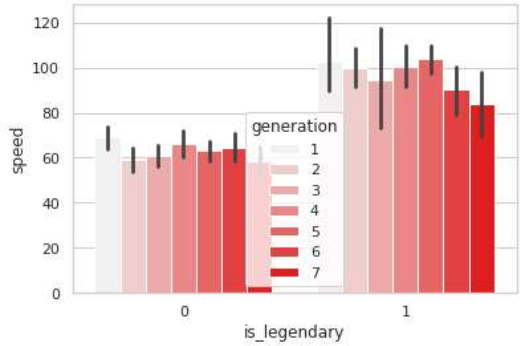
Green

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', color = 'green')
plt.show()
```



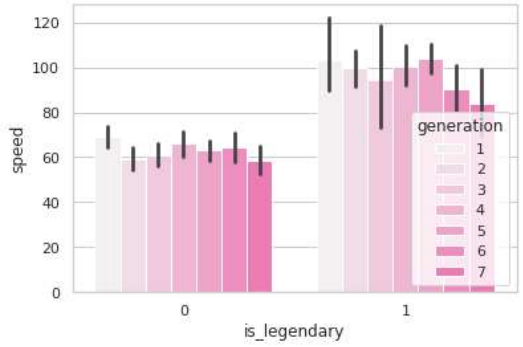
Red

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', color = 'red')
plt.show()
```



Hotpink

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', color = 'hotpink')
plt.show()
```



Black

```
sns.barplot(x = 'is_legendary', y = 'speed', data = pokemon, hue = 'generation', color = 'Black')
plt.show()
```

Loading iris dataset


```
iris = pd.read_csv('/content/drive/MyDrive/Notes/Iris.csv')
iris.head()
```

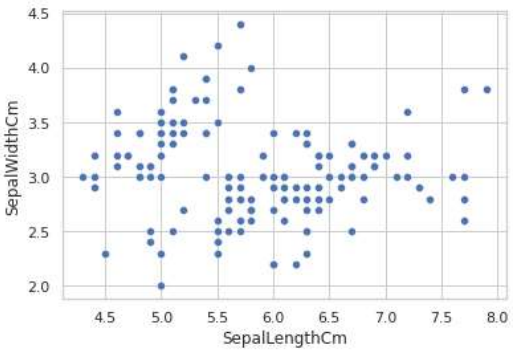
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

Seaborn Scatterplot

The relationship between x and y can be shown for different subsets of the data using the hue, size, and style parameters. These parameters control what visual semantics are used to identify the different subsets. It is possible to show up to three dimensions independently by using all three semantic types, but this style of plot can be hard to interpret and is often ineffective. Using redundant semantics (i.e. both hue and style for the same variable) can be helpful for making graphics more accessible.

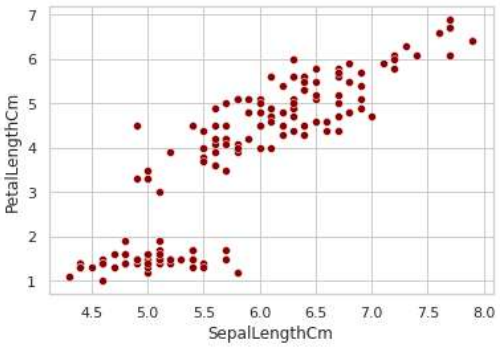
Scatterplot of iris dataset

```
sns.scatterplot(x = 'SepalLengthCm', y = 'SepalWidthCm', data = iris)
plt.show()
```



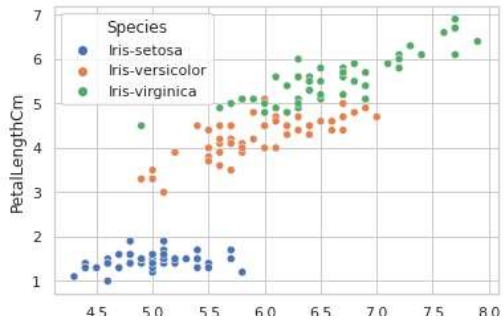
Change the color of scatterplot

```
sns.scatterplot(x = 'SepalLengthCm', y = 'PetalLengthCm', data = iris, color = 'darkred')
plt.show()
```

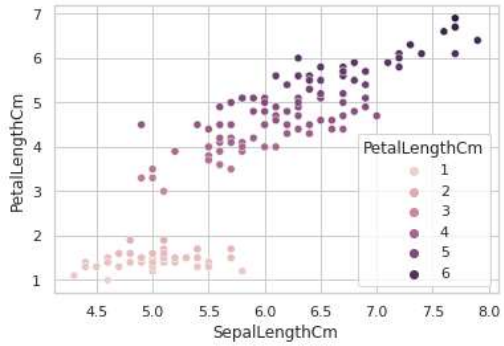


Seaborn Scatterplot with hue and style

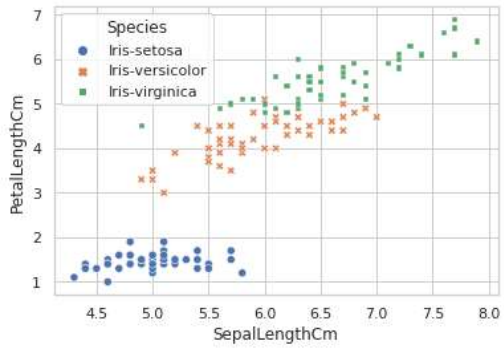
```
sns.scatterplot(x = 'SepalLengthCm', y = 'PetalLengthCm', data = iris, hue = 'Species')
plt.show()
```



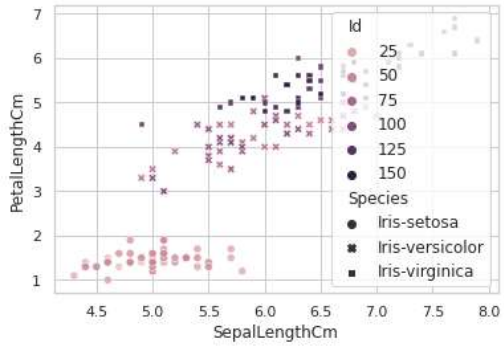

```
sns.scatterplot(x = 'SepalLengthCm', y = 'PetalLengthCm', data = iris, hue = 'PetalLengthCm')
plt.show()
```




```
sns.scatterplot(x = 'SepalLengthCm', y = 'PetalLengthCm', data = iris, hue = 'Species', style = 'Species')
plt.show()
```




```
sns.scatterplot(x = 'SepalLengthCm', y = 'PetalLengthCm', data = iris, hue = 'Id', style = 'Species')
plt.show()
```



Seaborn Histogram

displot( )

This function provides access to several approaches for visualizing the univariate or bivariate distribution of data, including subsets of data defined by semantic mapping and faceting across multiple subplots. The kind parameter selects the approach to use:

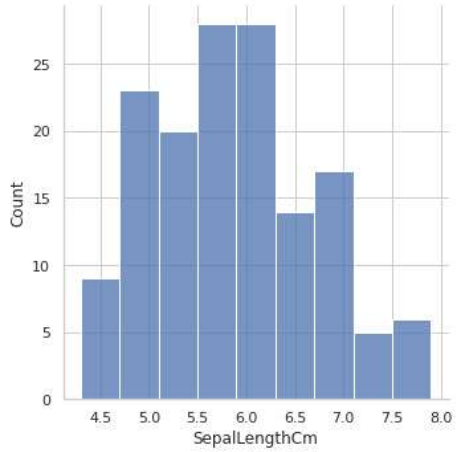
histplot( ) (with kind="hist"; the default)

kdeplot( ) (with kind="kde")

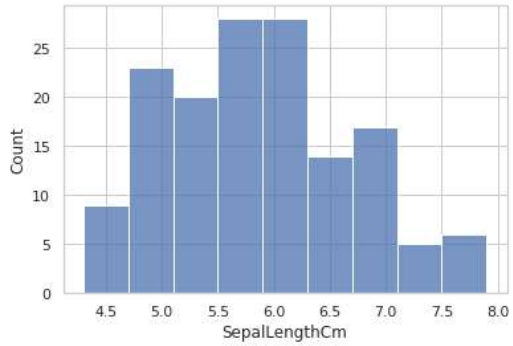
ecdfplot( ) (with kind="ecdf"; univariate-only)

Additionally, a rugplot( ) can be added to any kind of plot to show individual observations.

```
sns.displot(x = 'SepalLengthCm', data = iris)
plt.show()
```



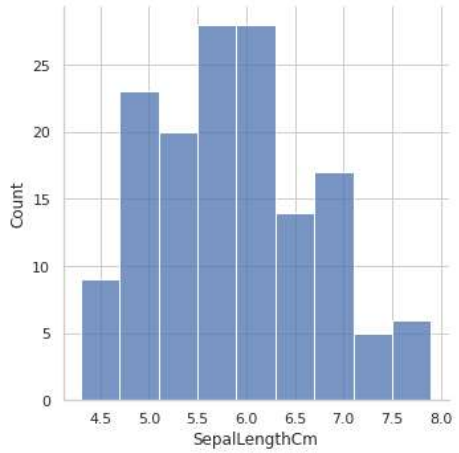

```
sns.histplot(x = 'SepalLengthCm', data = iris)
plt.show()
```



kind = 'kde' : Kernal Density Estimation Plot

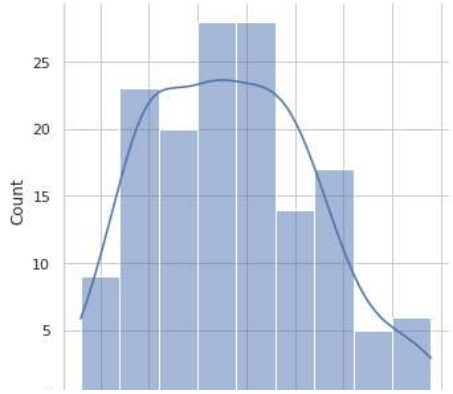
kind = 'hist'

```
sns.displot(x = 'SepalLengthCm', data = iris, kind = 'hist')
plt.show()
```



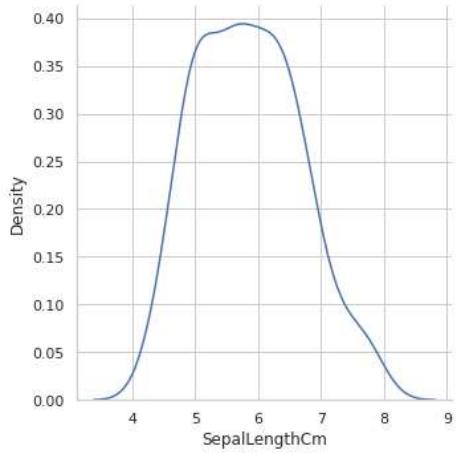
kde = True

```
sns.displot(x = 'SepalLengthCm', data = iris, kde = True)
plt.show()
```

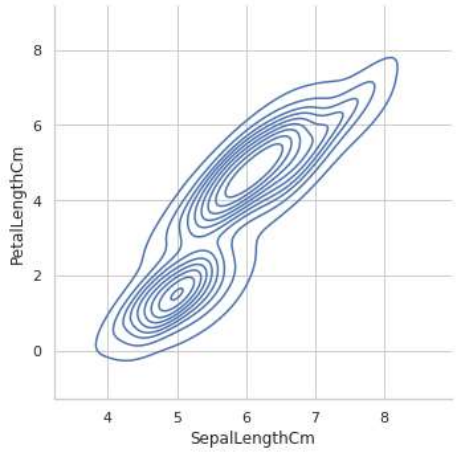


kind = 'kde'

```
sns.displot(x = 'SepalLengthCm', data = iris, kind = 'kde')
plt.show()
```

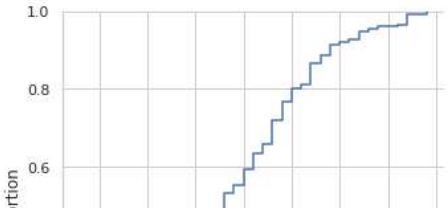



```
sns.displot(x = 'SepalLengthCm',y = 'PetalLengthCm', data = iris, kind = 'kde')
plt.show()
```



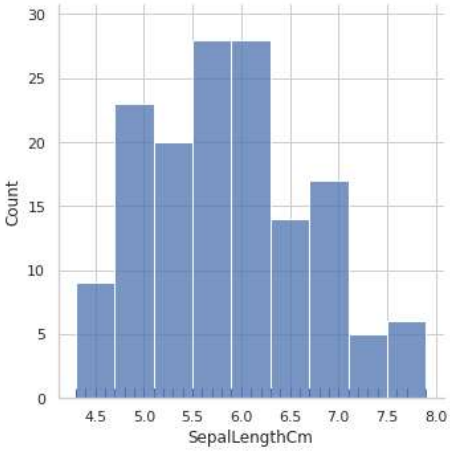
kind = 'ecdf' : Empirical Cumulative Distribution Function

```
sns.displot(x = 'SepalLengthCm',  data = iris, kind = 'ecdf')
plt.show()
```

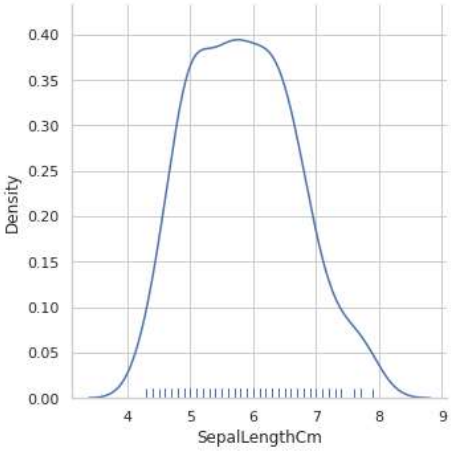


rug

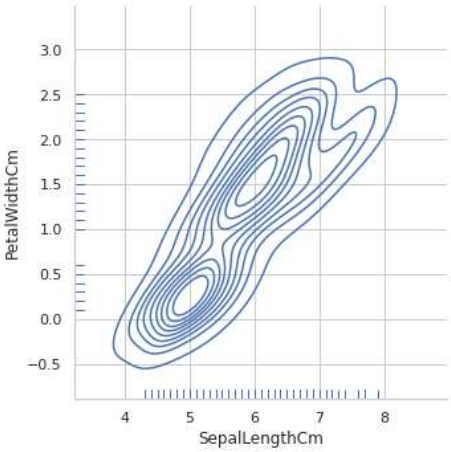

```
sns.displot(x = 'SepalLengthCm', data = iris, rug = True)
plt.show()
```




```
sns.displot(x = 'SepalLengthCm', data = iris, kind = 'kde', rug = True)
plt.show()
```

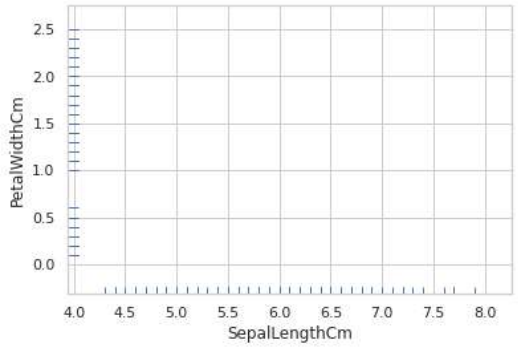



```
sns.displot(x = 'SepalLengthCm', y = 'PetalWidthCm', data = iris, kind = 'kde', rug = True)
plt.show()
```



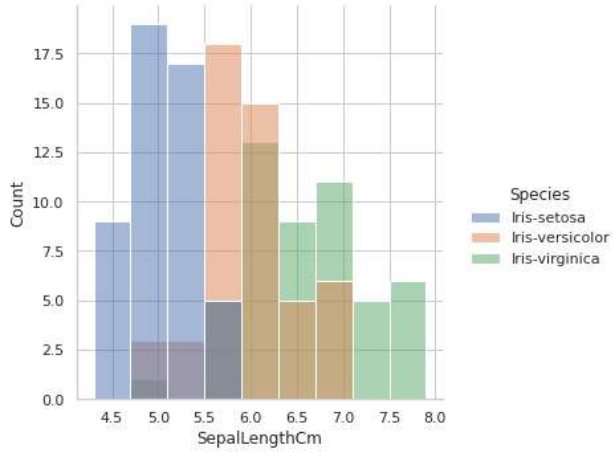
rugplot

```
sns.rugplot(x = 'SepalLengthCm', y = 'PetalWidthCm', data = iris)
plt.show()
```

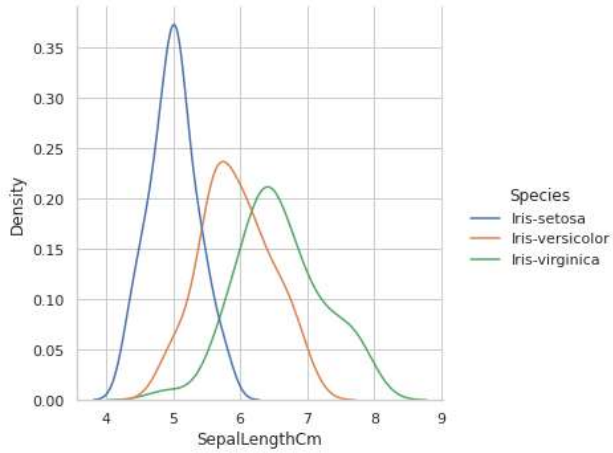


Categorical Variables : hue

```
sns.displot(x = 'SepalLengthCm', data = iris, hue = 'Species')
plt.show()
```

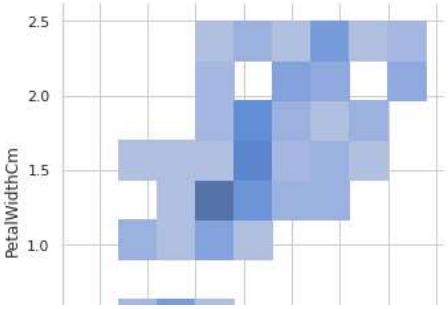



```
sns.displot(x = 'SepalLengthCm', data = iris, kind = 'kde', hue = 'Species')
plt.show()
```



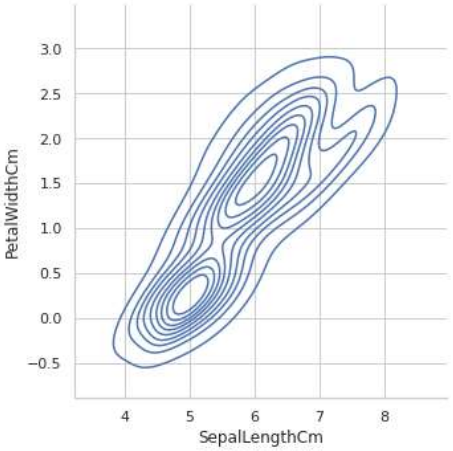
Bivariate (2D) Options

```
sns.displot(x = 'SepalLengthCm', y = 'PetalWidthCm', data = iris)
plt.show()
```



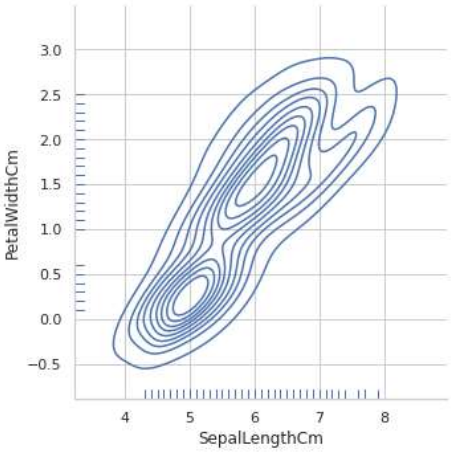
kind = 'kde'

```
sns.displot(x = 'SepalLengthCm', y = 'PetalWidthCm', data = iris, kind = 'kde')
plt.show()
```



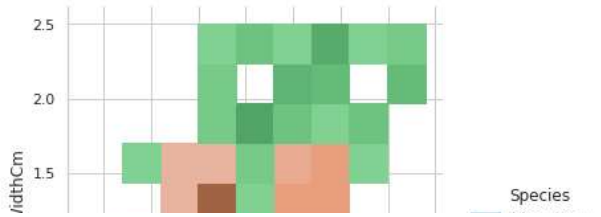
rug

```
sns.displot(x = 'SepalLengthCm', y = 'PetalWidthCm', data = iris, kind = 'kde', rug = True)
plt.show()
```

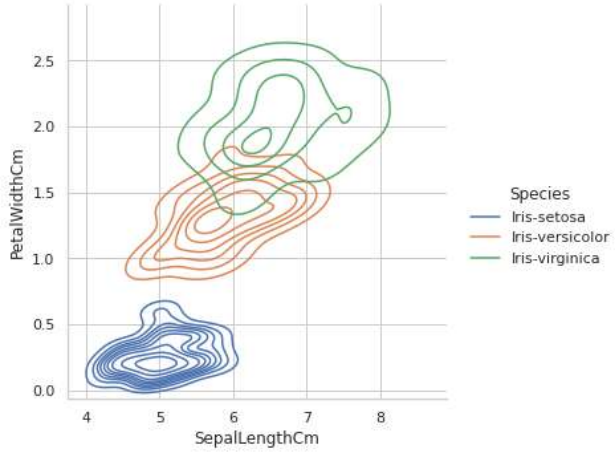


hue

```
sns.displot(x = 'SepalLengthCm', y = 'PetalWidthCm', data = iris, hue = 'Species')
plt.show()
```

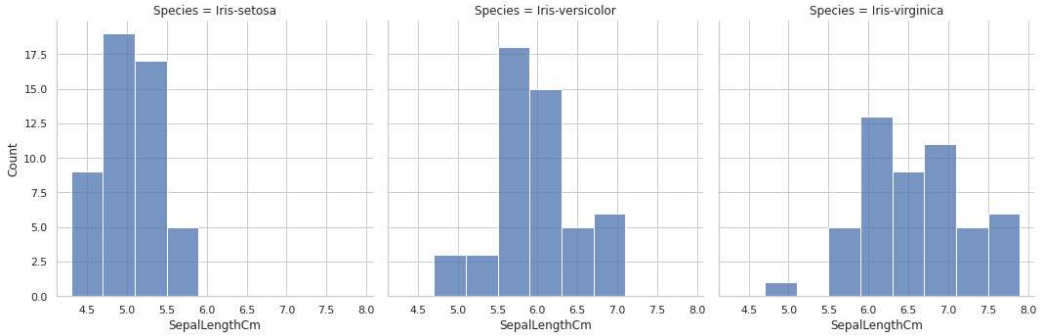



```
sns.displot(x = 'SepalLengthCm', y = 'PetalWidthCm', data = iris, kind = 'kde', hue = 'Species')
plt.show()
```

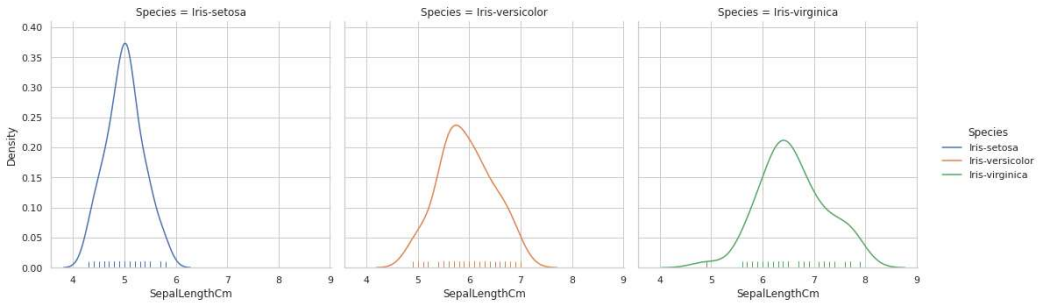


Small Multiples (FacetGrid)

```
sns.displot(x = 'SepalLengthCm', data = iris, col = 'Species')
plt.show()
```



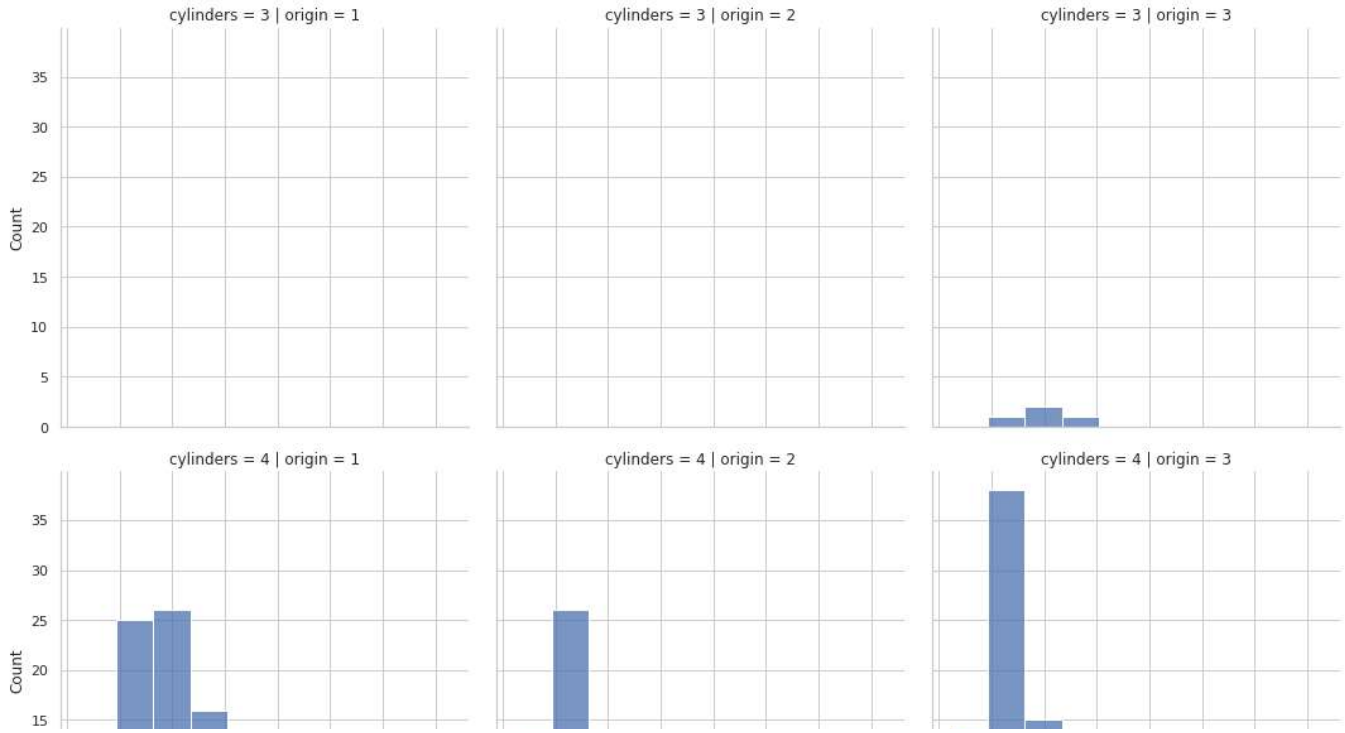

```
sns.displot(x = 'SepalLengthCm', data = iris, col = 'Species', kind = 'kde', rug = True, hue = "Species")
plt.show()
```



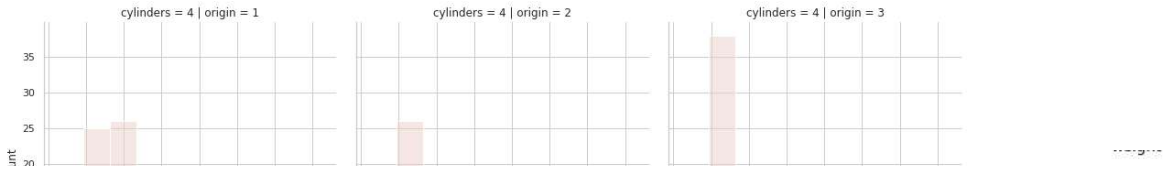
```
car = pd.read_csv('/content/drive/MyDrive/Notes/mpg.csv')
car.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino

```
sns.displot(x = 'weight', col = 'origin', row = 'cylinders', data = car)
plt.show()
```




```
sns.displot(x = 'weight', data = car[car.cylinders.isin([4,6,8])], hue = 'cylinders', col = 'origin', row = 'cylinders')
plt.show()
```



Styling

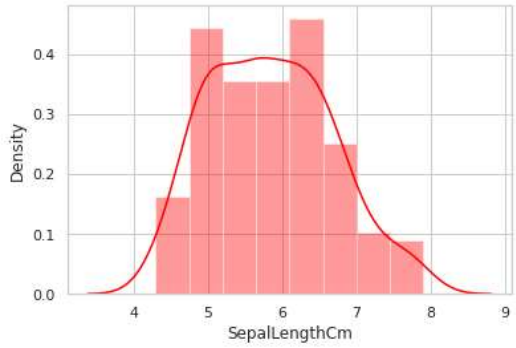
```
iris.head(5)
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

```
import seaborn as sns
sns.distplot(iris['SepalLengthCm'],color = 'red')
sns.set(style = 'ticks')        # style = white, dark, whitegrid, darkgrid, ticks

plt.show()
```

```
/usr/local/lib/python3.8/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be
  warnings.warn(msg, FutureWarning)
```



0s completed at 4:24 PM