MatlibPlot & Seaborn

Session Objectives

At the end of this session, you will be able:

- ❑ Grasp the key concepts in the design of matplotlib
- ❑ Understand subplots
- ❑ Visualize arrays with matplotlib
- ❑ Draw simple line Plots
- ❑ Draw scatter Plots, Histograms
- ❑ Customizing Plot Legends
- ❑ Seaborn Versus Matplotlib

What is 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

If you have Python and PIP already installed on a system, then installation of Matplotlib is very easy.
Install it using this command:

```
In [ ]:  1  !pip install matplotlib
```

Import Matplotlib

Once Matplotlib is installed, import it in your applications by adding the import module statement:

In [ ]:
```
1 import matplotlib
```

Checking Matplotlib Version

The version string is stored under **version** attribute.

In [ ]:
```
1 import matplotlib
2
3 print(matplotlib.__version__)
```

```
3.2.2
```

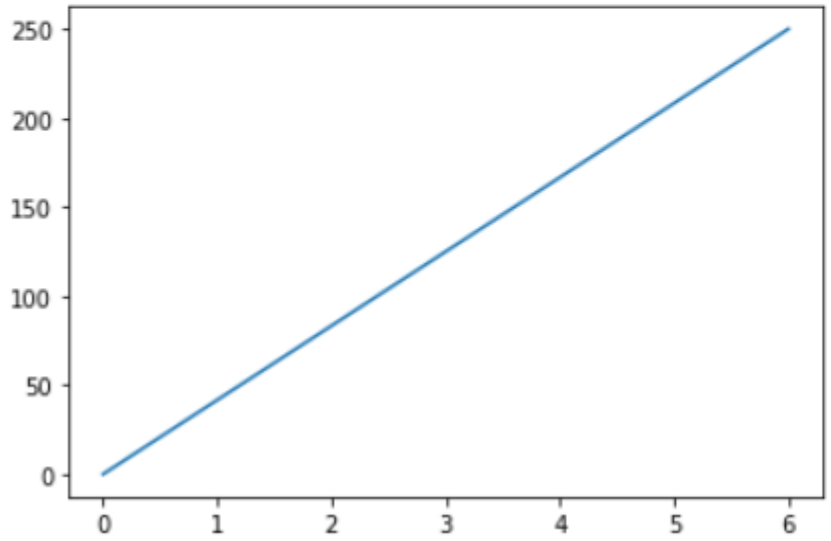
Pyplot

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:

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

Draw a line in a diagram from position (0,0) to position (6,250):

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 xpoints = np.array([0, 6])
5 ypoints = np.array([0, 250])
6 plt.plot(xpoints, ypoints)
7 plt.show()
```



❑ Understand subplots

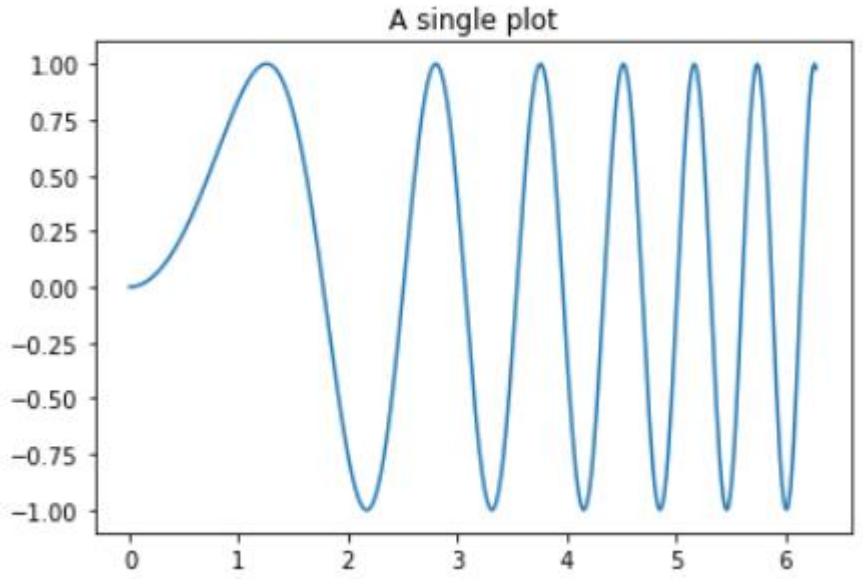
plt.subplots() creates a figure and a grid of subplots with a single call, while providing reasonable control over how the individual plots are created.
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html#matplotlib.pyplot.subplots

A figure with just one subplot

subplots() without arguments returns a Figure and a single Axes.
This is actually the simplest and recommended way of creating a single Figure and Axes.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Some example data to display
5  x = np.linspace(0, 2 * np.pi, 400)
6  y = np.sin(x ** 2)
```

```
1  fig, ax = plt.subplots()
2  ax.plot(x, y)
3  ax.set_title('A single plot')
4
```

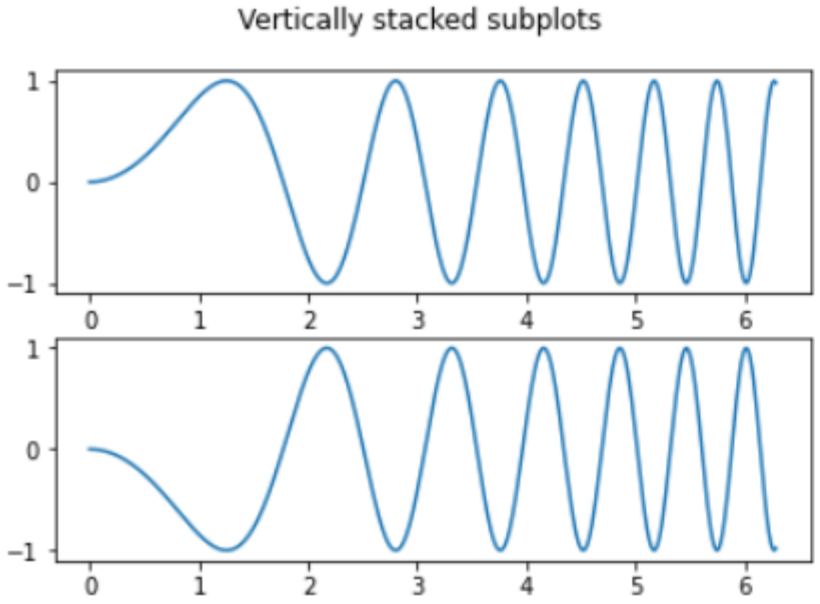


Stacking subplots in one direction

The first two optional arguments of plt.subplots define the number of rows and columns of the subplot grid.
When stacking in one direction only, the returned axs is a 1D numpy array containing the list of created Axes.

```
1  fig, axs = plt.subplots(2)
2  fig.suptitle('Vertically stacked subplots')
3  axs[0].plot(x, y)
4  axs[1].plot(x, -y)
```

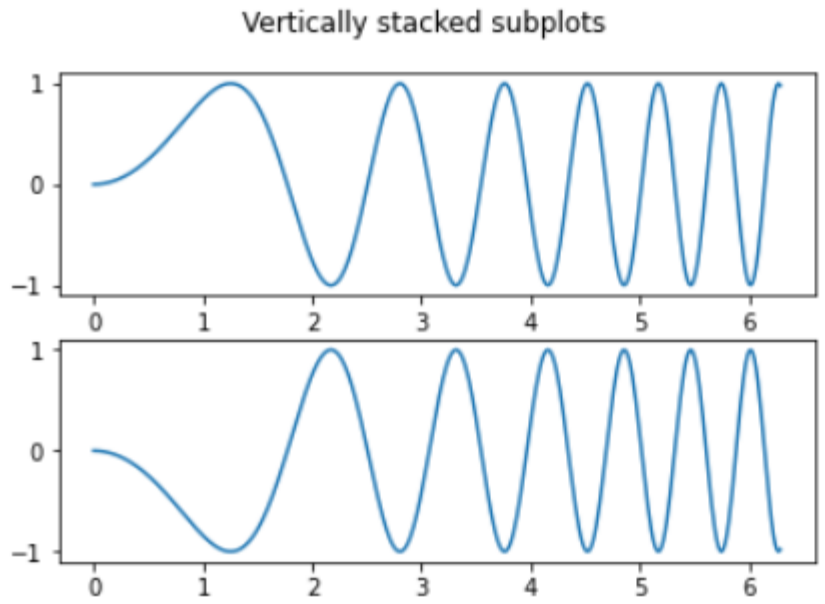
```
[<matplotlib.lines.Line2D at 0x1d97fe89dc8>]
```



If you are creating just a few Axes, it's handy to unpack them immediately to dedicated variables for each Axes. That way, we can use ax1 instead of the more verbose ax[0]

```
1 fig, (ax1, ax2) = plt.subplots(2)
2 fig.suptitle('Vertically stacked subplots')
3 ax1.plot(x, y)
4 ax2.plot(x, -y)
```

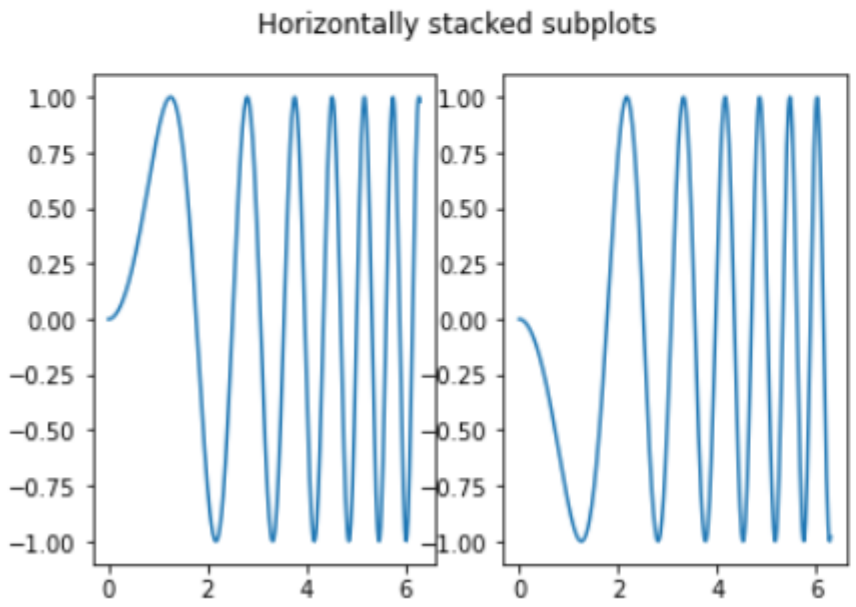
```
[<matplotlib.lines.Line2D at 0x1d9480bdc88>]
```



To obtain side-by-side subplots, pass parameters 1,2 for one row and two columns.

```
1 fig, (ax1, ax2) = plt.subplots(1, 2)
2 fig.suptitle('Horizontally stacked subplots')
3 ax1.plot(x, y)
4 ax2.plot(x, -y)
```

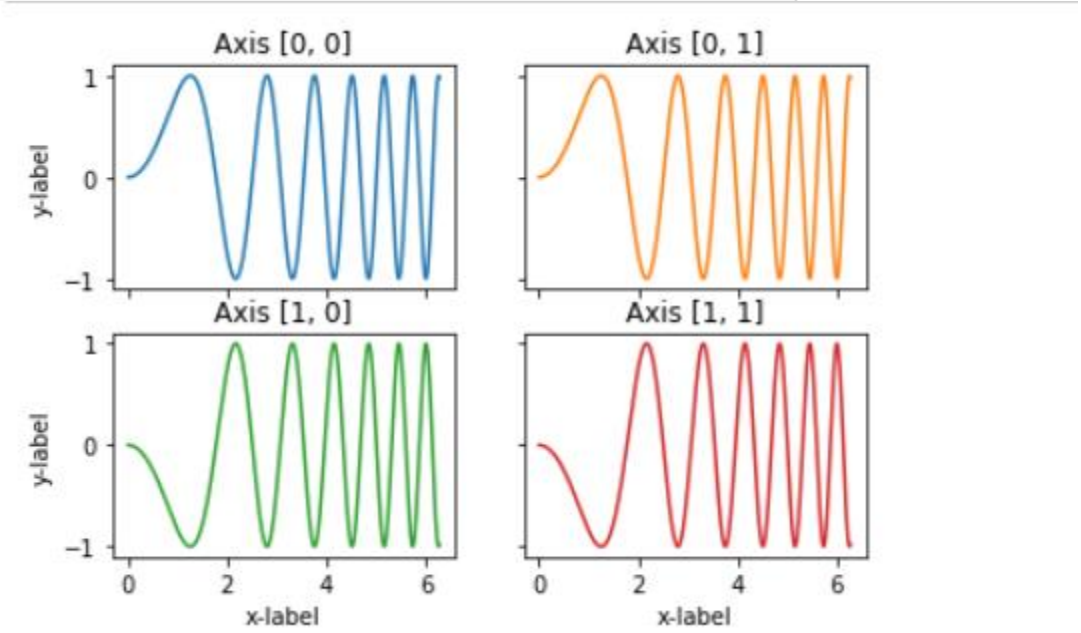
```
[<matplotlib.lines.Line2D at 0x1d97fde9048>]
```



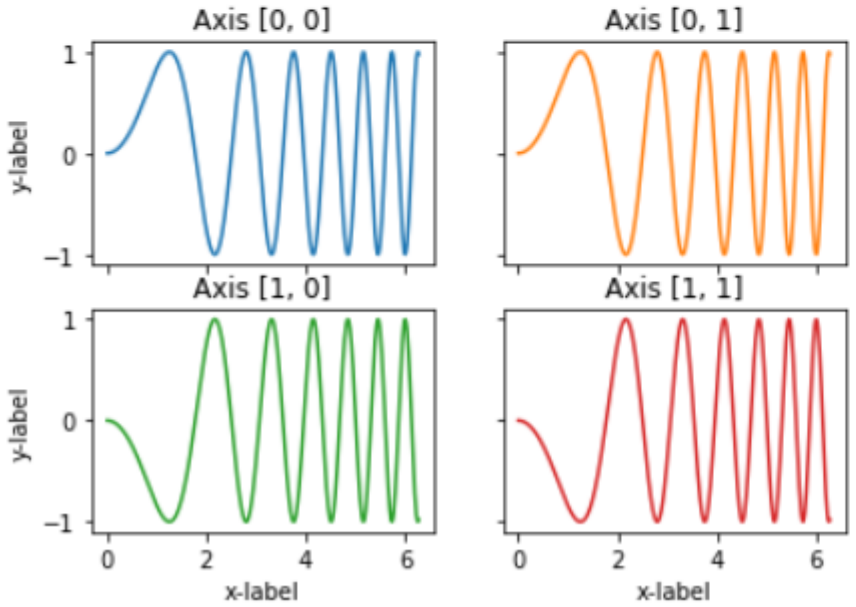
- **Stacking subplots in two directions**

  When stacking in two directions, the returned axs is a 2D NumPy array.
  If you have to set parameters for each subplot it's handy to iterate over all subplots in a 2D grid using for ax in axs.flat

```
fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Axis [0, 0]')
axs[0, 1].plot(x, y, 'tab:orange')
axs[0, 1].set_title('Axis [0, 1]')
axs[1, 0].plot(x, -y, 'tab:green')
axs[1, 0].set_title('Axis [1, 0]')
axs[1, 1].plot(x, -y, 'tab:red')
axs[1, 1].set_title('Axis [1, 1]')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```



```python
fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Axis [0, 0]')
axs[0, 1].plot(x, y, 'tab:orange')
axs[0, 1].set_title('Axis [0, 1]')
axs[1, 0].plot(x, -y, 'tab:green')
axs[1, 0].set_title('Axis [1, 0]')
axs[1, 1].plot(x, -y, 'tab:red')
axs[1, 1].set_title('Axis [1, 1]')

# Hide x labels and tick labels for top plots and y ticks for right plots.
for ax in axs.flat:
    ax.label_outer()
```



❑ simple line Plots

Plotting x and y points

By default, the plot() function draws a line from point to point.

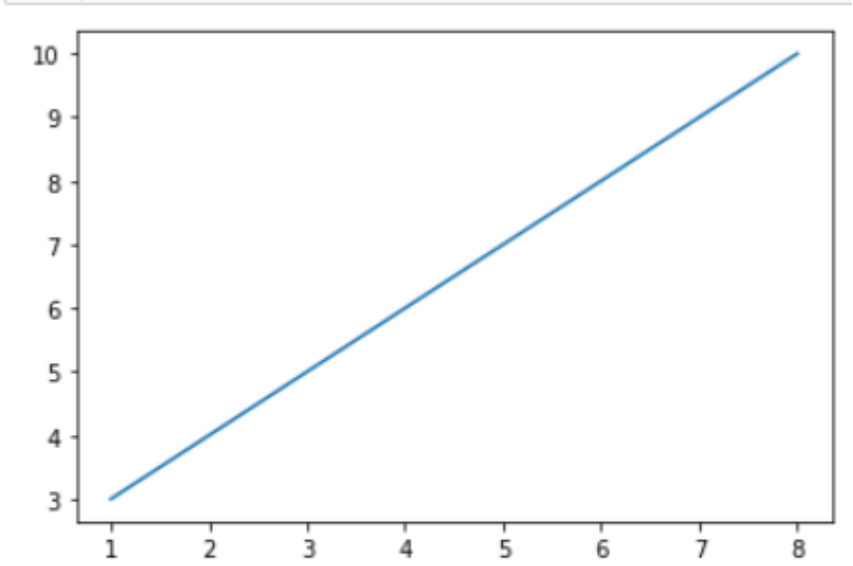
The function takes parameters for specifying points in the diagram.

Parameter 1 is an array containing the points on the x-axis.

Parameter 2 is an array containing the points on the y-axis.

If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  xpoints = np.array([1, 8])
5  ypoints = np.array([3, 10])
6
7  plt.plot(xpoints, ypoints)
8  plt.show()
```



plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

The x-axis is the horizontal axis.

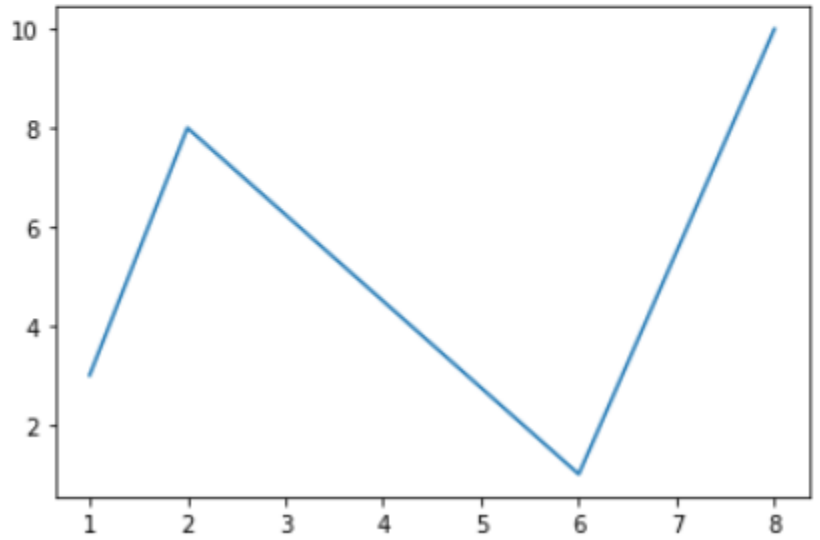
The y-axis is the vertical axis.

Multiple Points

You can plot as many points as you like, just make sure you have the same number of points in both axis.

Draw a line in a diagram from position (1, 3) to (2, 8) then to (6, 1) and finally to position (8, 10):

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  xpoints = np.array([1, 2, 6, 8])
5  ypoints = np.array([3, 8, 1, 10])
6
7  plt.plot(xpoints, ypoints)
8  plt.show()
```

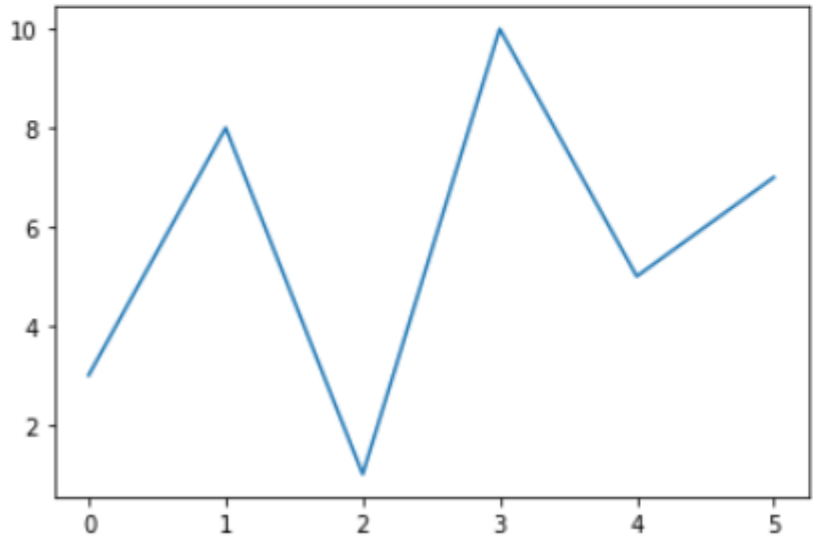


Default X-Points

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points.

So, if we take the same example as above, and leave out the x-points, the diagram will look like this:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 ypoints = np.array([3, 8, 1, 10, 5, 7])
5
6 plt.plot(ypoints)
7 plt.show()
```



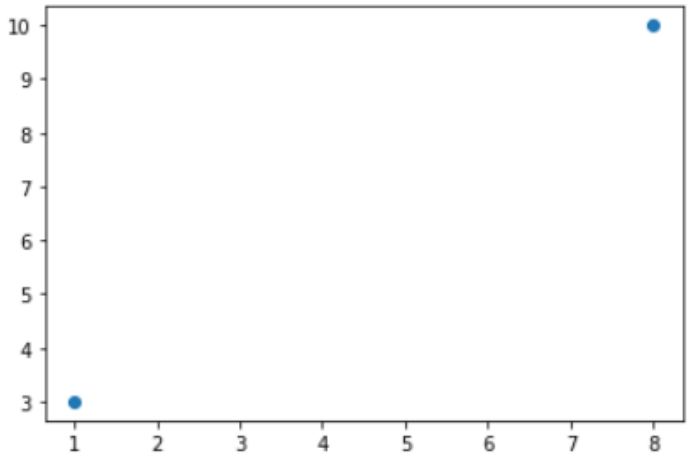
plot([x], y, [fmt], *, data=None, **kwargs) **plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)**

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html?highlight=matplotlib%20pyplot%20plot#matplotlib.pyplot.plot

The optional parameter *fmt* is a convenient way for defining basic formatting like color, marker and linestyle.

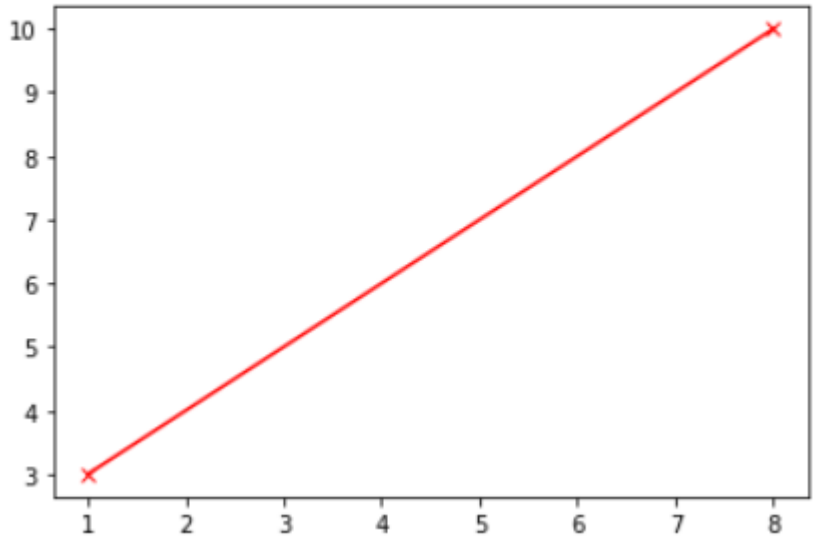
Draw two points in the diagram, one at position (1, 3) and one in position (8, 10):

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 xpoints = np.array([1, 8])
5 ypoints = np.array([3, 10])
6
7 plt.plot(xpoints, ypoints, 'o')
8 plt.show()
```



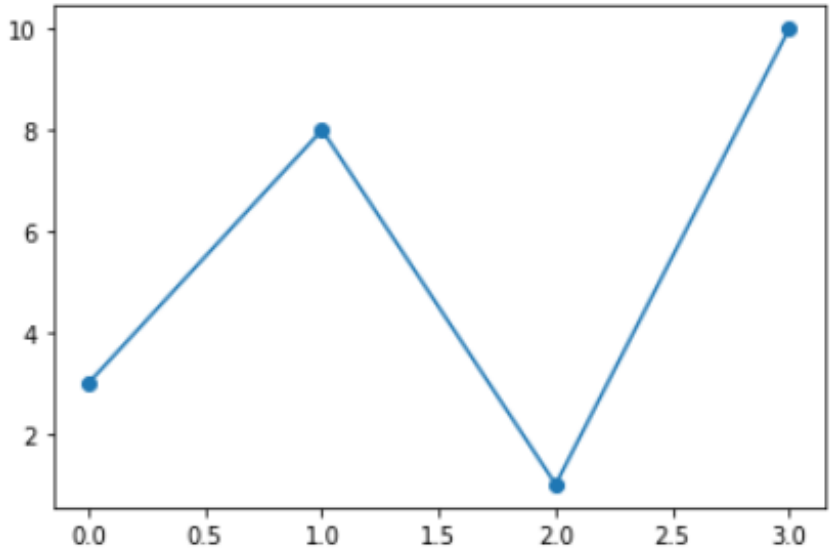
Draw Red straight line and make markers at the main points:

```
7 plt.plot(xpoints, ypoints, 'rx-')
8 plt.show()
```



'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down

```
1 ypoints = np.array([3, 8, 1, 10])
2
3 plt.plot(ypoints, marker = 'o')
4 plt.show()
```



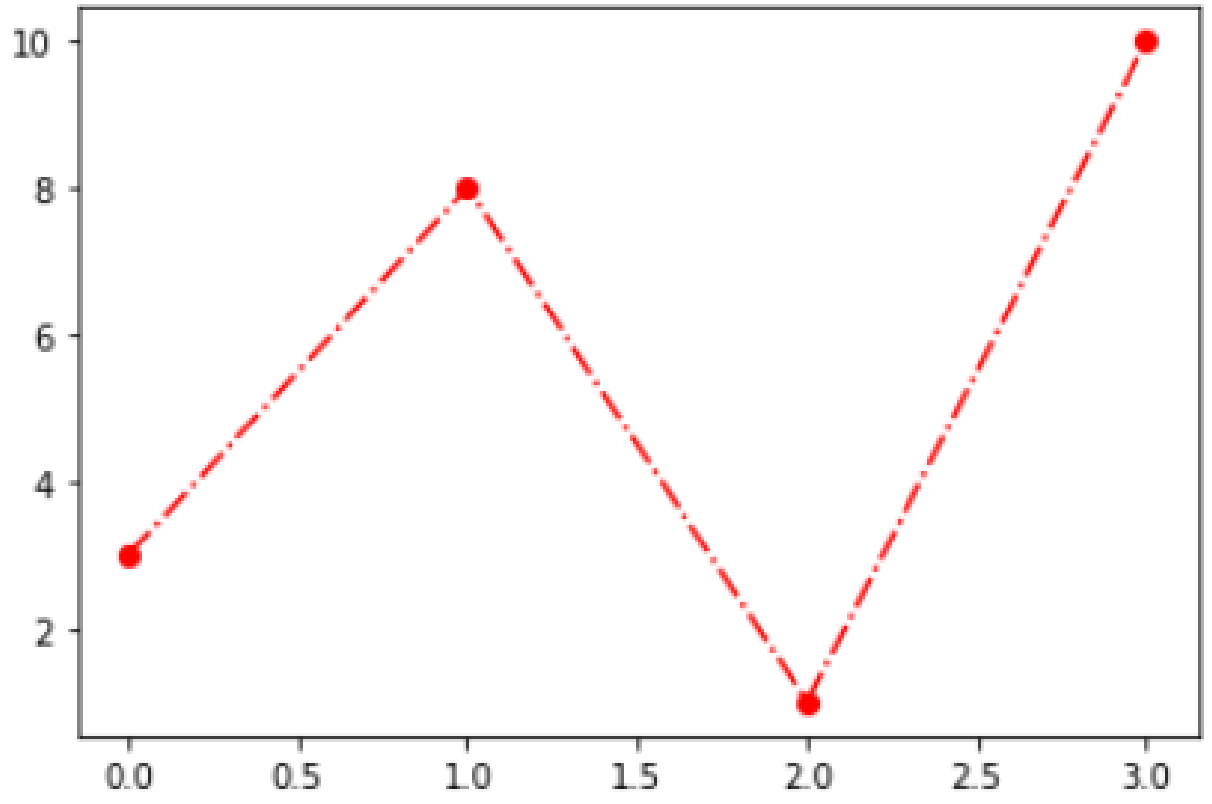
Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

```
1 ypoints = np.array([3, 8, 1, 10])
2
3 plt.plot(ypoints, 'o-.r')
4 plt.show()
```

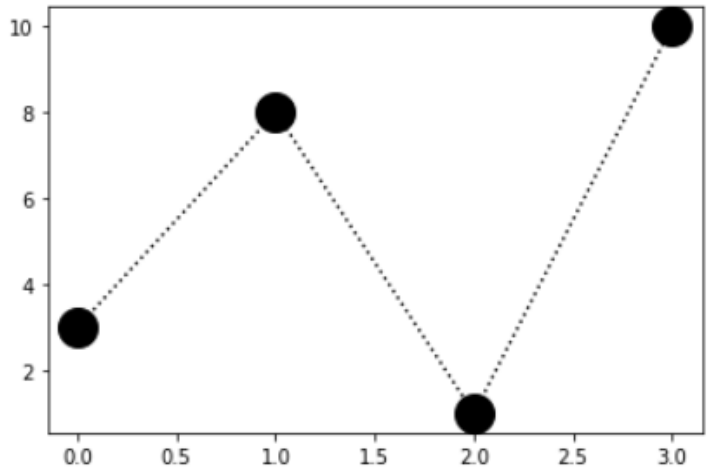


Marker Size

You can use the keyword argument **markersize** or the shorter version, ms to set the size of the markers:

Set the size of the markers to 20:

```
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, 'o:k', ms = 20)
plt.show()
```

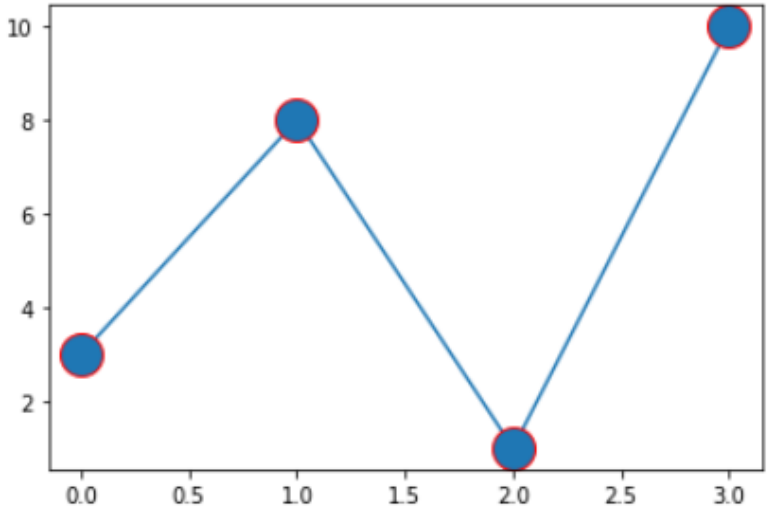


Marker Color

You can use the keyword argument **markeredgecolor** or the shorter **mec** to set the color of the edge of the markers:

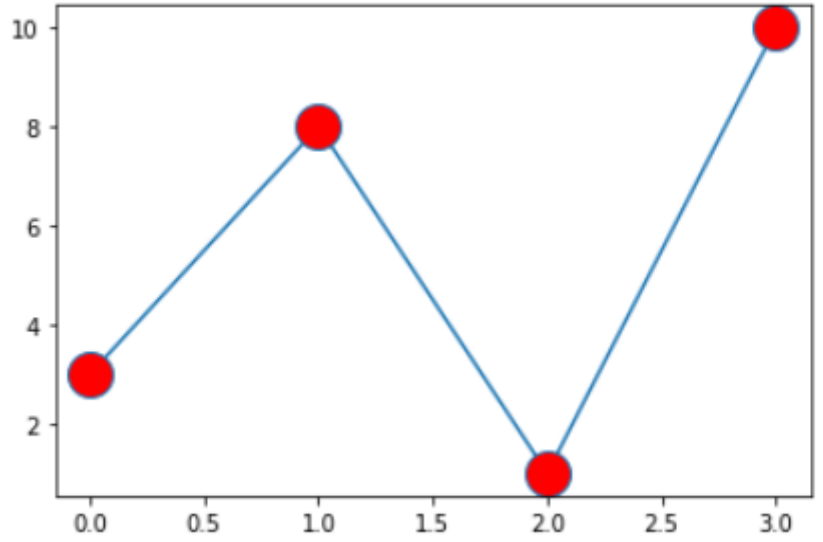
Set the EDGE color to red:

```
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r')
plt.show()
```

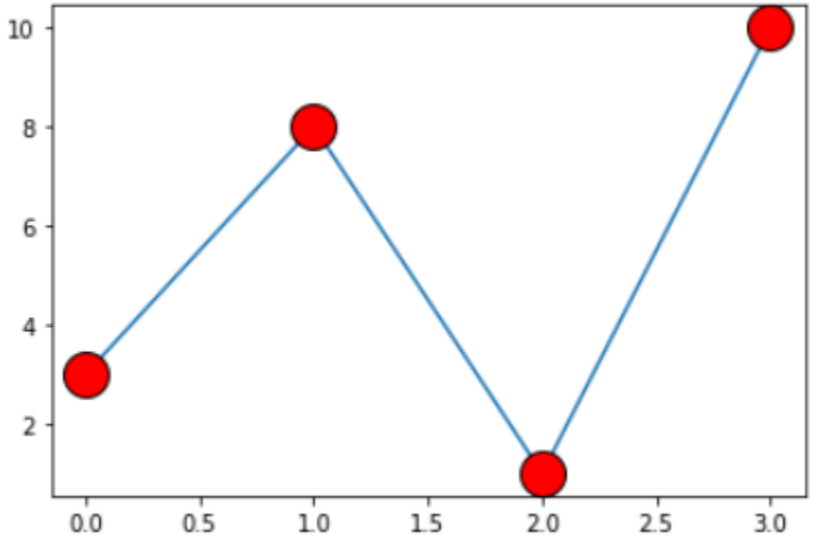


You can use the keyword argument **markerfacecolor** or the shorter **mfc** to set the color inside the edge of the markers:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mfc = 'r')
plt.show()
```



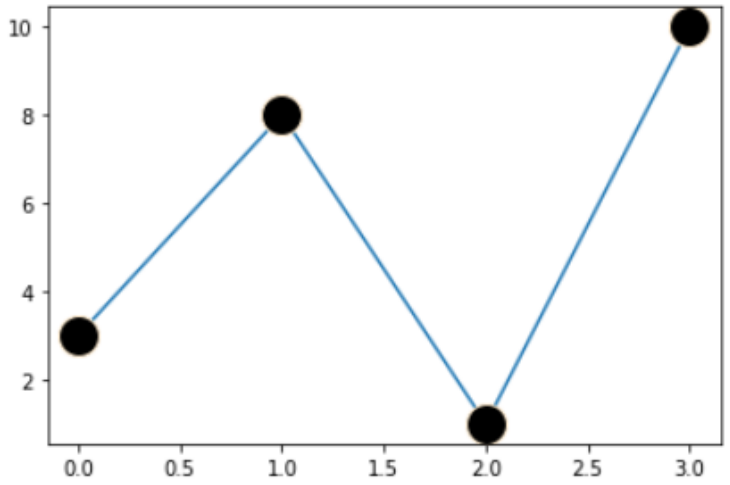

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'k', mfc = 'r')
plt.show()
```



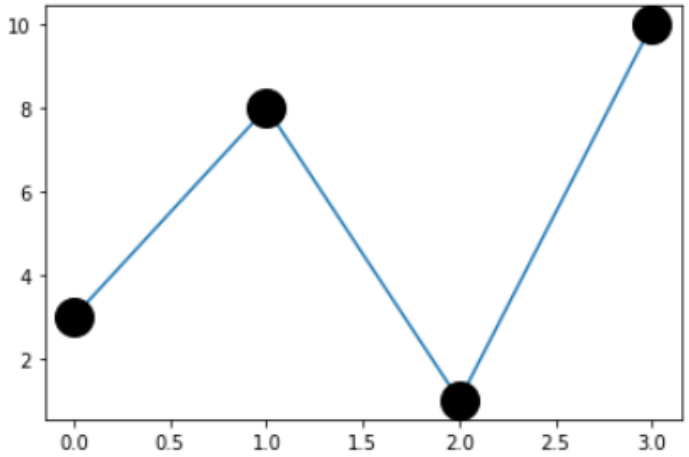
Also , you can use the color hex value or name .
Check the link below:

https://www.w3schools.com/colors/colors_names.asp

```
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = '#FFEBCD', mfc = 'k')
plt.show()
```



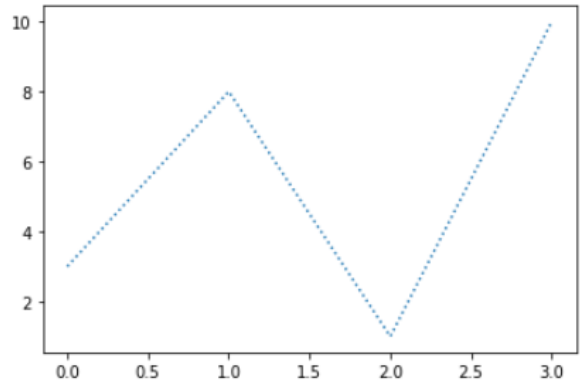

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'Black', mfc = 'Black')
plt.show()
```



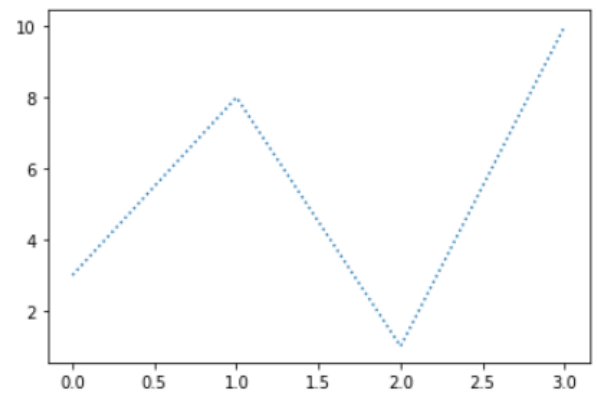
Linestyle

You can use the keyword argument **linestyle**, or shorter **ls**, to change the style of the plotted line:

```
6 plt.plot(ypoints, linestyle = 'dotted')
7 plt.show()
```




```
8 plt.plot(ypoints, ls = ':')
9 plt.show()
```



Line Styles

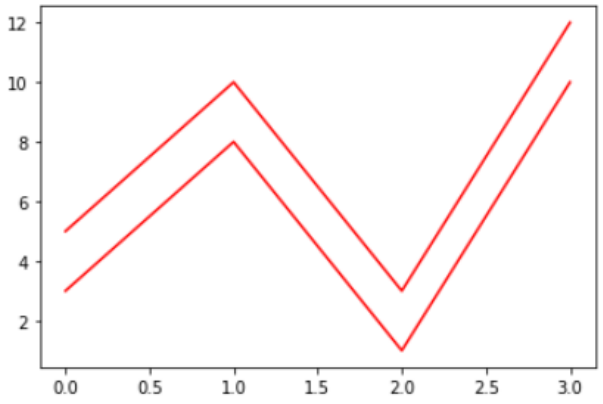
You can choose any of these styles:

Style	Or
'solid' (default)	'-'
'dotted'	':'
'dashed'	'--'
'dashdot'	'-.'
'None'	'' or ' '

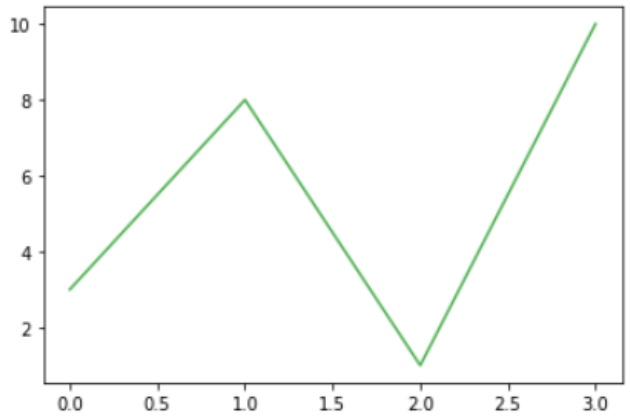
Line Color

You can use the keyword argument color or the shorter c to set the color of the line:

```
1 ypoints = np.array([3, 8, 1, 10])
2 plt.plot(ypoints, color = 'red')
3 plt.plot(ypoints+2, color = 'r')
4 plt.show()
```



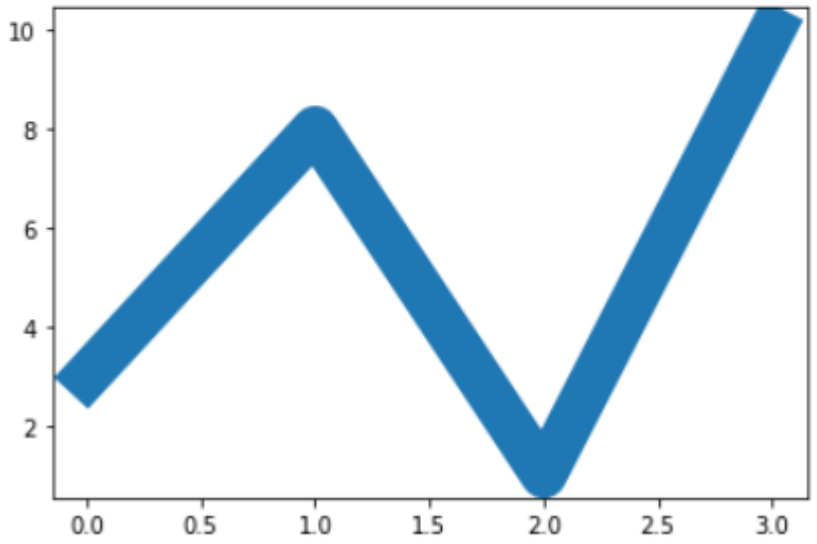

```
1 ypoints = np.array([3, 8, 1, 10])
2 plt.plot(ypoints,  c = '#4CAF50')
3 plt.show()
```



Line Width

You can use the keyword argument **linewidth** or the shorter **lw** to change the width of the line.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  ypoints = np.array([3, 8, 1, 10])
5
6  plt.plot(ypoints, linewidth = '20.5')
7  plt.show()
```

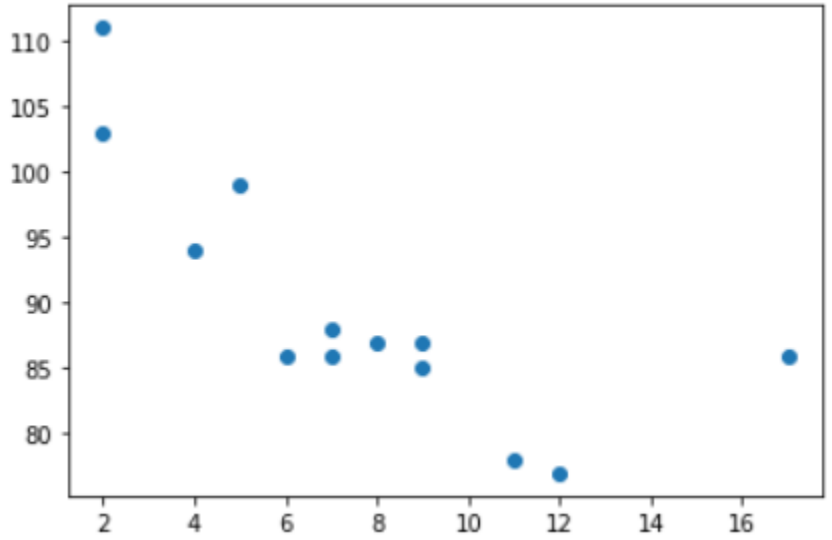


❑ Scatter Plots

The **scatter()** method in the matplotlib library is used to draw a scatter plot. Scatter plots are widely used to represent relation among variables and how change in one affects the other.

plt.scatter(x_axis_data, y_axis_data, s=None, c=None, marker=None, cmap=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors=None)

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
5  y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
6
7  plt.scatter(x, y)
8  plt.show()
```



The observation in the example above is the result of 13 cars passing by.
The X-axis shows how old the car is.
The Y-axis shows the speed of the car when it passes.

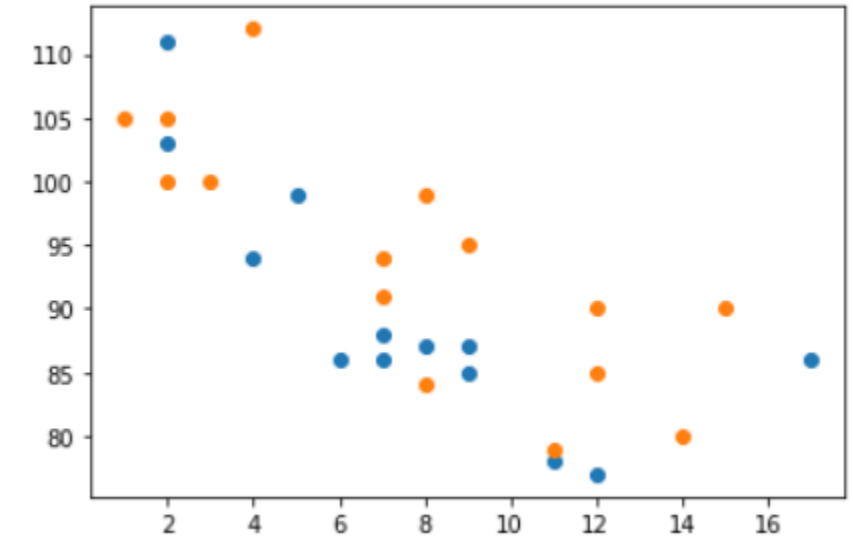
Are there any relationships between the observations?

It seems that the newer the car, the faster it drives, but that could be a coincidence, after all we only registered 13 cars.

Compare Plots

In the example above, there seems to be a relationship between speed and age, but what if we plot the observations from another day as well? Will the scatter plot tell us something else?

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

#day one, the age and speed of 13 cars:
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)

#day two, the age and speed of 15 cars:
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y)

plt.show()
```



By comparing the two plots, I think it is safe to say that they both gives us the same conclusion: the newer the car, the faster it drives

plt.scatter(x_axis_data, y_axis_data, s=None, c=None, marker=None, cmap=None, vmin=None, vmax=None, alpha=None, linewidths=None, edgecolors=None)

- **x_axis_data-** An array containing x-axis data
- **y_axis_data-** An array containing y-axis data
- **s-** marker size (can be scalar or array of size equal to size of x or y)
- **c-** color of sequence of colors for markers
- marker- marker style
- **cmap-** cmap name
- **linewidths-** width of marker border
- **edgecolor-** marker border color
- **alpha-** blending value, between 0 (transparent) and 1 (opaque)

Colormap

https://matplotlib.org/stable/tutorials/colors/colormaps.html

The Matplotlib module has a number of available colormaps.

A colormap is like a list of colors, where each color has a value that ranges from 0 to 100.

cmap_list:

```
: 1 import matplotlib.cm as cm
  2 cm.cmaps_listed
```

```
: {'magma': <matplotlib.colors.ListedColormap at 0x233a125a708>,
   'inferno': <matplotlib.colors.ListedColormap at 0x233a125a7c8>,
   'plasma': <matplotlib.colors.ListedColormap at 0x233a125a808>,
   'viridis': <matplotlib.colors.ListedColormap at 0x233a125a848>,
   'cividis': <matplotlib.colors.ListedColormap at 0x233a125a888>,
   'twilight': <matplotlib.colors.ListedColormap at 0x233a125a8c8>,
   'twilight_shifted': <matplotlib.colors.ListedColormap at 0x233a125a908>,
   'turbo': <matplotlib.colors.ListedColormap at 0x233a125a948>}
```

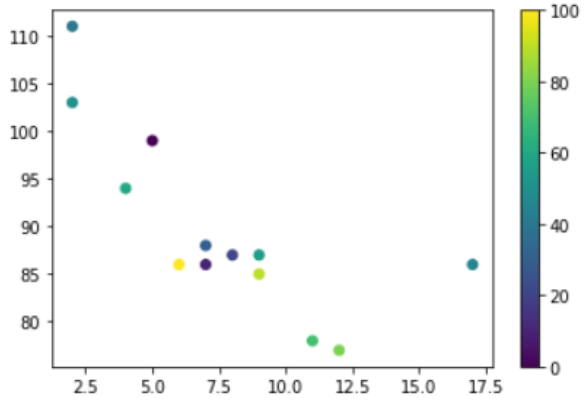
Here is an example of a colormap:

This colormap is called 'viridis' and as you can see it ranges from 0, which is a purple color, and up to 100, which is a yellow color.

How to Use the ColorMap You can specify the colormap with the keyword argument **cmap** with the value of the colormap, in this case 'viridis' which is one of the built-in colormaps available in Matplotlib.

In addition you have to create an array with values (from 0 to 100), one value for each of the point in the scatter plot:

```
:  1 x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
   2 y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
   3 colors = np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])
   4
   5 plt.scatter(x, y, c=colors, cmap='viridis')
   6 plt.colorbar()
   7 plt.show()
```



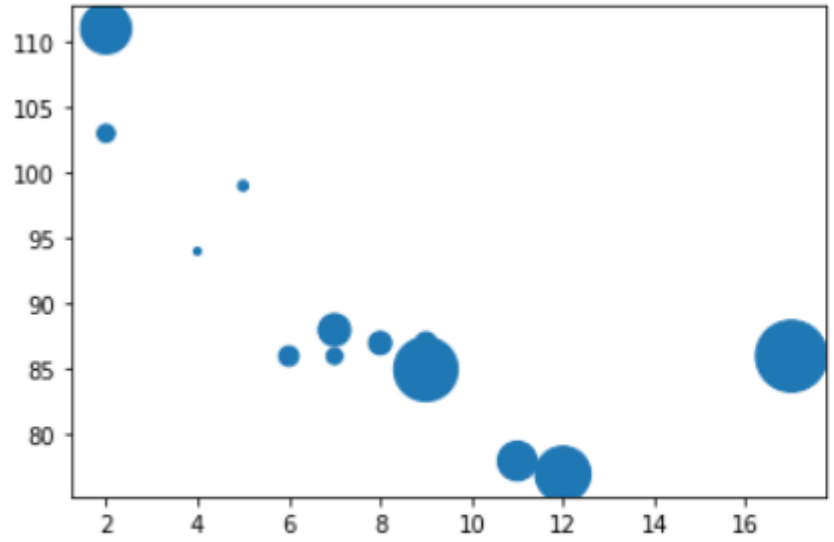
Size

You can change the size of the dots with the s argument.

Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:

```
:
```

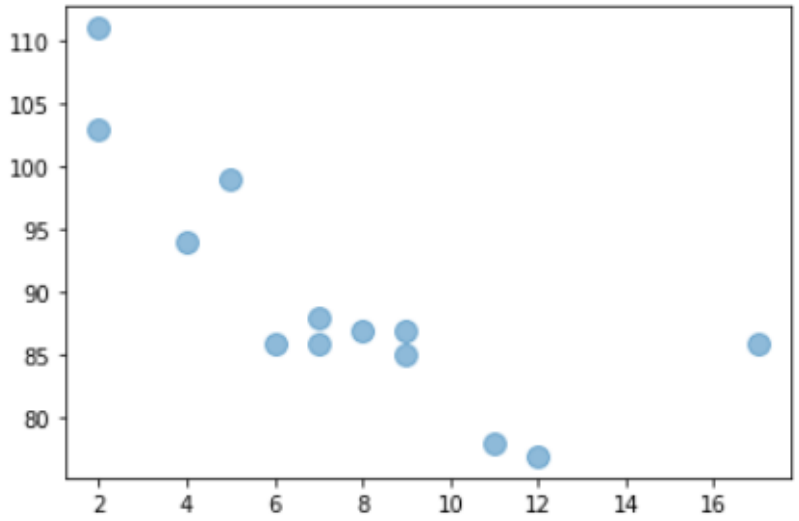
```python
import matplotlib.pyplot as plt
import numpy as np

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])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])

plt.scatter(x, y, s=sizes)

plt.show()
```



Alpha

You can adjust the transparency of the dots with the alpha argument.
Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis

```
#Set your own size for the markers:

import matplotlib.pyplot as plt
import numpy as np

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])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])

plt.scatter(x, y, s=100, alpha=0.5)

plt.show()
```

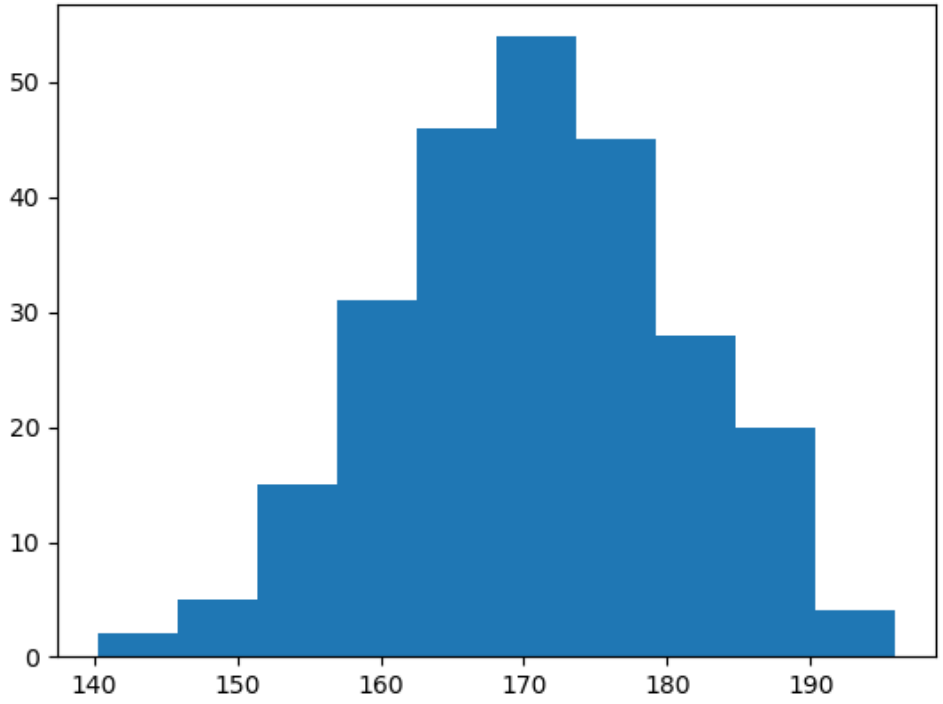


❑ Histogram

plt.hist(x, bins=None, range=None, density=False, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, *, data=None, **kwargs)[so

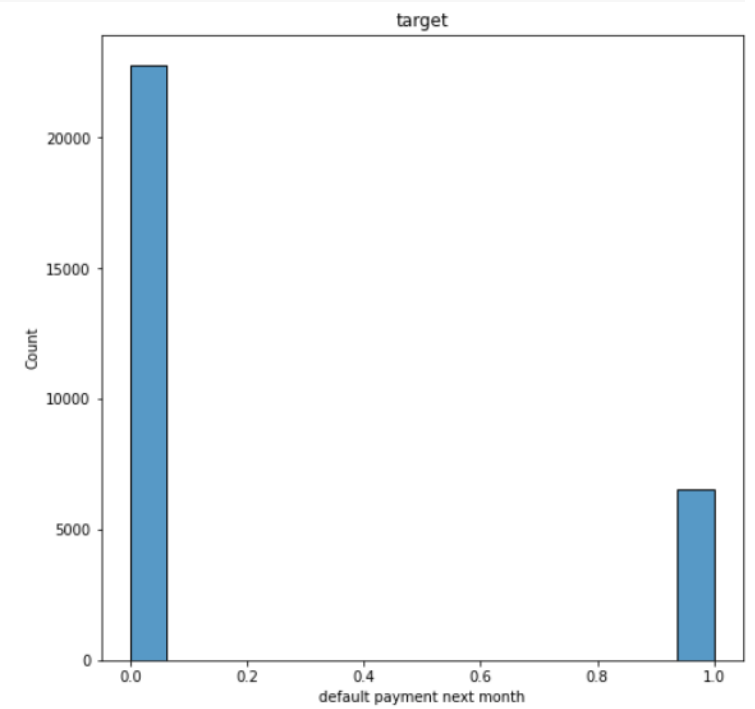
A histogram is a graph showing frequency distributions.
It is a graph showing the number of observations within each given interval.
Example: Say you ask for the height of 250 people, you might end up with a histogram like this:

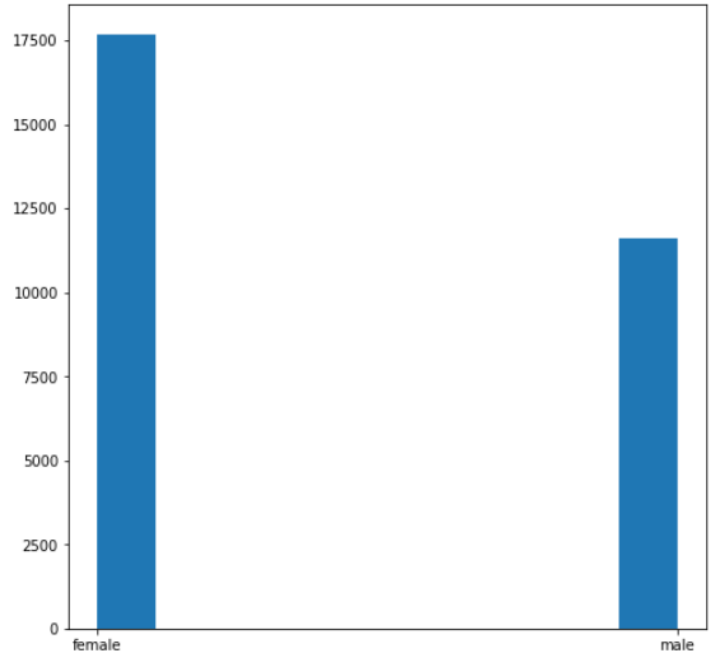
You can read from the histogram that there are approximately:
2 people from 140 to 145cm
5 people from 145 to 150cm
15 people from 151 to 156cm
31 people from 157 to 162cm
46 people from 163 to 168cm
53 people from 168 to 173cm
45 people from 173 to 178cm
28 people from 179 to 184cm
21 people from 185 to 190cm
4 people from 190 to 195cm



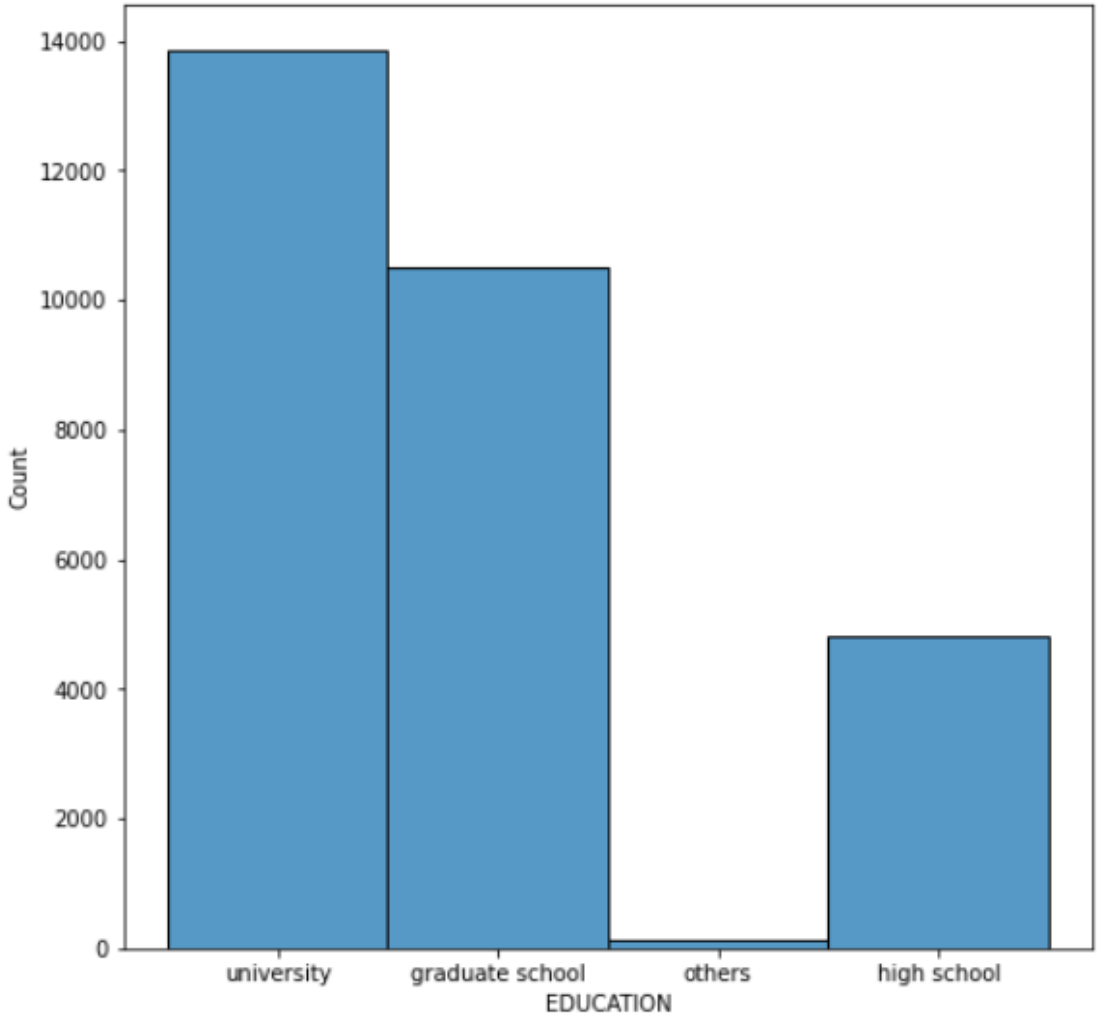
Also, at Data Science Histogram plot helps to check if data balanced or imbalanced

```
:  1  plt.figure(figsize=(8,8))
   2  target=sns.histplot(data_After_pro["default payment next month"])
   3  target.set_title("target")
```






```
1  plt.figure(figsize=(8,8))
2  plt.hist(data_A_filteration["SEX"])
3  plt.show()
```


14000
12000
10000
8000
6000
4000
2000
0
Count
university
graduate school
others
high school
EDUCATION

❑ Customizing Plot Legends

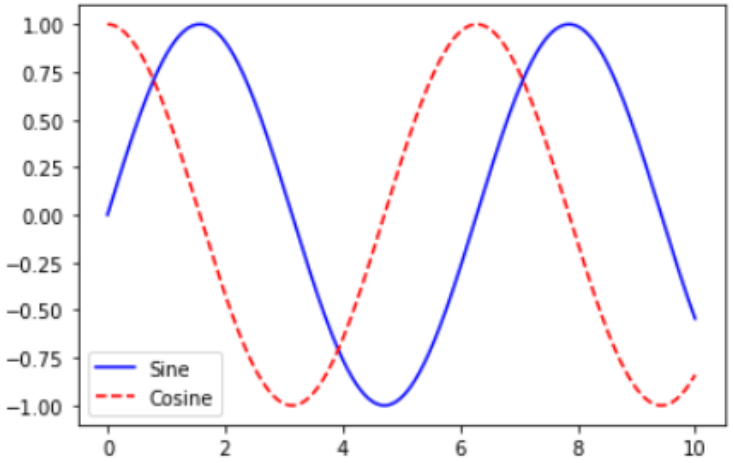
Plot legends give meaning to a visualization, assigning meaning to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib.

The simplest legend can be created with the plt.legend() command, which automatically creates a legend for any labeled plot elements:

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

```
%matplotlib inline
import numpy as np
```

```
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x), '-b', label='Sine')
plt.plot(x, np.cos(x), '--r', label='Cosine')
leg = plt.legend()
```



Characteristics	Matplotlib	Seaborn
Use Cases	Matplotlib plots various graphs using Pandas and Numpy	Seaborn is the extended version of Matplotlib which uses Matplotlib along with Numpy and Pandas for plotting graphs
Complexity of Syntax	It uses comparatively complex and lengthy syntax.	It uses comparatively simple syntax which is easier to learn and understand.
Multiple figures	Matplotlib has multiple figures can be opened	Seaborn automates the creation of multiple figures which sometimes leads to out of memory issues
Flexibility	Matplotlib is highly customizable and powerful.	Seaborn avoids a ton of boilerplate by providing default themes which are commonly used.

❑ Seaborn Plots

VIP: https://www.kaggle.com/learn/data-visualization

Exploring Seaborn Plots

The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

Let's take a look at a few of the datasets and plot types available in Seaborn. Note that all of the following *could* be done using raw Matplotlib commands (this is, in fact, what Seaborn does under the hood) but the Seaborn API is much more convenient.

data set existed at seaborn : https://github.com/mwaskom/seaborn-data

Download seaborn package

```
1 !pip install seaborn
```

Seaborn usually imported under the sns alias

```
1 import seaborn as sns
```

Seaborn Data visualization

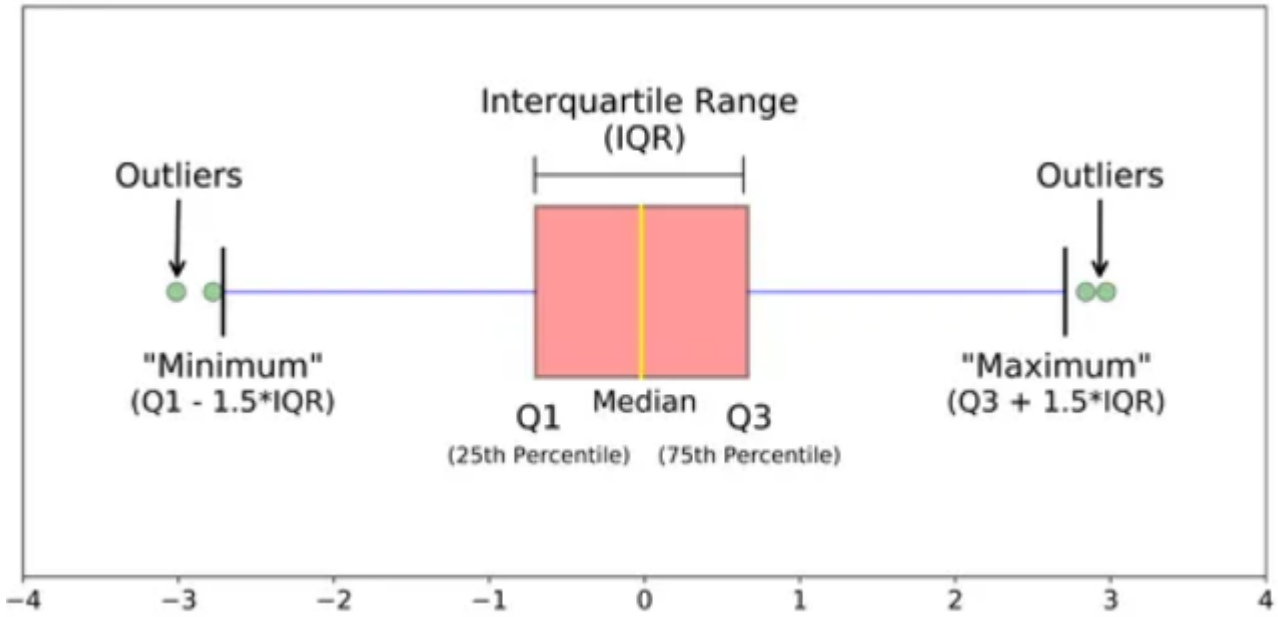
Trends (time series)	Relationships	distribution	Outliers
Sns.lineplot(data)	sns.barplot(x,y)	sns.distplot(x,kde="false")	sns.boxplot(data)
	sns.heatmap(data)	sns.kdeplot(data)	
	sns.scatterplot(x,y,hue)	sns.jointplot(x,y,kind="kde")	
	sns.lmplot(data,x,y,hue)	sns.histplot()	
	sns.swarmplot(x,y)		
	sns.regplot(x,y) sns.pairplot(data)		

VIP: https://www.kaggle.com/learn/data-visualization **mandatory**

Outliers

seaborn.boxplot(*, x=None, y=None, hue=None, data=None, order=None, hue_order=None, orient=None, color=None, palette=None, saturation=0.75, width=0.8, dodge=True, fliersize=5, linewidth=None, whis=1.5, ax=None, **kwargs)

Example : Age >100 or < 0



```
1 import seaborn as sns
```

```
1 data = sns.load_dataset('tips')
```

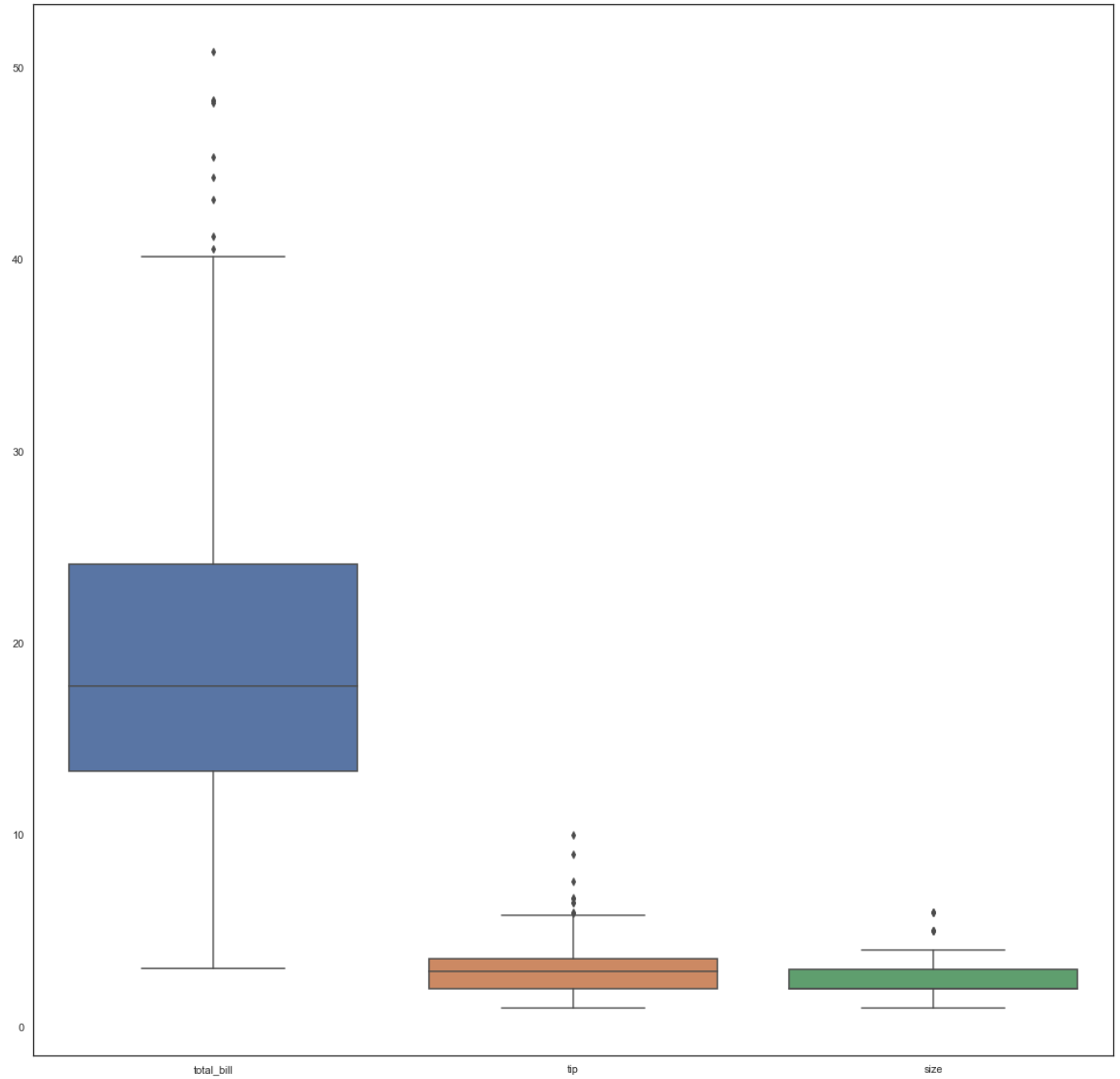
```
1 data.head()
```

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

Data[“total_bill”] >>>

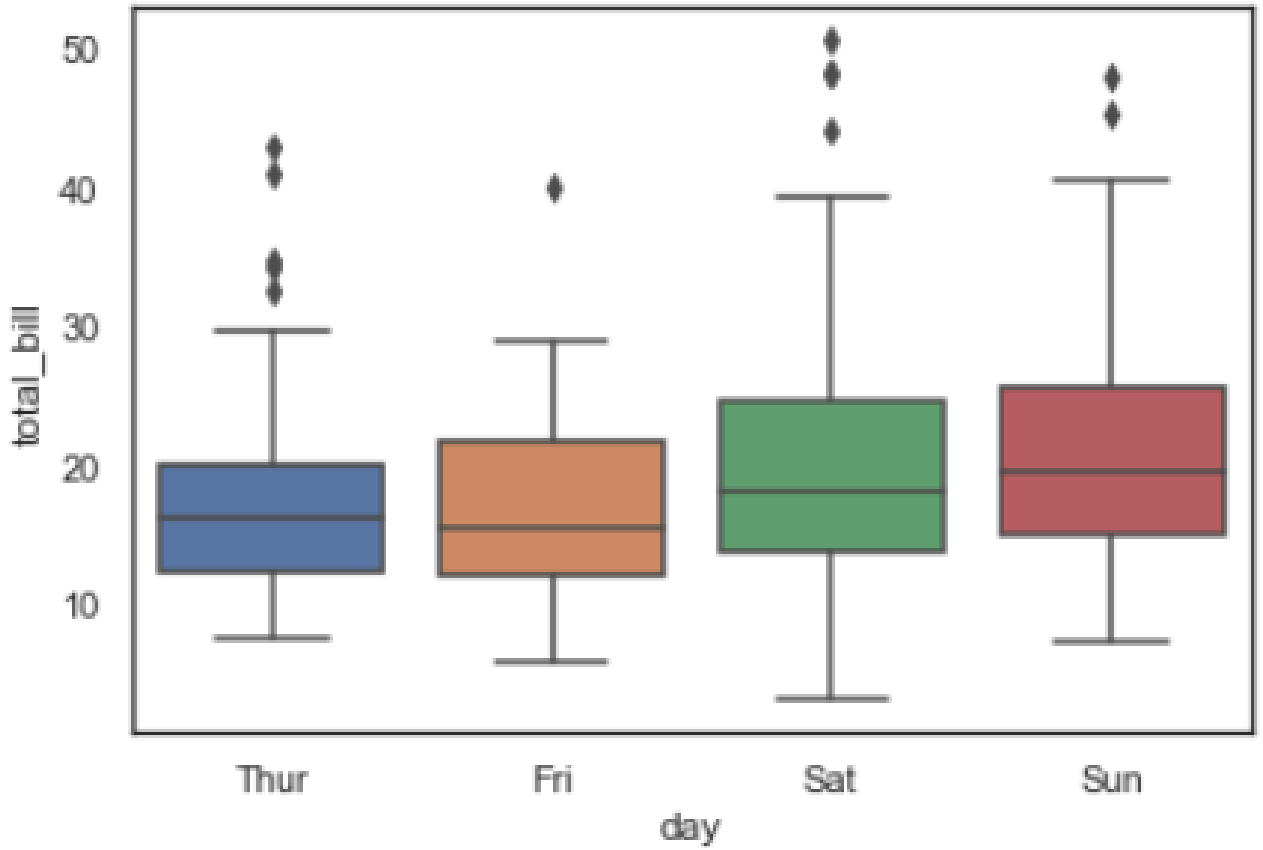
total_bill
16.99
10.34
21.01
23.68
24.59

:
```
1 import seaborn as sns
2 sns.set_theme(style="white")
3 tips = sns.load_dataset("tips")
4 plt.figure(figsize=(20,20))
5 sns.boxplot(data=tips)
6 plt.show()
```


50
40
30
20
10
0
total_bill
tip
size

:

```
1 ax = sns.boxplot(x="day", y="total_bill", data=tips)
```



Scipy

Session Objectives

At the end of this session, you will be able:

- ❑ Use linear algebra SciPy package
- ❑ Use optimization SciPy package
- ❑ Use interpolation SciPy package

What is SciPy?
SciPy is a scientific computation library that uses NumPy underneath.
SciPy stands for Scientific Python.
It provides more utility functions for optimization, stats and signal processing.
Like NumPy, SciPy is open source so we can use it freely.
SciPy was created by NumPy's creator Travis Olliphant.

Why Use SciPy?
If SciPy uses NumPy underneath, why can we not just use NumPy?
SciPy has optimized and added functions that are frequently used in NumPy and Data Science.

Which Language is SciPy Written in?
SciPy is predominantly written in Python, but a few segments are written in C

SciPy Installation

In [ ]:
```
!pip install scipy
```

```
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages
(1.4.1)
Requirement already satisfied: numpy>=1.13.3 in /usr/local/lib/python3.7/dist-p
ackages (from scipy) (1.19.5)
```

In [ ]:
```
import scipy

print(scipy.__version__)
```

```
1.4.1
```

SciPy Constants

A list of all units under the constants module can be seen using the dir() function

In [ ]:

```
1 from scipy import constants
2
3 print(dir(constants))
```

```
['Avogadro', 'Boltzmann', 'Btu', 'Btu_IT', 'Btu_th', 'ConstantWarning', 'G', 'Julian_year', 'N_A', 'Planck', 'R', 'Rydberg', 'S
tefan_Boltzmann', 'Wien', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package_
_', '__path__', '__spec__', '_obsolete_constants', 'absolute_import', 'acre', 'alpha', 'angstrom', 'arcmin', 'arcminute', 'arcs
ec', 'arcsecond', 'astronomical_unit', 'atm', 'atmosphere', 'atomic_mass', 'atto', 'au', 'bar', 'barrel', 'bbl', 'blob', 'c',
'calorie', 'calorie_IT', 'calorie_th', 'carat', 'centi', 'codata', 'constants', 'convert_temperature', 'day', 'deci', 'degree',
'degree_Fahrenheit', 'deka', 'division', 'dyn', 'dyne', 'e', 'eV', 'electron_mass', 'electron_volt', 'elementary_charge', 'epsi
lon_0', 'erg', 'exa', 'exbi', 'femto', 'fermi', 'find', 'fine_structure', 'fluid_ounce', 'fluid_ounce_US', 'fluid_ounce_imp',
'foot', 'g', 'gallon', 'gallon_US', 'gallon_imp', 'gas_constant', 'gibi', 'giga', 'golden', 'golden_ratio', 'grain', 'gram', 'g
ravitational_constant', 'h', 'hbar', 'hectare', 'hecto', 'horsepower', 'hour', 'hp', 'inch', 'k', 'kgf', 'kibi', 'kilo', 'kilog
ram_force', 'kmh', 'knot', 'lambda2nu', 'lb', 'lbf', 'light_year', 'liter', 'litre', 'long_ton', 'm_e', 'm_n', 'm_p', 'm_u', 'm
ach', 'mebi', 'mega', 'metric_ton', 'micro', 'micron', 'mil', 'mile', 'milli', 'minute', 'mmHg', 'mph', 'mu_0', 'nano', 'nautic
al_mile', 'neutron_mass', 'nu2lambda', 'ounce', 'oz', 'parsec', 'pebi', 'peta', 'physical_constants', 'pi', 'pico', 'point', 'p
ound', 'pound_force', 'precision', 'print_function', 'proton_mass', 'psi', 'pt', 'short_ton', 'sigma', 'slinch', 'slug', 'speed
_of_light', 'speed_of_sound', 'stone', 'survey_foot', 'survey_mile', 'tebi', 'tera', 'test', 'ton_TNT', 'torr', 'troy_ounce',
'troy_pound', 'u', 'unit', 'value', 'week', 'yard', 'year', 'yobi', 'yotta', 'zebi', 'zepto', 'zero_Celsius', 'zetta']
```

Unit Categories The units are placed under these categories:

```
Metric
Binary
Mass
Angle
Time
Length
Pressure
Volume
Speed
Temperature
Energy
Power
Force
```

Print the constant value of PI:

In [ ]:

```
1 from scipy import constants
2
3 print(constants.pi)
```

```
3.141592653589793
```

scipy.cluster	Vector quantization / Kmeans
scipy.constants	Physical and mathematical constants
scipy.fftpack	Fourier transform
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	Data input and output
scipy.linalg	Linear algebra routines
scipy.ndimage	n-dimensional image package
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrices
scipy.spatial	Spatial data structures and algorithms
scipy.special	Any special mathematical functions
scipy.stats	Statistics

scipy is composed of task-specific sub-modules

❑ Optimizers in SciPy

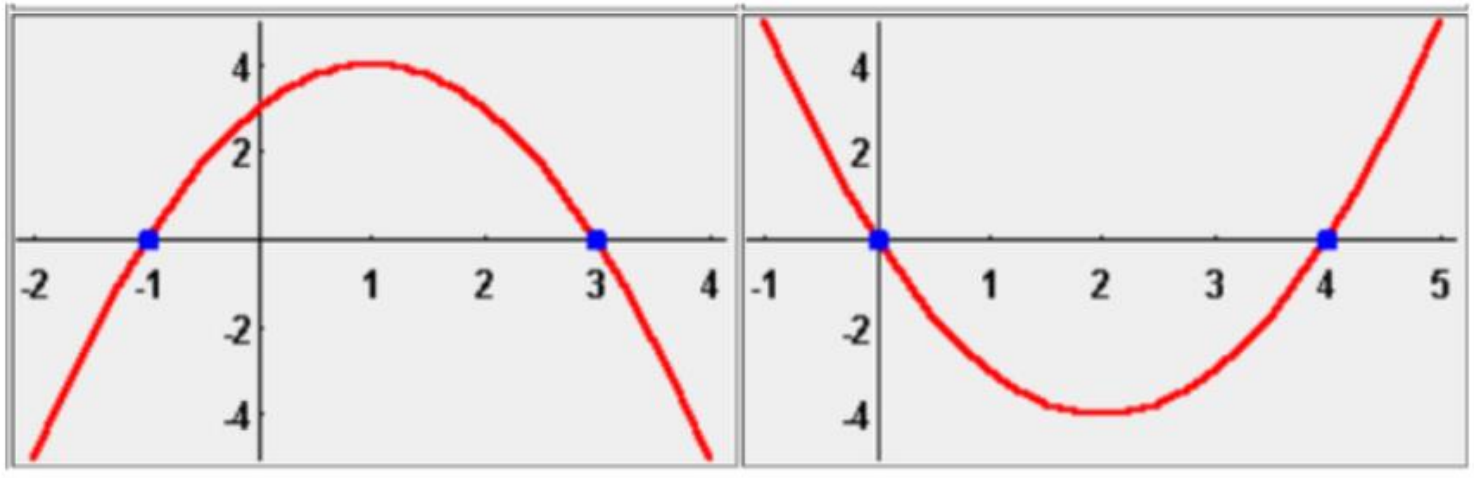
Optimizers are a set of procedures defined in SciPy that either find the minimum value of a function, or the root of an equation.

Optimizing Functions

Essentially, all of the algorithms in Machine Learning are nothing more than a complex equation that needs to be minimized with the help of given data.

1.Roots of an Equation

For an equation ax^2 + bx + c = 0, whichever value of x satisfies the equation is called a root. Value of X for which the equation is satisfied (that is the equation equals 0) is what the roots are. Those x for which f(x) = 0 is called a root.



NumPy is capable of finding roots for polynomials and linear equations, but it can not find roots for non linear equations, like this one: **x + cos(x)**

For that you can use SciPy's optimze.root function.

This function takes two required arguments:

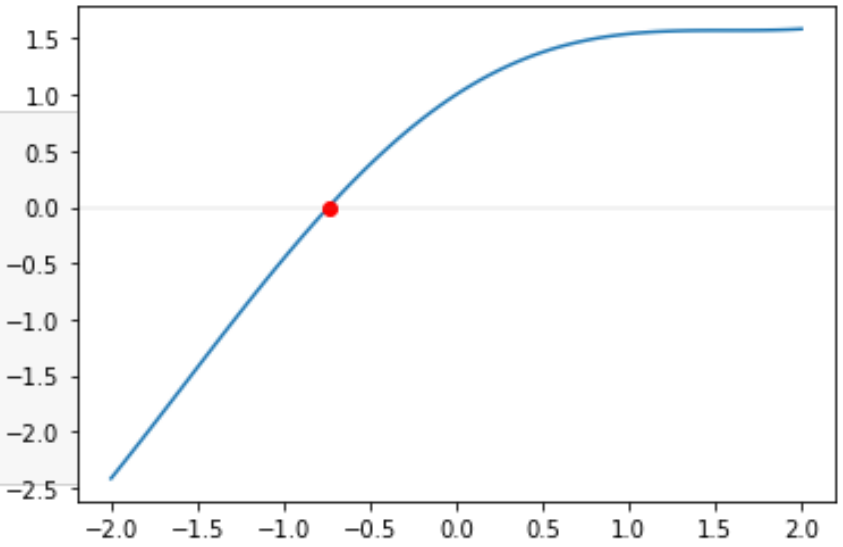
- fun - a function representing an equation.
- x0 - an initial guess for the root.

The function returns an object with information regarding the solution.

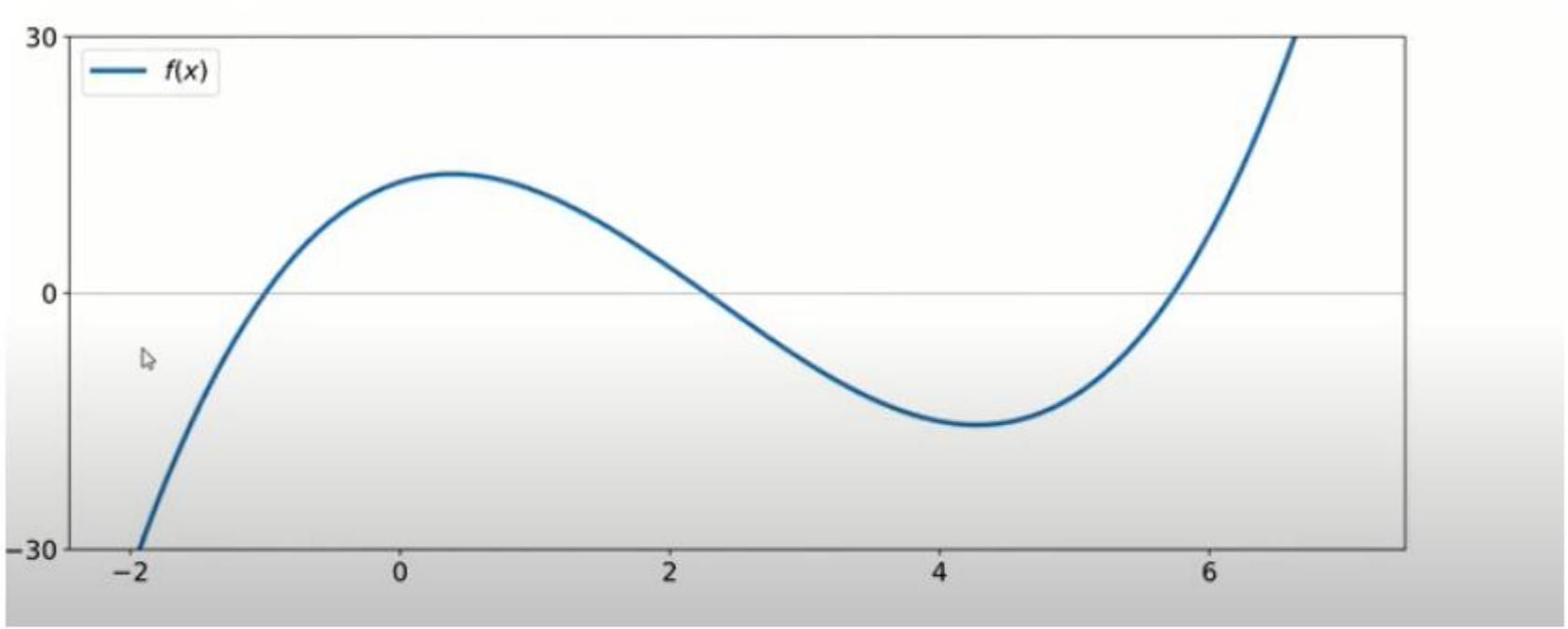
The actual solution is given under attribute x of the returned object:

```
1  from scipy.optimize import root
2  from math import cos
3
4  def eqn(x):
5    return x + cos(x)
6
7  myroot = root(eqn, 0)
8
9  print(myroot.x)
```

```
[-0.73908513]
```



Example2:



$$f(x) = x^3 - 7x^2 + 5x + 13$$

In [126]:
```
from scipy.optimize import root,minimize
import matplotlib.pyplot as plt
```

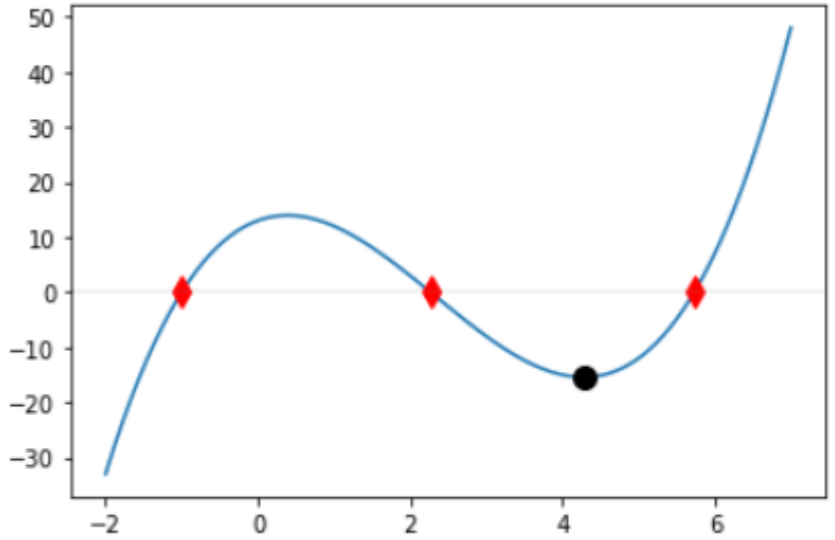
In [127]:
```
x0=[-2,2,6]
```

In [128]:
```
def equ(x):
    return (x**3) - 7* (x**2) + 5 *x + 13

```

In [153]:
```
optim_root= root(equ,x0)
```

In [154]:
```
optim_root.x
```

Out[154]:
```
array([-1.        ,  2.26794919,  5.73205081])
```



2. Minimizing a Function

A function, in this context, represents a curve, curves have high points and low points.

- High points are called maxima.
- Low points are called minima.

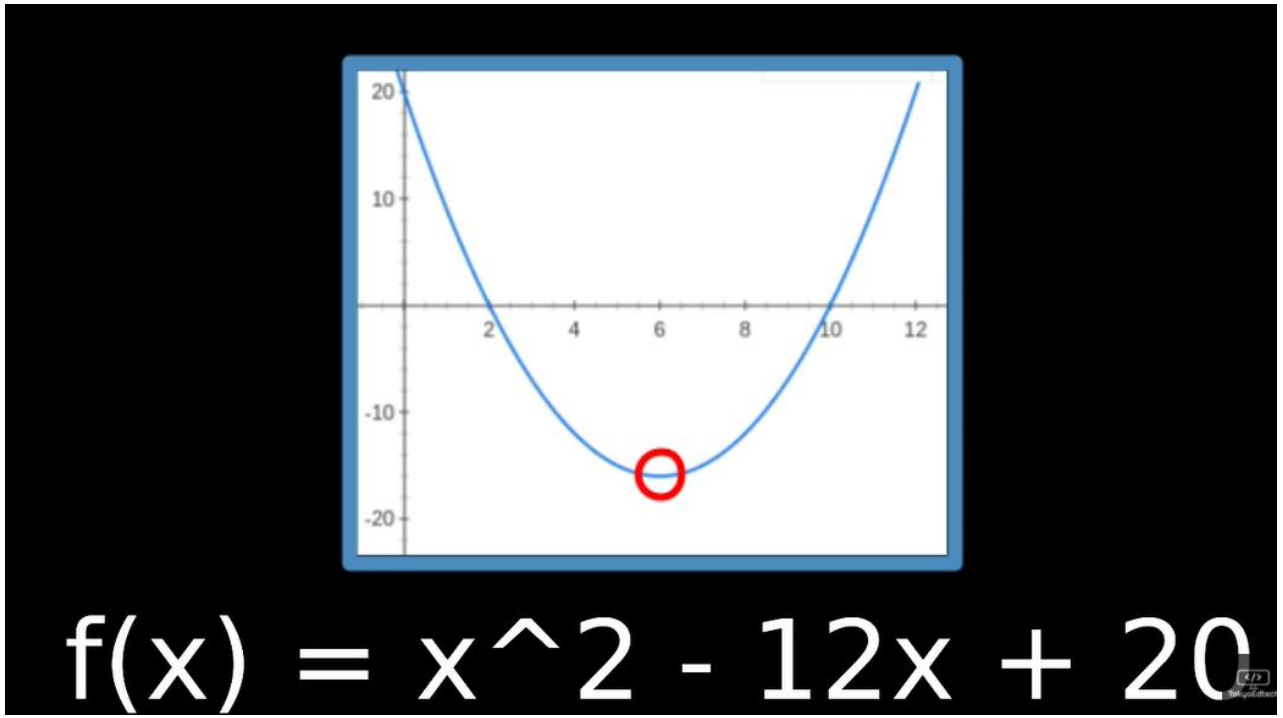
The highest point in the whole curve is called global maxima, whereas the rest of them are called local maxima.
The lowest point in whole curve is called global minima, whereas the rest of them are called local minima.

Finding Minima

We can use **scipy.optimize.minimize()** function to minimize the function.
The minimize() function takes the following arguments:

- fun - a function representing an equation.
- x0 - an initial guess for the root.
- method - name of the method to use.
  Legal values: 'CG' ,'BFGS' ,'Newton-CG' ,'L-BFGS-B', 'TNC' ,'COBYLA' ,'SLSQP'

Example




```
In [1]: import numpy as np
        from scipy.optimize import minimize

In [2]: def equ(x):
            return x**2-12*x+20

In [3]: optmizer=minimize(equ,0,method='BFGS')

In [4]: optmizer.x
Out[4]: array([5.99999969])

        ##########################################################################

In [5]: optmizer2=minimize(equ,0)

In [6]: optmizer2.x
Out[6]: array([5.99999969])
```

❑ SciPy - Linalg

It has very fast linear algebra capabilities. All of these linear algebra routines expect an object that can be converted into a two-dimensional array. The output of these routines is also a two-dimensional array.

1.Solve linear algebra problems

The scipy.linalg.solve feature solves the linear equation a * x + b * y = Z, for the unknown x, y values.

As an example, assume that it is desired to solve the following simultaneous equations.

- x + 3y + 5z = 10
- 2x + 5y + z = 8
- 2x + 3y + 8z = 3

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy arrays
a = np.array([[1, 3, 5], [2, 5, 1], [2, 3, 8]])
b = np.array([10, 8, 3])

#Passing the values to the solve function
x = linalg.solve(a, b)

#printing the result array
print(x)
```

```
[-9.28  5.16  0.76]
```

```
b_result=a@x
b_result
```

```
array([10.,  8.,  3.])
```

2. Finding a Determinant

The determinant of a square matrix A is often denoted as |A| and is a quantity often used in linear algebra. In SciPy, this is computed using the **det()** function. It takes a matrix as input and returns a scalar value.
Let us consider the following example.

In [ ]:

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy array
A = np.array([[1,2],[3,4]])

#Passing the values to the det function
x = linalg.det(A)

#printing the result
print(x)
```

```
-2.0
```

3. Eigenvalues and Eigenvectors

The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. We can find the Eigen values (λ) and the corresponding Eigen vectors (v) of a square matrix (A) by considering the following relation −

$$Av = \lambda v$$

scipy.linalg.eig computes the eigenvalues from an ordinary or generalized eigenvalue problem. This function returns the Eigen values and the Eigen vectors.

Let us consider the following example.

In [ ]:

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy array
A = np.array([[1,2],[3,4]])

#Passing the values to the eig function
l, v = linalg.eig(A)

#printing the result for eigen values
print(l)

#printing the result for eigen vectors
print(v)
```

```
[-0.37228132+0.j  5.37228132+0.j]
[[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

4.Singular Value Decomposition

A Singular Value Decomposition (SVD) can be thought of as an extension of the eigenvalue problem to matrices that are not square.

The **scipy.linalg.svd** the matrix ‘a’ into two unitary matrices ‘U’ and ‘VT’ and a 1-D array ‘D’ of singular values (real, non-negative) such that a == U*D*VT, where ‘D’ is a suitably shaped matrix of zeros with the main diagonal ‘D’.

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

```python
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy array
a = np.random.randn(3, 2) + 1.j*np.random.randn(3, 2)
print(f"a = {a}\n")

#Passing the values to the eig function
U, D, VT = linalg.svd(a)

# printing the result
print( f"u= {U}\n\nVT={VT}\n\nD={D}")
```

```
a = [[-1.31447089+0.33949471j -0.98291491+0.57892273j]
 [ 1.00348029+2.35093192j  0.95483016+0.93578759j]
 [-0.62010432-0.82101414j -2.39575475+1.50319078j]]

u= [[-0.41258363+0.00828716j -0.13286839+0.22711379j -0.7571609 +0.43264161j]
 [ 0.17024253+0.61954045j  0.38536872+0.46985164j -0.24108939-0.39974166j]
 [-0.60247203-0.23220817j  0.737433  -0.13299728j  0.09005949-0.11619126j]]

VT=[[ 0.66270619+0.j          0.54411103-0.51455193j]
 [ 0.7488795 +0.j         -0.48150036+0.45534262j]]

D=[4.12960796 1.8627322 ]
```

Check out other scipy Linear algebra functions:

https://docs.scipy.org/doc/scipy/reference/linalg.html#module-scipy.linalg

❑ SciPy Interpolation

Interpolation is a method for generating points between given points.

For example: for points 1 and 2, we may interpolate and find points 1.33 and 1.66.

Interpolation has many usage, in Machine Learning we often deal with missing data in a dataset, interpolation is often used to substitute those values.

This method of filling values is called imputation.

Apart from imputation, interpolation is often used where we need to smooth the discrete points in a dataset.

1. interp1d

Interpolate a 1-D function.

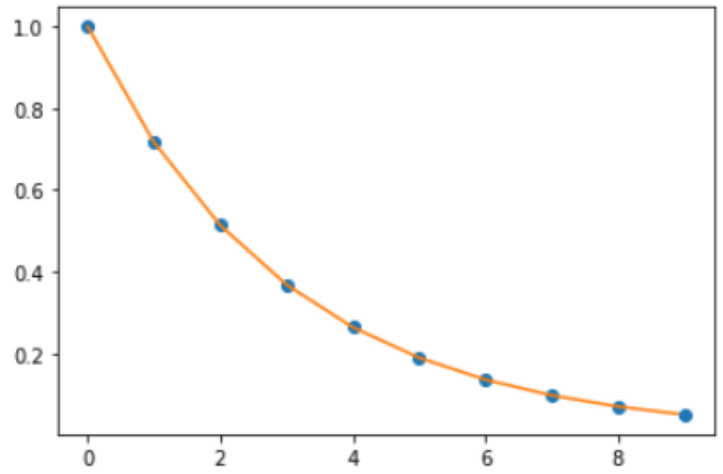
x and y are arrays of values used to approximate some function f: y = f(x). This class returns a function whose call method uses interpolation to find the value of new points.

Example1:

```
1  import matplotlib.pyplot as plt
2  from scipy.interpolate import interp1d
3  import numpy as np
4  x = np.arange(0, 10)
5  y = np.exp(-x/3.0)
6  print(x)
7  f =interp1d(x, y)
8
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
1  xnew = np.arange(0, 9,0.1)
2  ynew = f(xnew)   # use interpolation function returned by `interp1d`
3  ##print(ynew)
4  plt.plot(x, y, 'o', xnew, ynew)
5  plt.show()
```

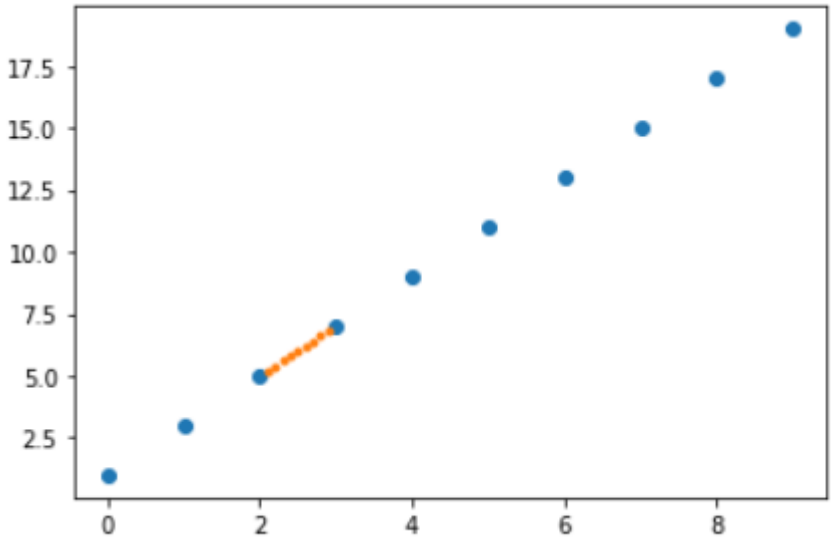


Print(ynew)

```
[1.         0.97165313 0.94330626 0.91495939 0.88661252 0.85826566
 0.82991879 0.80157192 0.77322505 0.74487818 0.71653131 0.69621989
 0.67590847 0.65559705 0.63528563 0.61497421 0.5946628  0.57435138
 0.55403996 0.53372854 0.51341712 0.49886335 0.48430958 0.46975582
 0.45520205 0.44064828 0.42609451 0.41154074 0.39698698 0.38243321
 0.36787944 0.35745121 0.34702298 0.33659475 0.32616652 0.31573829
 0.30531006 0.29488183 0.2844536  0.27402537 0.26359714 0.25612498
 0.24865283 0.24118068 0.23370852 0.22623637 0.21876422 0.21129206
 0.20381991 0.19634776 0.1888756  0.18352157 0.17816754 0.17281351
 0.16745947 0.16210544 0.15675141 0.15139738 0.14604335 0.14068932
 0.13533528 0.13149895 0.12766262 0.12382629 0.11998996 0.11615363
 0.11231729 0.10848096 0.10464463 0.1008083  0.09697197 0.09422312
 0.09147426 0.08872541 0.08597656 0.08322771 0.08047886 0.07773001
 0.07498115 0.0722323  0.06948345 0.06751381 0.06554417 0.06357454
 0.0616049  0.05963526 0.05766562 0.05569598 0.05372634 0.05175671]
```

Example2:

```
1  from scipy.interpolate import interp1d
2  import numpy as np
3
4  xs = np.arange(10)
5  ys = 2*xs + 1
6
7  interp_func = interp1d(xs, ys)
8  new_x=np.arange(2.1, 3, 0.1)
9  y_new = interp_func(new_x)
10
11 print(f"xs={xs}\nys={ys}\ny_new={y_new}")
12 plt.plot(xs,ys,"o",new_x,y_new,".")
```




```
xs=[0 1 2 3 4 5 6 7 8 9]
ys=[ 1  3  5  7  9 11 13 15 17 19]
y_new=[5.2 5.4 5.6 5.8 6.  6.2 6.4 6.6 6.8]
```

$$y = y_1 + (x - x_1)\frac{y_2 - y_1}{x_2 - x_1}$$

$$y(x = 2.1) = 1 + \frac{3-1}{1-0} * (2.1 - 0) \quad \textbf{= 5.2}$$

2.Spline Interpolation

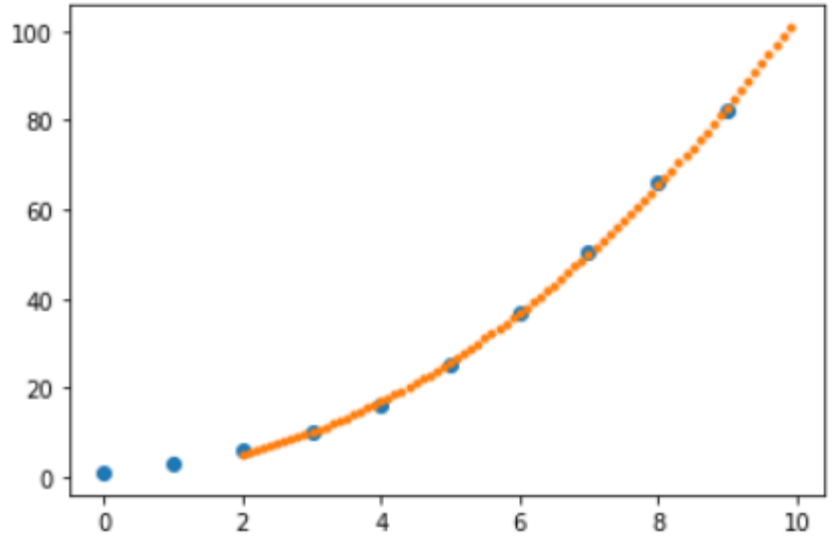
1-D smoothing spline fit to a given set of data points.

In 1D interpolation the points are fitted for a single curve whereas in Spline interpolation the points are fitted against a piecewise function defined with polynomials called splines.

The **UnivariateSpline()** function takes xs and ys and produce a callable funciton that can be called with new xs.

Find univariate spline interpolation for 2.1, 2.2... 2.9 for the following non-linear points:

```
:
1  from scipy.interpolate import UnivariateSpline
2  import numpy as np
3
4  xs = np.arange(10)
5  ys = xs**2 + np.sin(xs) + 1
6
7  interp_func = UnivariateSpline(xs, ys)
8  xnew=np.arange(2, 10, 0.1)
9  newarr = interp_func(xnew)
10 plt.plot(xs, ys, 'o', xnew, newarr, '.')
11 plt.show()
```



3.Interpolation with Radial Basis Function

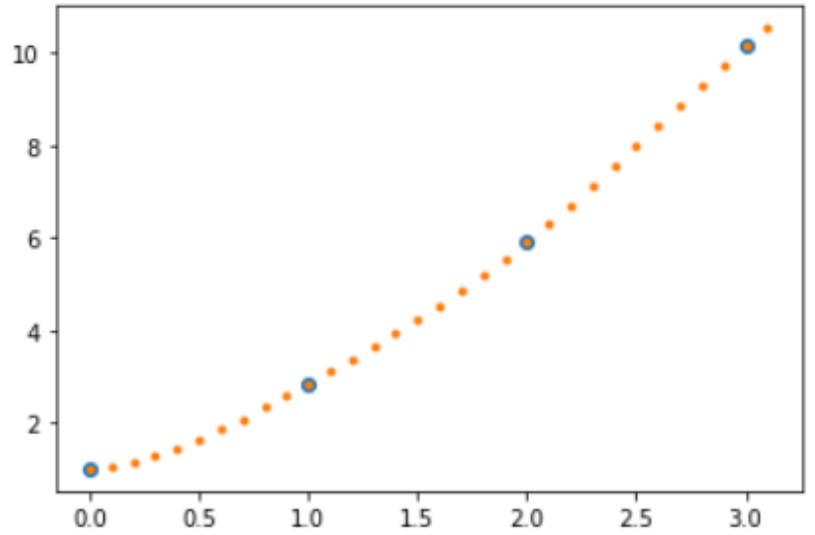
A class for radial basis function interpolation of functions from N-D scattered data to an M-D domain.

Radial basis function is a function that is defined corresponding to a fixed reference point.

The Rbf() function also takes xs and ys as arguments and produces a callable function that can be called with new xs.

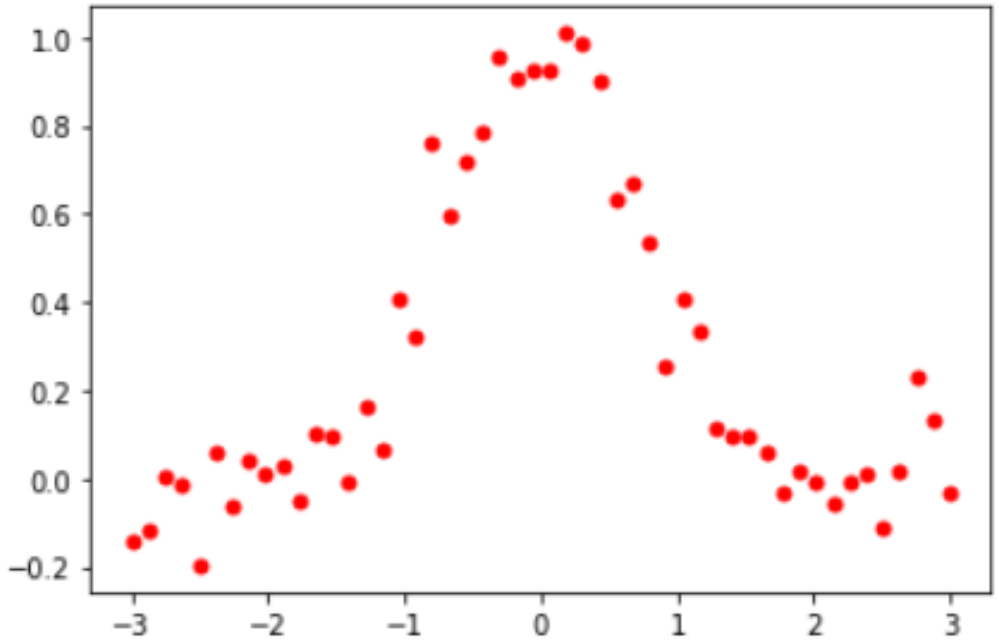
In [18]:

```
from scipy.interpolate import Rbf
import numpy as np

xs = np.arange(pi)
ys = xs**2 + np.sin(xs) + 1

interp_func = Rbf(xs, ys)
ynew = interp_func(xnew)

plt.plot(xs, ys, 'o', xnew, ynew, '.')
plt.show()

newarr = interp_func(np.arange(0, pi, 0.1))

print(newarr)
```




```
[ 1.          1.0498861   1.14195986  1.27270156  1.43717516  1.62981354
  1.84506698  2.07783088  2.32369637  2.57910646  2.84147098  3.10924007
  3.38190376  3.65990061  3.94445953  4.23741926  4.54104772  4.85784101
  5.19025493  5.54033755  5.90929743  6.29712807  6.7024397   7.12256928
  7.55390042  7.99224688  8.43318845  8.87233933  9.30559594  9.72941546
 10.14112001 10.53914236]
```

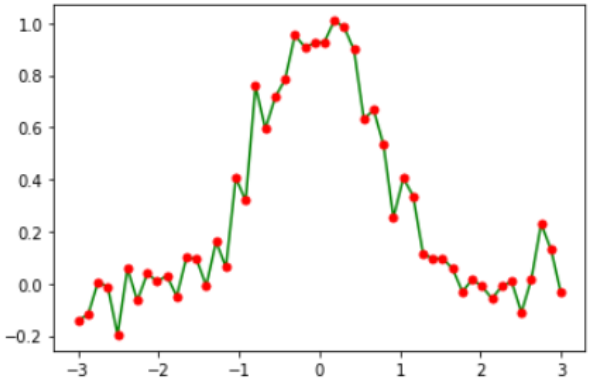
```python
import matplotlib.pyplot as plt
from scipy.interpolate import UnivariateSpline
from scipy.interpolate import interp1d
from scipy.interpolate import Rbf
rng = np.random.default_rng()
x = np.linspace(-3, 3, 50)
y = np.exp(-x**2) + 0.1 * rng.standard_normal(50)
plt.plot(x, y, 'ro', ms=5)
plt.show()
```



General example for spline, iterp1d & Radial interpolation

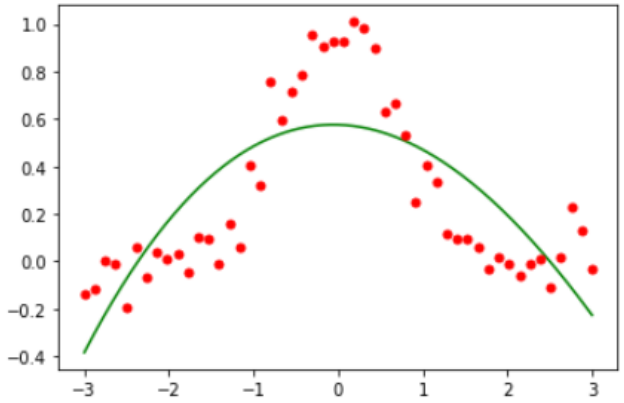
1- interp1d

```
]:
onedinter=interp1d(x,y)
xs = np.linspace(-3, 3, 1000)
plt.plot(xs, onedinter(xs), 'g', x, y, 'ro', ms=5 )
plt.show()
```



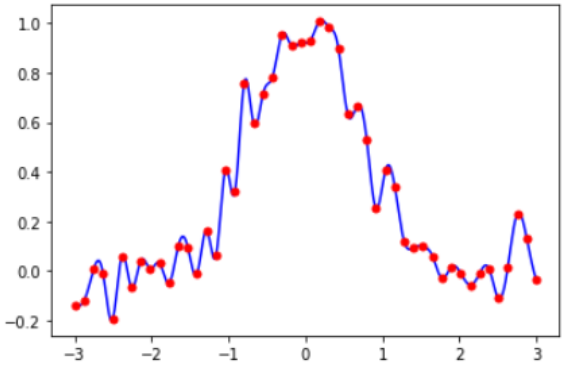
2-UnivariateSpline

```
spl = UnivariateSpline(x, y)
xs = np.linspace(-3, 3, 1000)
plt.plot(xs, spl(xs), 'g', x, y, 'ro', ms=5)
plt.show()
```



3-Rbf

```
radial_fun=Rbf(x,y)
ynew=radial_fun(xs)
plt.plot(xs, ynew, 'b', x, y, 'ro', ms=5)
plt.show()
```



```
spl.set_smoothing_factor(0.5)
plt.plot(xs, spl(xs), 'b', x, y, 'ro', ms=5)
plt.show()
```