Data Visualisation with Matplotlib

Matplotlib

- We can visualise a dataset in Jupyter Notebook using Pandas and Matplotlib libraries.
- `Matplotlib` is a python specialising in the development of two dimensional chart (including 3D charts)
- Most used tool in the graphical representation of data.
- The `pyplot` package provides classic Python interface for programming the matplotlib library.
- Pyplot requires the import of `Numpy` package separately.
- At the beginning, you need to import pyplot and rename is as plt:

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

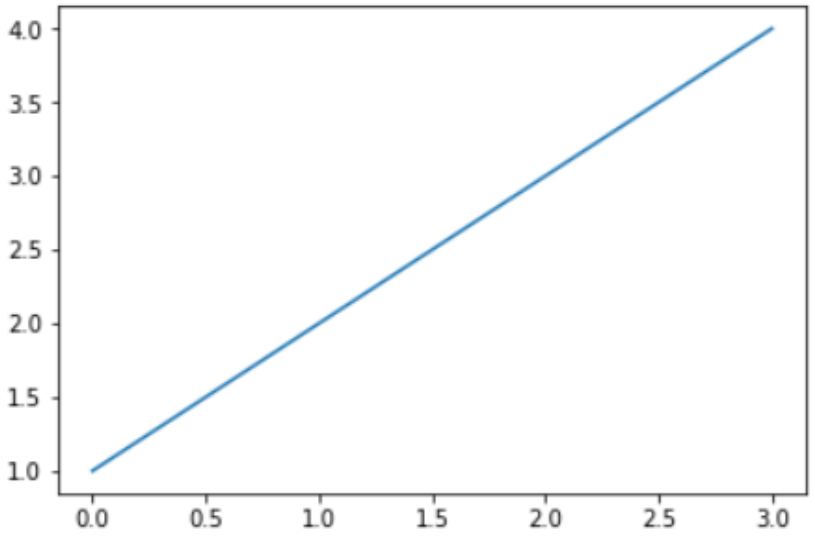
A simple Interactive Chart

A simple plot: A blue line connection the points

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

```
plt.plot([1,2,3,4])
```

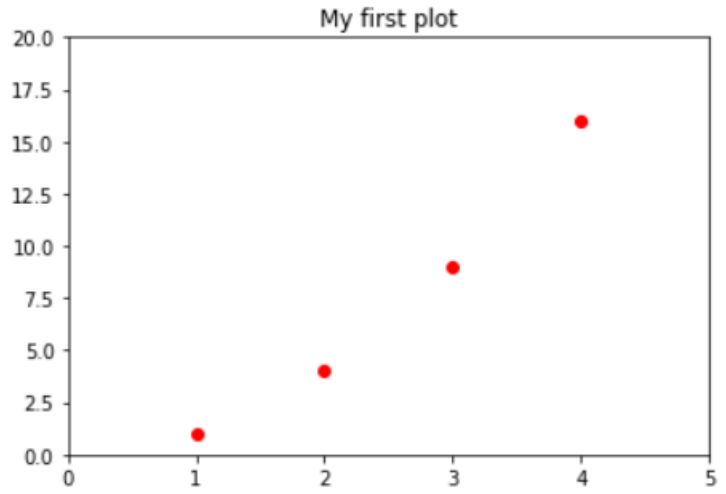
```
[<matplotlib.lines.Line2D at 0x22671759470>]
```



Set the properties of the plot. Each pair of values(x,y) is represented by a red dot.

```
plt.axis([0,5,0,20])
plt.title('My first plot')
plt.plot([1,2,3,4],[1,4,9,16], 'ro')
```

```
[<matplotlib.lines.Line2D at 0x22671f83048>]
```



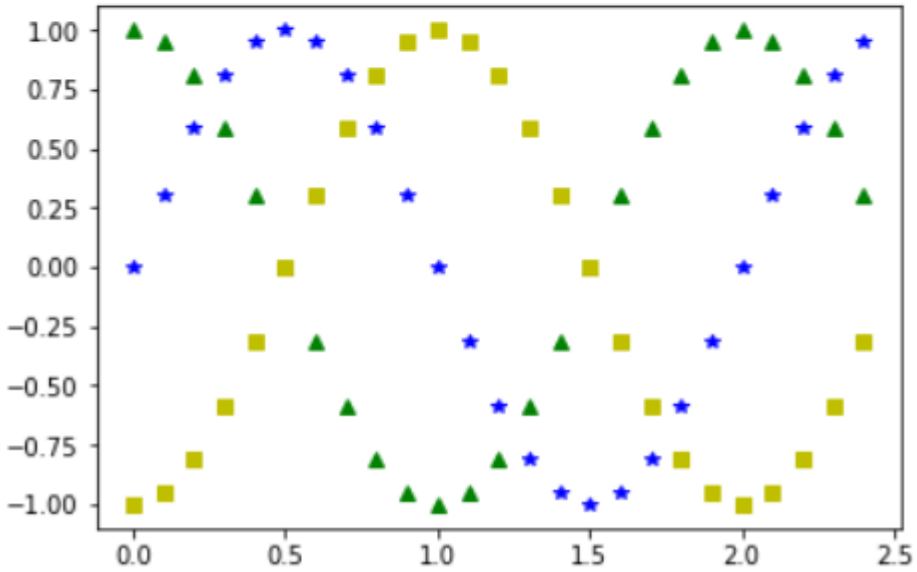
- It is possible to plot three different trends in the same plots.

- The example below shows three sinusoidal trends.

```
import math
import numpy as np
```

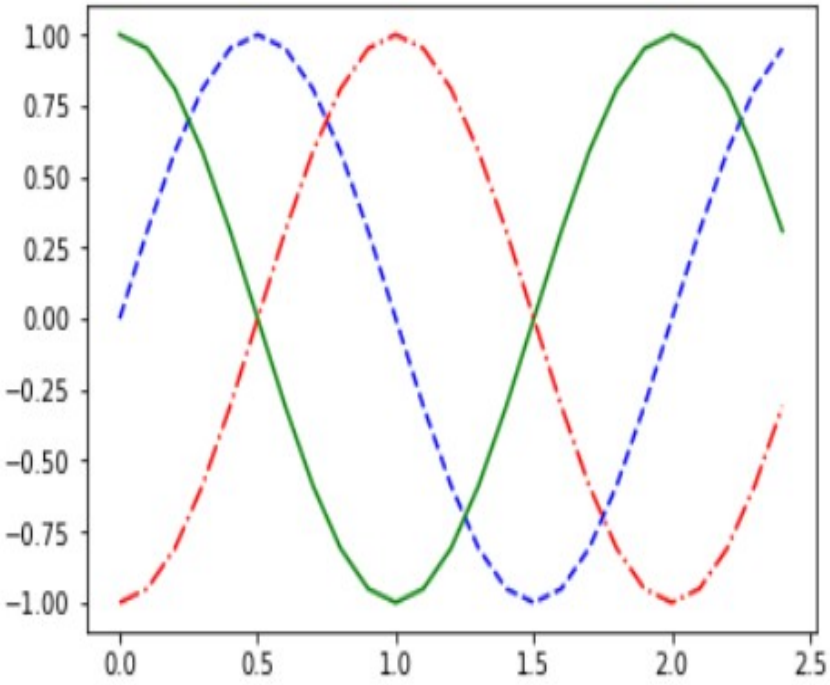
```
t = np.arange(0,2.5,0.1)
y1 = np.sin(math.pi*t)
y2 = np.sin(math.pi*t+math.pi/2)
y3 = np.sin(math.pi*t-math.pi/2)
plt.plot(t,y1,'b*',t,y2,'g^',t,y3,'ys')
```

```
[<matplotlib.lines.Line2D at 0x226720852b0>,
 <matplotlib.lines.Line2D at 0x226720854e0>,
 <matplotlib.lines.Line2D at 0x226720857f0>]
```




```
plt.plot(t,y1,'b--',t,y2,'g',t,y3,'r-.')
```

```
[<matplotlib.lines.Line2D at 0x226720eba90>,
 <matplotlib.lines.Line2D at 0x226720ebbe0>,
 <matplotlib.lines.Line2D at 0x226720ebf60>]
```



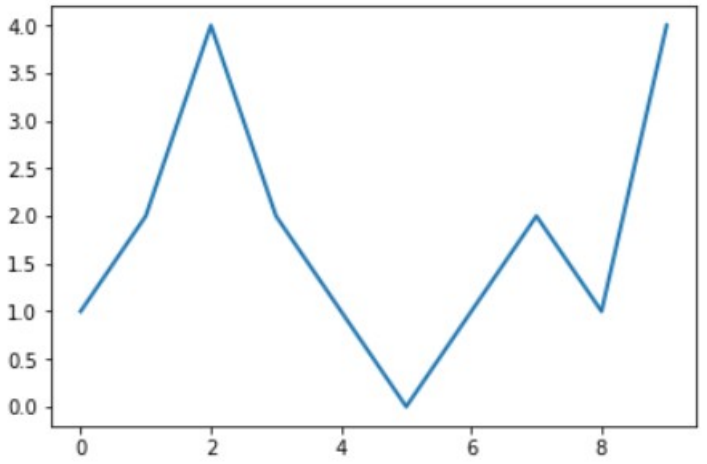
- The object that makes up the chart have many attributes that characterise them. The attributes are all default values but can be set through the use of `keyword args` known as `kwargs`.

- These keywords are passed as arguments to functions.

- For example the thickness of a line can be changed if we set the `linewidth` keyword.

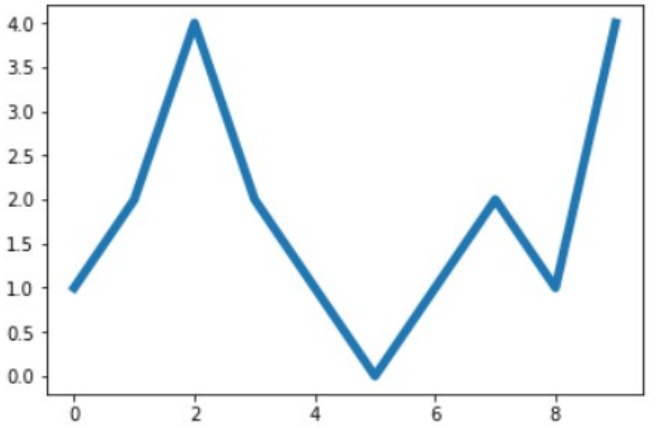
```
plt.plot([1,2,4,2,1,0,1,2,1,4],linewidth=2.0)
```

```
[<matplotlib.lines.Line2D at 0x22672150f28>]
```




```
plt.plot([1,2,4,2,1,0,1,2,1,4],linewidth=5.0)
```

```
[<matplotlib.lines.Line2D at 0x22675f4ec18>]
```



Working with Multiple Figures and Axes

- Various subplots can be represented in a single figure.
- The `subplot()` function subdivides the figure in different drawing areas. It also used to focus the commands on a specific subplot.

- The argument passed to the `subplot()` function sets the mode of subdivision and determines which is the current subplot. The current subplot will be the only figure affected by the commands.

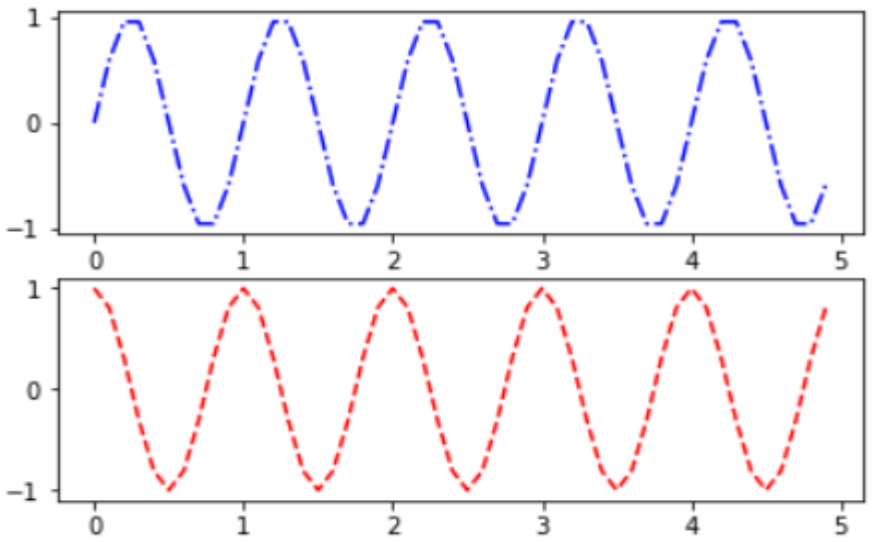
- The `subplot()` function is composed of three integers.
  - the first number determines how many parts the figure is split into vertically
  - The second determines how many parts the figure is split into horizontally.
  - The third number selects which is the current subplot on which we can direct commands.

Working with Multiple Figures and Axes

- Examples: two sinusoidal trends (sine and cosine)
- In the first image below, the canvas is divided in two horizontal subplots with number `211` and `212` as arguments to the `subplot()` function.
- In the second one, the canvas is divided in two vertical subplots (121 and 122)

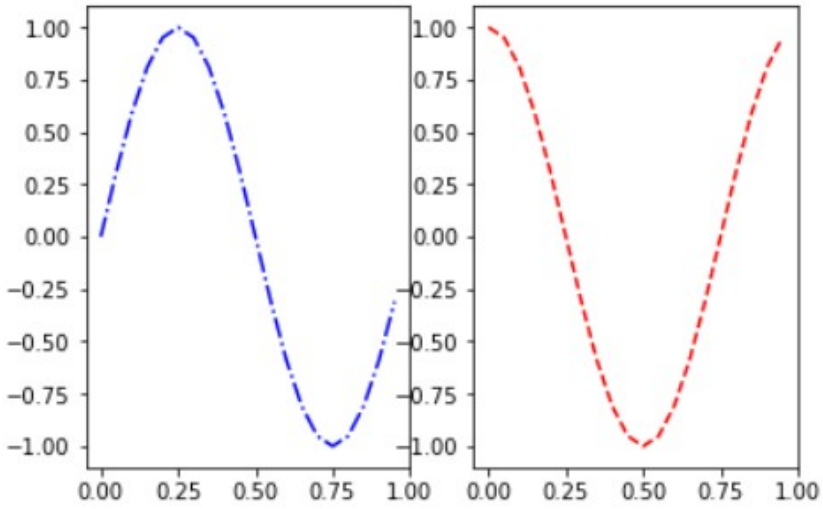
```
t = np.arange(0,5,0.1)
y1 = np.sin(2*np.pi*t)
y2 = np.cos(2*np.pi*t)
plt.subplot(211)
plt.plot(t,y1,'b-.')
plt.subplot(212)
plt.plot(t,y2,'r--')
```

```
[<matplotlib.lines.Line2D at 0x22677409e48>]
```




```
t = np.arange(0.,1.,0.05)
y1 = np.sin(2*np.pi*t)
y2 = np.cos(2*np.pi*t)
plt.subplot(121)
plt.plot(t,y1,'b-.')
plt.subplot(122)
plt.plot(t,y2,'r--')
```

```
[<matplotlib.lines.Line2D at 0x22675cb5cf8>]
```



Adding a Title and Text

The `text()` function and the `title()` function

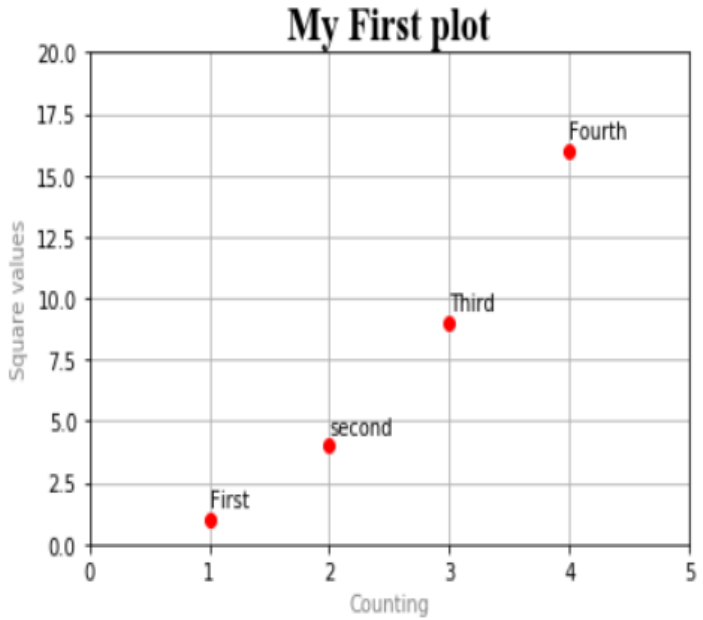
```
plt.axis([0,5,0,20])
plt.title('My First plot',fontsize = 20,fontname = 'Times New Roman')
plt.xlabel('Counting', color='gray')
plt.ylabel('Square values', color='gray')
plt.text(1,1.5,'First')
plt.text(2,4.5,'second')
plt.text(3,9.5,'Third')
plt.text(4,16.5,'Fourth')
plt.plot([1,2,3,4],[1,4,9,16], 'ro')
```

```
[<matplotlib.lines.Line2D at 0x2267221d4a8>]
```

Adding a Grid and a Legend

The `grid()` function and `legend()` function

```
plt.axis([0,5,0,20])
plt.title('My First plot',fontsize = 20,fontname = 'Times New Roman')
plt.xlabel('Counting', color='gray')
plt.ylabel('Square values', color='gray')
plt.text(1,1.5,'First')
plt.text(2,4.5,'second')
plt.text(3,9.5,'Third')
plt.text(4,16.5,'Fourth')
plt.plot([1,2,3,4],[1,4,9,16], 'ro')
plt.grid(True)
```




```
plt.axis([0,5,0,20])
plt.title('My First plot',fontsize = 20,fontname = 'Times New Roman')
plt.xlabel('Counting', color='gray')
plt.ylabel('Square values', color='gray')
plt.text(1,1.5,'First')
plt.text(2,4.5,'second')
plt.text(3,9.5,'Third')
plt.text(4,16.5,'Fourth')
plt.plot([1,2,3,4],[1,4,9,16], 'ro')
plt.grid(True)
plt.legend(['First Series'])
```

```
<matplotlib.legend.Legend at 0x226701a8f98>
```

Adding a Legend

- A legend is added in the upper-right corner by default.

- You can use the `loc` keyword to change this behaviour. This can be achieved by assigning numbers from 0 to 10 to the `loc` kwarg. Each number characterises one of the corner of the chart and the default value is 1, the upper-right corner.

Location Code	Location String
0	best
1	upper right
2	upper left
3	lower left
4	lower right
5	right
6	center left
7	center right
8	lower center
9	upper center
10	center

In this example, the *loc* kwargs was set to 2, which is upper-left

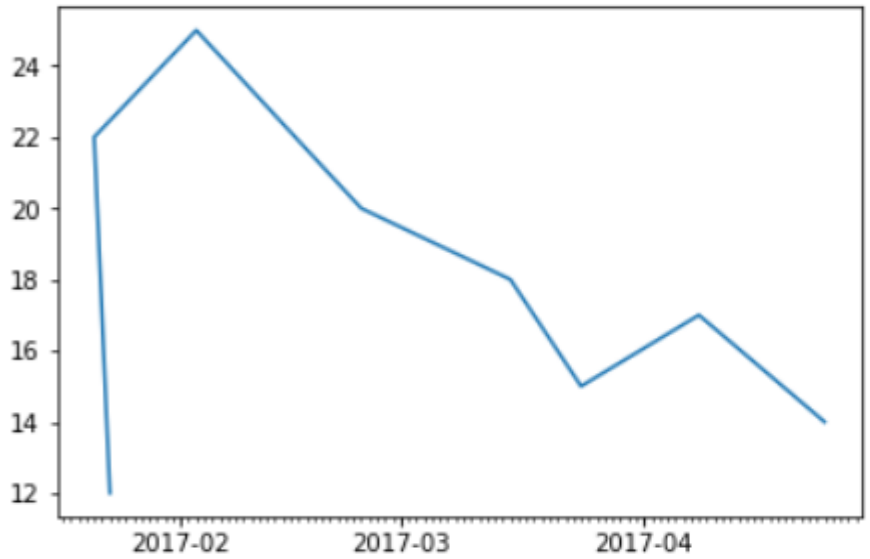
Saving charts as Image

```
plt.axis([0,5,0,20])
plt.title('My First plot',fontsize = 20,fontname = 'Times New Roman')
plt.xlabel('Counting', color='gray')
plt.ylabel('Square values', color='gray')
plt.text(1,1.5,'First')
plt.text(2,4.5,'second')
plt.text(3,9.5,'Third')
plt.text(4,16.5,'Fourth')
plt.plot([1,2,3,4],[1,4,9,16], 'ro')
plt.grid(True)
plt.legend(['First Series'])
plt.savefig('my_chart.png')
```

```
<matplotlib.legend.Legend at 0x226701a8f98>
```

Handling Date Values

```
import datetime
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
months = mdates.MonthLocator()
days = mdates.DayLocator()
timeFmt = mdates.DateFormatter('%Y-%m')
events = [datetime.date(2017,1,23),datetime.
          date(2017,1,21),datetime.date(2017,2,3), datetime.
          date(2017,2,24),datetime.date(2017,3,15),datetime.
          date(2017,3,24),datetime.date(2017,4,8),datetime.date(2017,4,24)]
readings = [12,22,25,20,18,15,17,14]
fig, ax = plt.subplots()
plt.plot(events,readings)
ax.xaxis.set_major_locator(months)
ax.xaxis.set_major_formatter(timeFmt)
ax.xaxis.set_minor_locator(days)
```

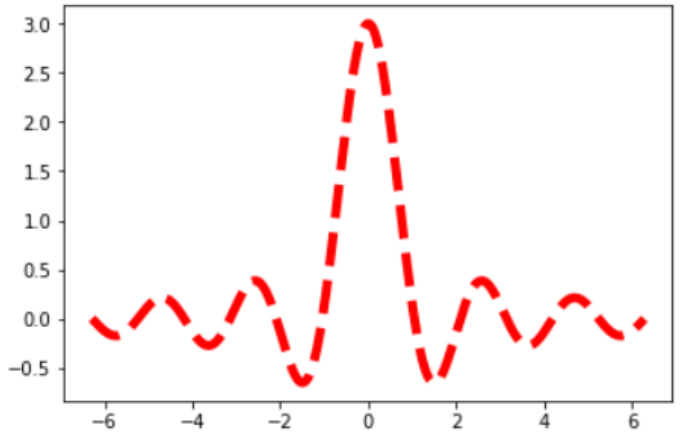


Line Charts

- The simplest chart
- Linear chart is a sequence of data points connected by a line.
- Each point consists of a pair values (x, y)
- You can use the color and linestyle kwargs to define the stroke.
- The table below presents the different colour codes

```
x = np.arange(-2*np.pi,2*np.pi,0.01)
y = np.sin(3*x)/x
plt.plot(x,y, color='r',linestyle='--', linewidth=5)
```

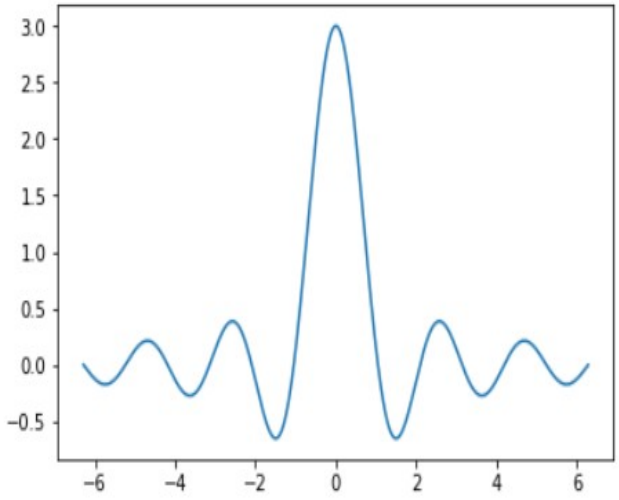
```
[<matplotlib.lines.Line2D at 0x226779d1668>]
```



Code	Colour
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

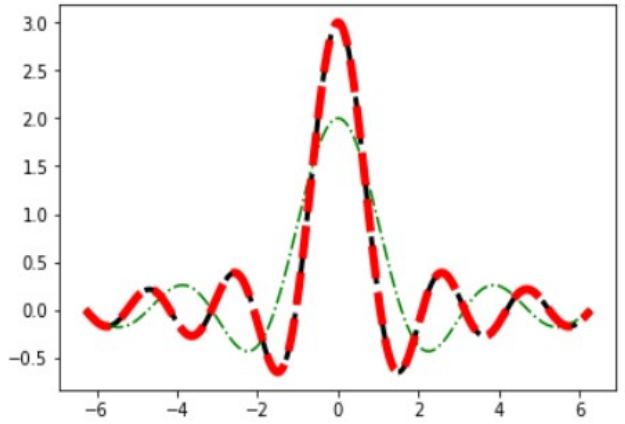
```
x = np.arange(-2*np.pi,2*np.pi,0.01)
y = np.sin(3*x)/x
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x226724ca588>]
```




```
x = np.arange(-2*np.pi,2*np.pi,0.01)
y = np.sin(3*x)/x
y2 =np.sin(2*x)/x
y3 = np.sin(3*x)/x
plt.plot(x,y,'k--', linewidth= 3)
plt.plot(x,y2,'g-.')
plt.plot(x,y3, color='r',linestyle='--', linewidth=5)
```

```
[<matplotlib.lines.Line2D at 0x226776bf630>]
```

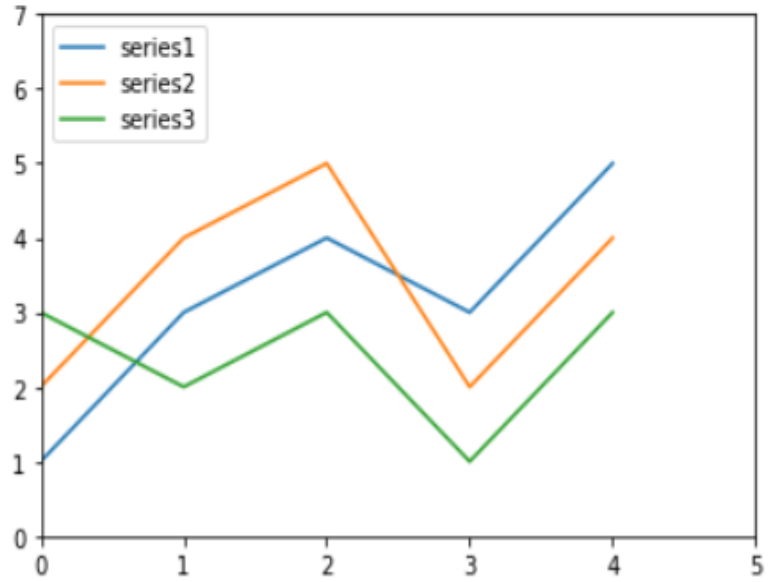


Line charts with Dataframe

- The visualisation of the data in a dataframe as a linear chart is very easy.
- Pass the dataframe as an argument to the plot() function to obtain a multiseries linear chart

```
import numpy as np
data = {'series1':[1,3,4,3,5],
        'series2':[2,4,5,2,4],
        'series3':[3,2,3,1,3]}
df = pd.DataFrame(data)
x = np.arange(5)
plt.axis([0,5,0,7])
plt.plot(x,df)
plt.legend(data,loc=2)
```

```
<matplotlib.legend.Legend at 0x226723989e8>
```



Histograms

- A histogram consists of adjacent rectangles erected on the `x-axis`, split into discrete intervals called `bins`. X-axis is used to reference numerical values.
- The *`hist()`* function allows you to represent a histogram **.**

- **Practical example:** Let's generate a population of 100 random values from 0 to 100 using `random.randint()` function as seen below.

```
import matplotlib.pyplot as plt
import numpy as np
pop = np.random.randint(0,100,100)
pop
```

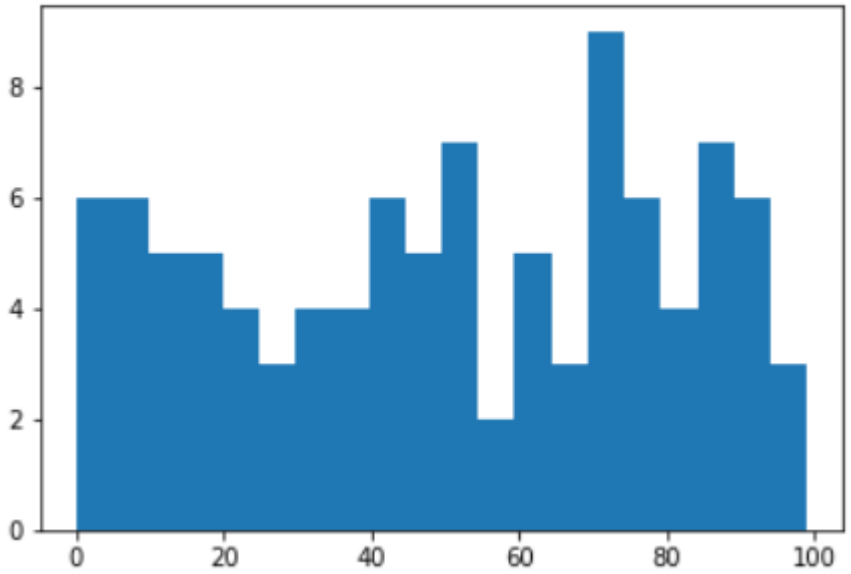
```
array([35, 66, 70, 30, 21, 61, 68, 43, 57, 86, 57, 69, 41, 72, 61, 44, 56,
       79, 50, 86, 46, 57, 35, 39, 30, 40, 96, 70, 28, 23, 54, 23, 11,  8,
       88, 33, 18, 15, 51, 73, 62, 19, 57, 28, 56, 73, 95, 18, 97, 37, 86,
       81, 96, 25, 88, 84, 22, 10, 31, 77, 80, 29,  1, 36, 71, 38, 35, 59,
       41, 71, 72, 68, 91, 94, 45, 27, 93, 88, 89, 43, 88, 90, 49, 71,  4,
        8, 58, 82, 54, 42, 66, 27, 85, 79, 17, 95, 84, 94, 13, 66])
```

Histograms

- Now we will create the histogram of these samples by passing as an argument the `hist()` function.
- We want to divide the occurrences in 20 bins (if not specified, the default value is 10 bins)
- To do that, we have to use the kwarg `bin`.

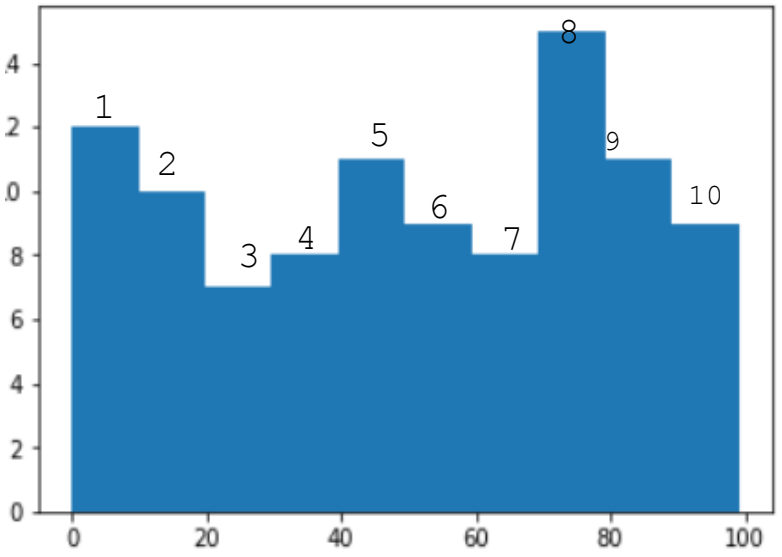
```
n,bins,patches = plt.hist(pop,bins=20)
```

20 bins



The histogram shows the number of occurrences in each bin

Default: 10 bins

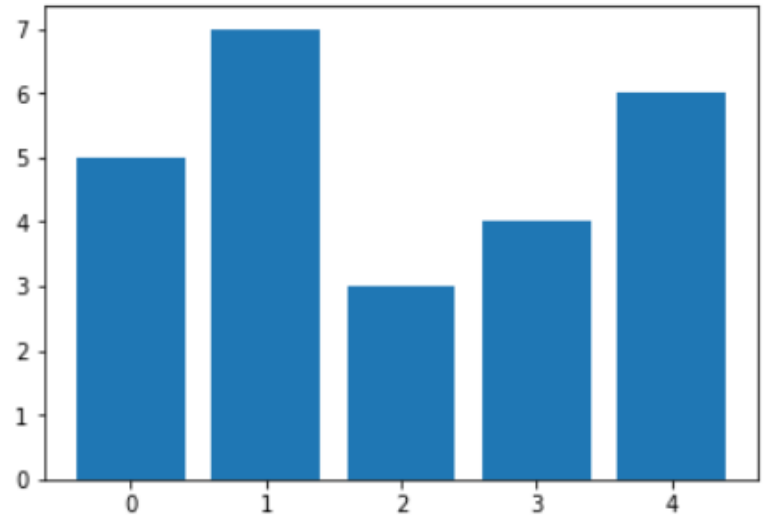


Bar Charts

- Another common type of chart, similar to histogram but the x-axis is used to reference categories.
- The `bar()` function is used to create a bar chart.

```
index = [0,1,2,3,4]
values = [5,7,3,4,6]
plt.bar(index,values)
```

```
<BarContainer object of 5 artists>
```



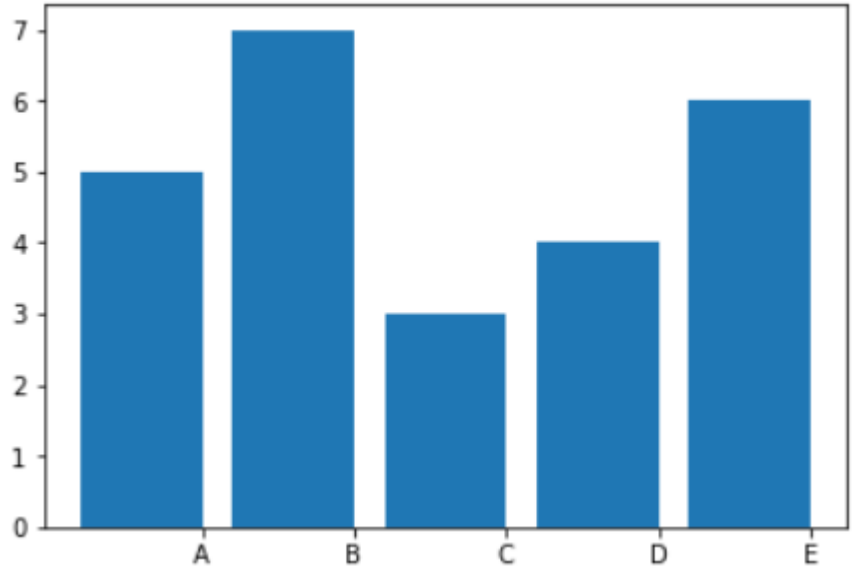
This bar chart shows that the indices are drawn on the x-axis. But because each bar corresponds to a category, it would be best if we can specify the categories through the tick label.

Bar Chart

- The tick label is defined by a list of strings passed to the `xticks()` function.

- For the location of the ticks, we have to pass a list containing the values corresponding to their positions on the x-axis as the first argument of the xticks() function.

```
index = np.arange(5)
values1 = [5,7,3,4,6]
plt.bar(index,values1)
plt.xticks(index+0.4,['A','B','C','D','E'])
```



Bar Chart – Using kwargs

- We can add a specific kwarg as an argument in the `bar()` function.

- In the next example, we add the standard deviation values of the bar through the `yerr` kwarg along with a list containing the standard deviations.

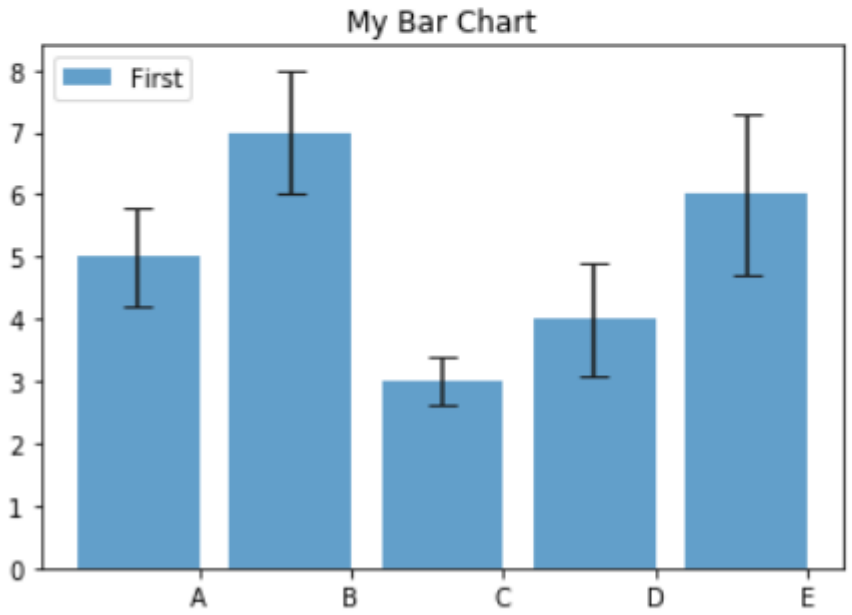
- The kwarg is usually combined with another kwarg called `error_kw`, which, in turn, can be used with other kwargs such:
  - `eColor`: specifies the colour of the error bars)
  - `capsize`: defines the width of the transverse lines that mark the ends of the error bars.

- The `alpha` kwarg indicates the degree of transparency of the coloured bar. Alpha is a value ranging from `0` to `1`. When the value is 0 the object is completely transparent to become gradually more significant as it increases. When the value reaches 1, the colour is fully represented.
- A legend is recommended. We use the kwarg `label` to identify the series we represent.

Bar Chart – Using kwargs

```
index = np.arange(5)
values1 = [5,7,3,4,6]
std1 = [0.8,1,0.4,0.9,1.3]
plt.title('My Bar Chart')
plt.bar(index,values1, yerr=std1,error_kw={'ecolor':'0.1',
                                           'capsize':6}, alpha=0.7,
       label='First')
plt.xticks(index+0.4,['A','B','C','D','E'])
plt.legend(loc=2)
```

```
<matplotlib.legend.Legend at 0x226745d5240>
```

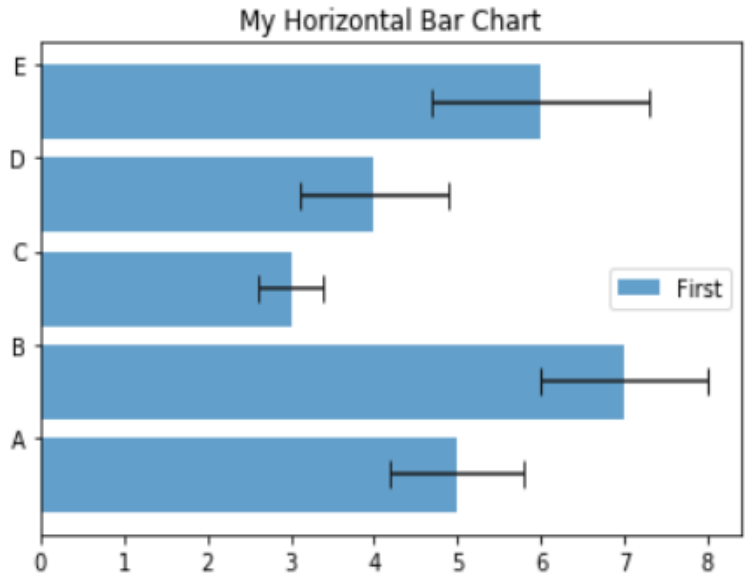


Horizontal Bar Charts

- Horizontal bar chart are implemented using the `barh()` function.
- The arguments and the kwargs valid for the bar() function remain the same for this function.
- The only change is that the roles of the axes are reversed. Now the categories are represented on the y-axis and the numreical values are on the x-axis.

```
index = np.arange(5)
values1 = [5,7,3,4,6]
std1 = [0.8,1,0.4,0.9,1.3]
plt.title('My Horizontal Bar Chart')
plt.barh(index,values1, xerr=std1,error_kw={'ecolor':'0.1',
                                            'capsize':6}, alpha=0.7,
       label='First')
plt.yticks(index+0.4,['A','B','C','D','E'])
plt.legend(loc=5)
```

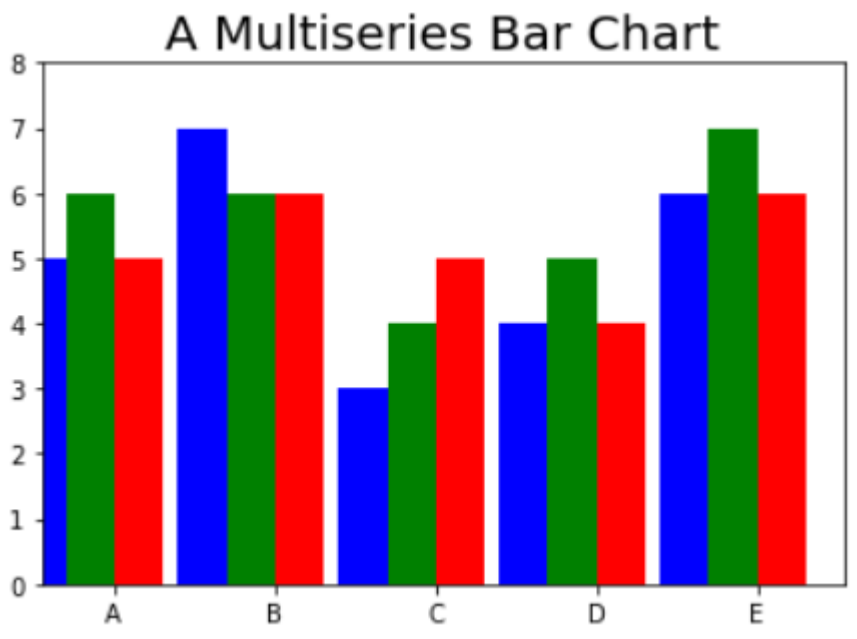
```
<matplotlib.legend.Legend at 0x226746b6240>
```



Multiseries Bar Charts

- As with line charts, bar charts can be used to display larger series of values.
- In a simple bar chart, each index corresponds to a bar and is assigned to the x-axis. These represents categories.
- In a multiseries bar chart, the bars must share the same category.
- To overcome that issue, the space occupied by an index is divided as many parts as the bars sharing that index.
- It is advisable to add space which will serve as the gap to separate a category with respect to the next.

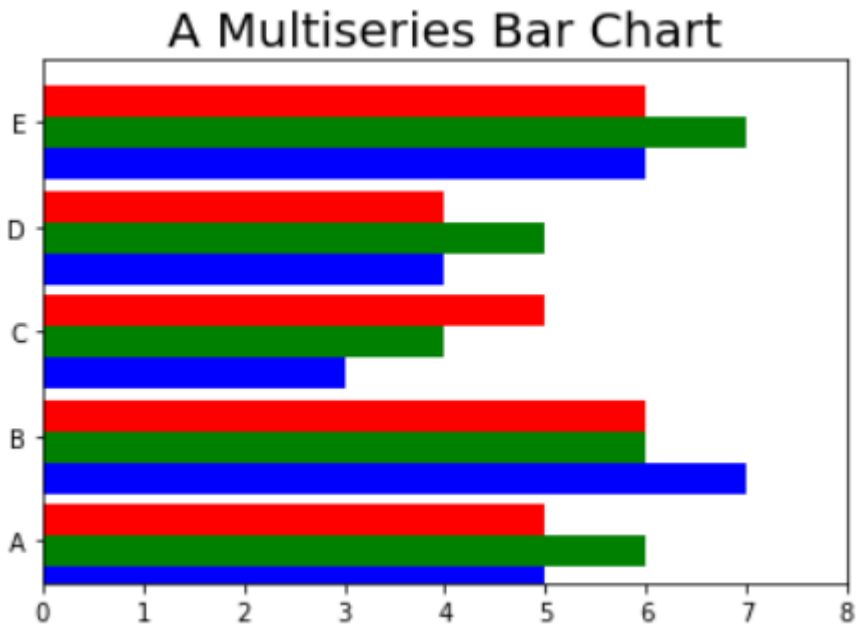
```
index = np.arange(5)
values1 = [5,7,3,4,6]
values2 = [6,6,4,5,7]
values3 = [5,6,5,4,6]
bw = 0.3
plt.axis([0,5,0,8])
plt.title('A Multiseries Bar Chart',fontsize=20)
plt.bar(index,values1,bw,color='b')
plt.bar(index+bw,values2,bw,color='g')
plt.bar(index+2*bw,values3,bw,color='r')
plt.xticks(index+1.5*bw,['A','B','C','D','E'])
```



Multiseries Horizontal Bar Chart

- Uses `barh()` function instead of `bar()` function.
- Uses `yticks()` function instead of `xticks()` function.
- Reverse the range of values covered by the axes in the `axis()` function.

```
index = np.arange(5)
values1 = [5,7,3,4,6]
values2 = [6,6,4,5,7]
values3 = [5,6,5,4,6]
bw = 0.3
plt.axis([0,8,0,5])
plt.title('A Multiseries Bar Chart',fontsize=20)
plt.barh(index,values1,bw,color='b')
plt.barh(index+bw,values2,bw,color='g')
plt.barh(index+2*bw,values3,bw,color='r')
plt.yticks(index+0.4,['A','B','C','D','E'])
```

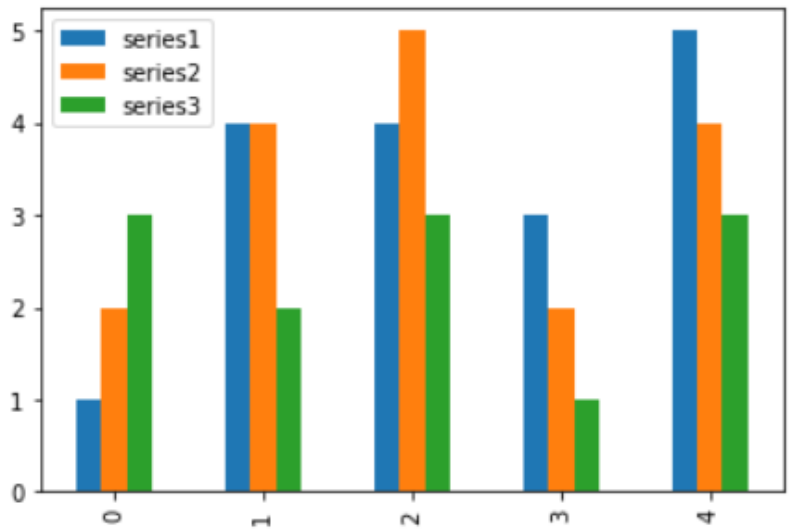


Multiseries Bar Chart with pandas Dataframe

- Use the `plot()` function applied to the dataframe object.
- Specify inside a kwarg called `kind` to which you have to assign the type of chart you want to represent, which in this case is `bar` for vertical bar chart and `barh` for a horizontal bar chart.

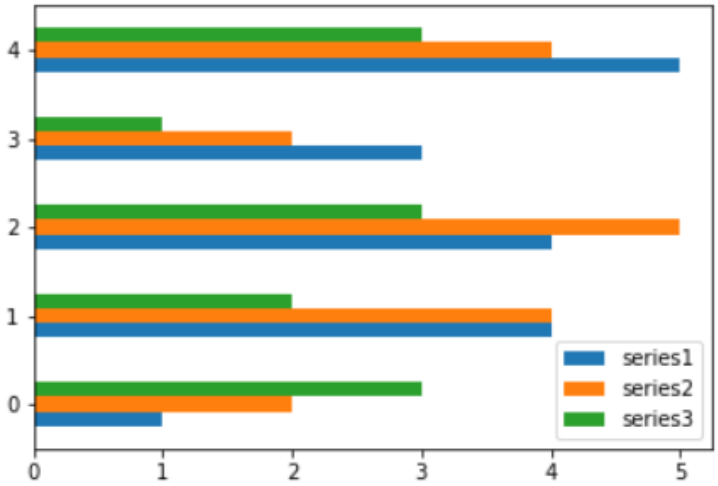
```
data = {'series1':[1,4,4,3,5],
        'series2':[2,4,5,2,4],
        'series3':[3,2,3,1,3]}
df = pd.DataFrame(data)
df.plot(kind='bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x226
```




```
data = {'series1':[1,4,4,3,5],
        'series2':[2,4,5,2,4],
        'series3':[3,2,3,1,3]}
df = pd.DataFrame(data)
df.plot(kind='barh')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x22677c33128>
```

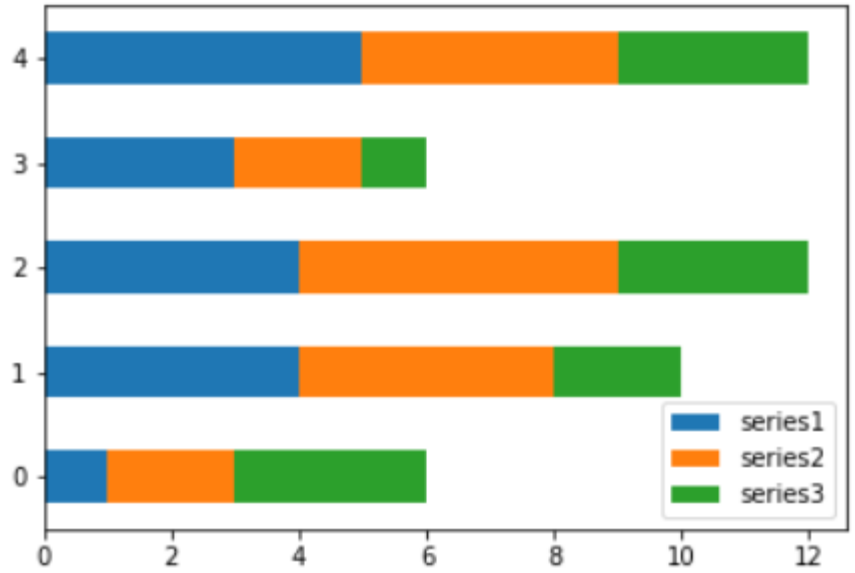


Stacked Bar Chart with pandas Dataframe

- Use a kwarg called `stacked`

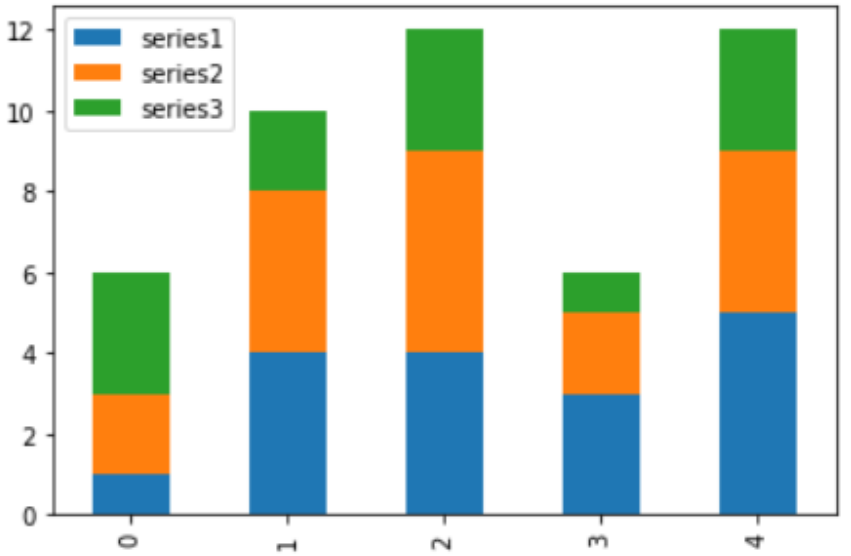
```
data = {'series1':[1,4,4,3,5],
        'series2':[2,4,5,2,4],
        'series3':[3,2,3,1,3]}
df = pd.DataFrame(data)
df.plot(kind='barh', stacked=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x226
```




```
data = {'series1':[1,4,4,3,5],
        'series2':[2,4,5,2,4],
        'series3':[3,2,3,1,3]}
df = pd.DataFrame(data)
df.plot(kind='bar', stacked=True)
```

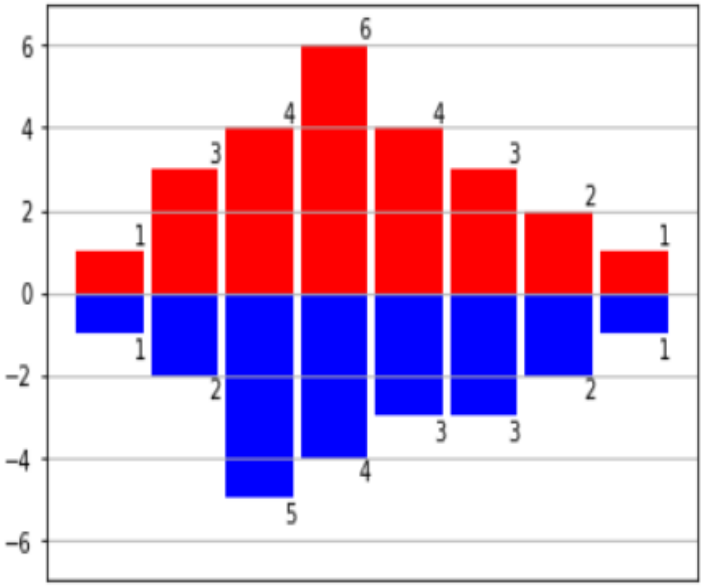
```
<matplotlib.axes._subplots.AxesSubplot at 0x2267
```



Other Bar Chart

- In this example, you want to represent one of the two series in a negative form.
- Use the `facecolor` kwargs to colour the inner colour of the bar in a different way.
- In order to add the `y` value with a label at the end of each bar (good for readability), you can use a `for loop` in which the `text()` function will show the `y` value.
- You can adjust the label position using the `ha` and `va`, which control the horizontal and vertical alignment.

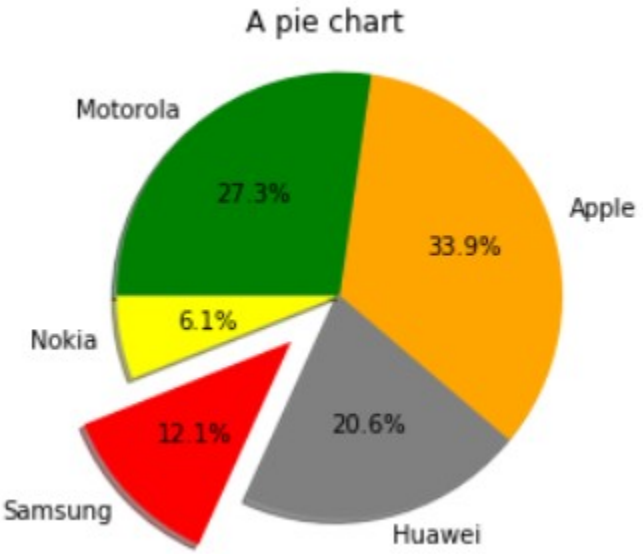
```
x0 = np.arange(8)
y1 = np.array([1,3,4,6,4,3,2,1])
y2 = np.array([1,2,5,4,3,3,2,1])
plt.ylim(-7,7)
plt.bar(x0,y1,0.9,facecolor='r')
plt.bar(x0,-y2,0.9,facecolor='b')
plt.xticks(())
plt.grid(True)
for x,y in zip(x0,y1):
    plt.text(x + 0.4, y + 0.05, '%d' % y, ha='center', va='bottom')
for x,y in zip(x0,y2):
    plt.text(x + 0.4, -y - 0.05, '%d' % y, ha='center', va='top')
```



Pie chart

- Use the `pie()` function to inherently calculate the percentage occupied by each value.
- Use the `explode` kwarg to highlight a slice. E.g. Samsung
- Use the `title()` function to add a title
- Set the `axis()` function at end to 'equal' to have a perfectly spherical pie chart.
- User the `startangle` to adjust the rotation of the pie. It takes an integer value between 0 and 360 which are the degree of rotation. (0 is the default)
- Use the `autopct` kwarg to add to the center of each slice a text label showing the corresponding value.
- Use the `shadow` kwarg to add a shadow to an image by setting it to True.

```
labels = ['Nokia','Samsung','Huawei','Apple','Motorola']
values = [10,20,34,56,45]
colors = ['yellow','red','gray','orange','green']
explode = [0,0.3,0,0,0]
plt.title('A pie chart')
plt.pie(values,labels=labels,colors=colors, explode=explode,
        shadow=True,autopct='%1.1f%%',startangle=180)
plt.axis('equal')
```

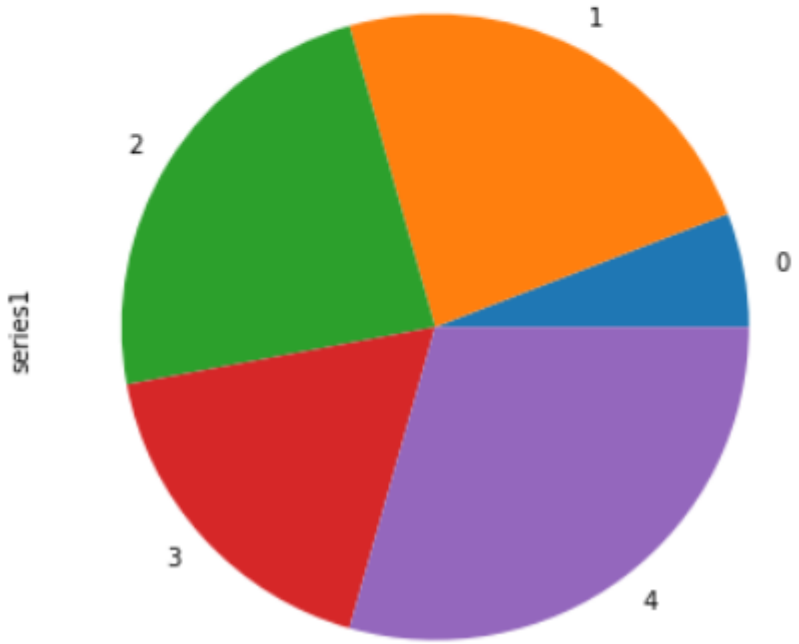


Pie chart with a pandas Dataframe

- The pie chart can only represent `one series at a time`.
- In this example, we display only the values of the first series by specifying `df['series1'].`
- The `kind` kwarg is used to specify the type of chart in the `plot()` function.
- To represent a perfectly circular pie chart, we use the `figsize` kwarg

```
data = {'series1':[1,4,4,3,5],
        'series2':[2,4,5,2,4],
        'series3':[3,2,3,1,3]}
df = pd.DataFrame(data)
df['series1'].plot(kind='pie',figsize=(6,6))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x226
```

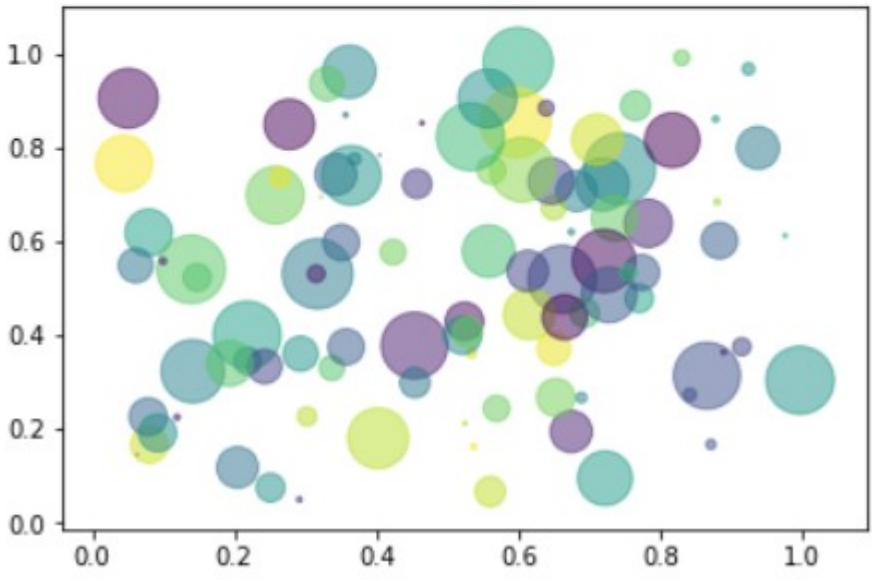


Scatter Plot

- `Scatter()` function can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

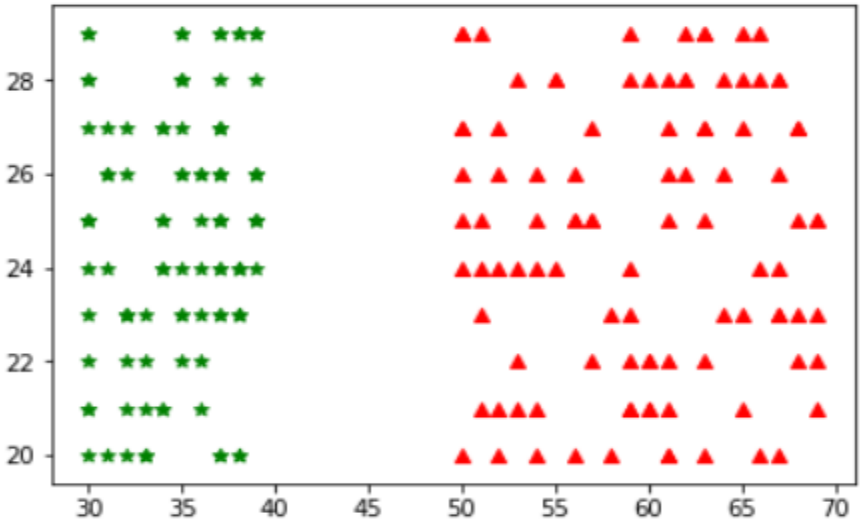
```
N = 100
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = (30 * np.random.rand(N))**2
plt.scatter(x, y, s=area, c=colors, alpha=0.5)
```

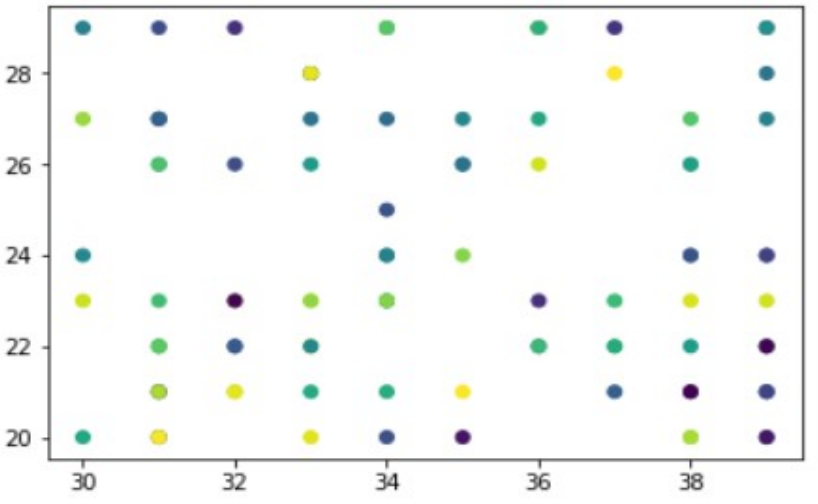
```
<matplotlib.collections.PathCollection at 0x2260
```




```
xs = np.random.randint(30,40,100)
ys = np.random.randint(20,30,100)
xs2 = np.random.randint(50,60,100)
ys2 = np.random.randint(30,40,100)
fig = plt.figure()
plt.scatter(xs,ys, c='g',marker='*')
plt.scatter(xs1,ys1,c='r',marker='^')
```

```
<matplotlib.collections.PathCollection at 0x2260
```



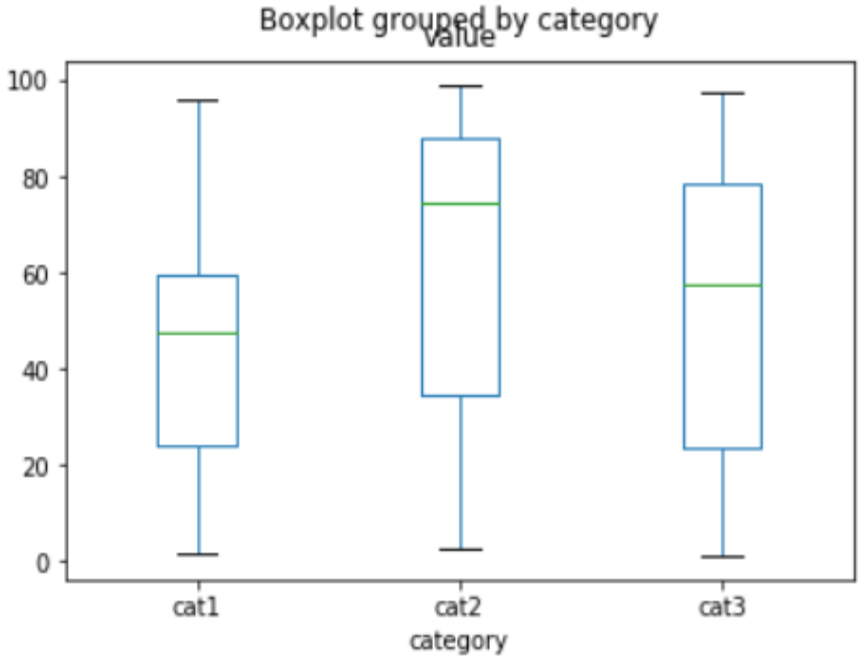


Box Plots

```
categories = ('cat1','cat2','cat3')

data = {
            'days':      np.random.randint(120, size=100),
            'category':  np.random.choice(categories, 100),
            'value':     100.0 * np.random.random_sample(100)
       }
df = pd.DataFrame(data)
df.describe()
df.boxplot(by='category',
           column=['value'],
           grid=False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2260fefeb70>
```




```
categories=('Cat1','cat2','cat3')

data = {
            'days':      np.random.randint(120, size=
            'category':  np.random.choice(categories,
            'value':     100.0 * np.random.random_sam
       }

df = pd.DataFrame(data)
```

```
df
```

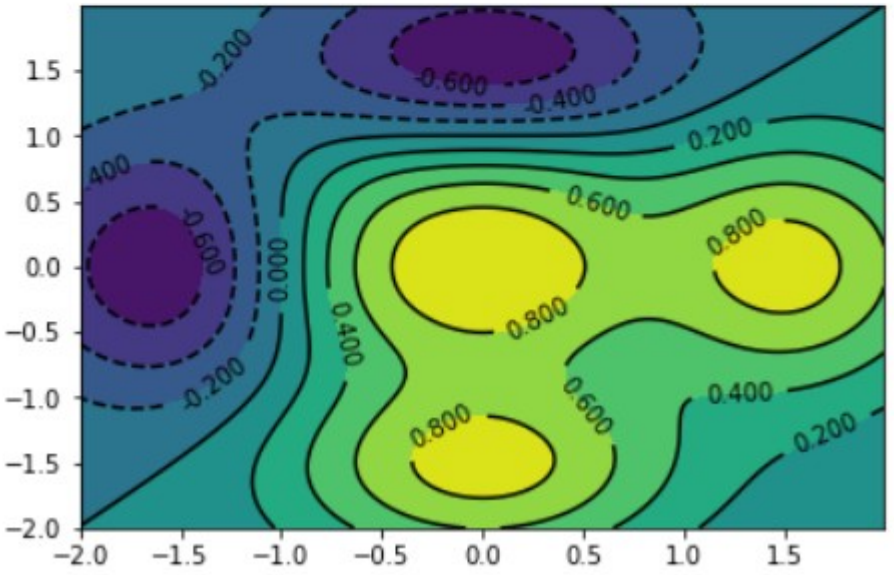
	days	category	value
0	100	Cat1	87.824120
1	102	cat3	93.348353
2	27	Cat1	2.412673

Advanced chart – Contour plot

- Contour plot or contour map is suitable for displaying three-dimensional surfaces.
- You need to $z = f(x, y)$ for generating a three-dimensional surface.
- Define a range of values for x, y that will define the area of the map to be displayed.
- Then calculate the z values for each pair (x, y), applying the function $f(x, y)$ in order to obtain a matrix of z values.
- Finally, use the `contour()` function to generate the contour of the map.
- Areas delimited by the curves of level are filled by a colour gradient, defined by a colour map. For example negative values can indicate dark shades of blue and move to yellow and then red with the increase of positive values.

```
dx = 0.01; dy = 0.01
x = np.arange(-2.0,2.0,dx)
y = np.arange(-2.0,2.0,dy)
X,Y = np.meshgrid(x,y)
def f(x,y):
    return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
C = plt.contour(X,Y,f(X,Y),8,colors='black')
plt.contourf(X,Y,f(X,Y),8)
plt.clabel(C,inline=1,fontsize=10)
```

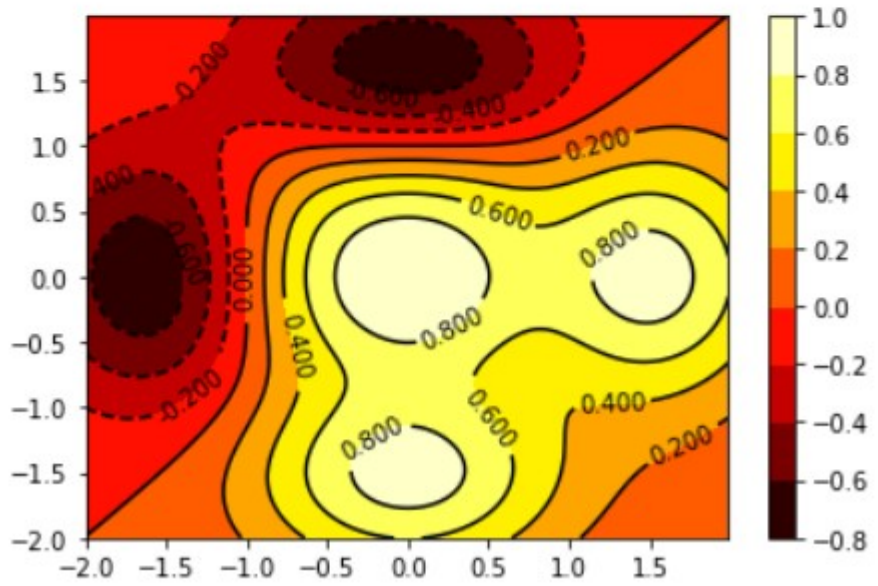
```
<a list of 16 text.Text objects>
```



- You can choose among a large number of color map available by specifying them with the `cmap` kwarg.
- Same example as previous with the 'hot' color map gradient.
- To add a colour scale as a reference by the side of the graph, use the `colorbar()` function.

```
dx = 0.01; dy = 0.01
x = np.arange(-2.0,2.0,dx)
y = np.arange(-2.0,2.0,dy)
X,Y = np.meshgrid(x,y)
def f(x,y):
    return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
C = plt.contour(X,Y,f(X,Y),8,colors='black')
plt.contourf(X,Y,f(X,Y),8, cmap=plt.cm.hot)
plt.clabel(C,inline=1,fontsize=10)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x2267a0f4fd0>
```



Using mplot3d Toolkit for 3D Surfaces

- The mplot3d toolkits is included in all standard installation of matplotlib and allows us to extend the capabilities of visualisation to 3D data.
- We use an object called Axes3D

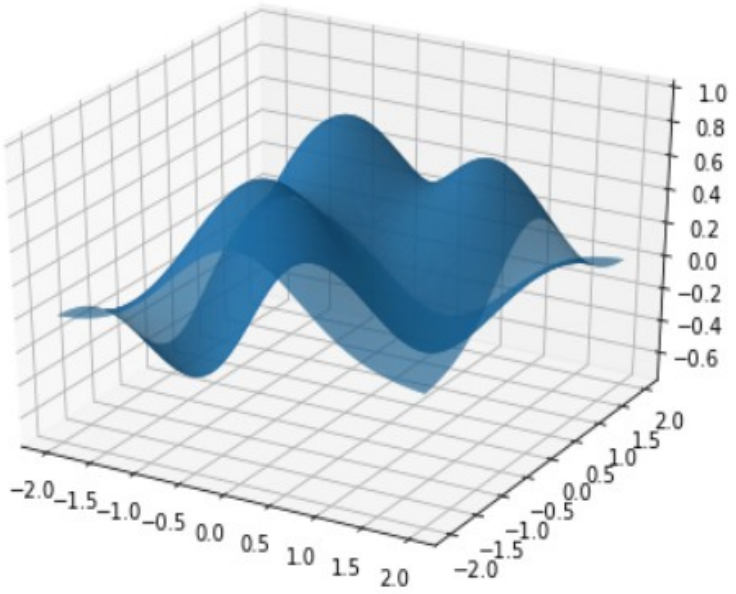
```
from mpl_toolkits.mplot3d import Axes3D
```

- In this example, we use the same function: `z =f(x,y)` used for contour map

- Once we have calculated the `meshgrid`, we can view the surface with the `plot_surface()` function.

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = Axes3D(fig)
dx = 0.01; dy = 0.01
x = np.arange(-2.0,2.0,dx)
y = np.arange(-2.0,2.0,dy)
X,Y = np.meshgrid(x,y)
def f(x,y):
    return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
ax.plot_surface(X,Y,f(X,Y),rstride=1,cstride=1)
```

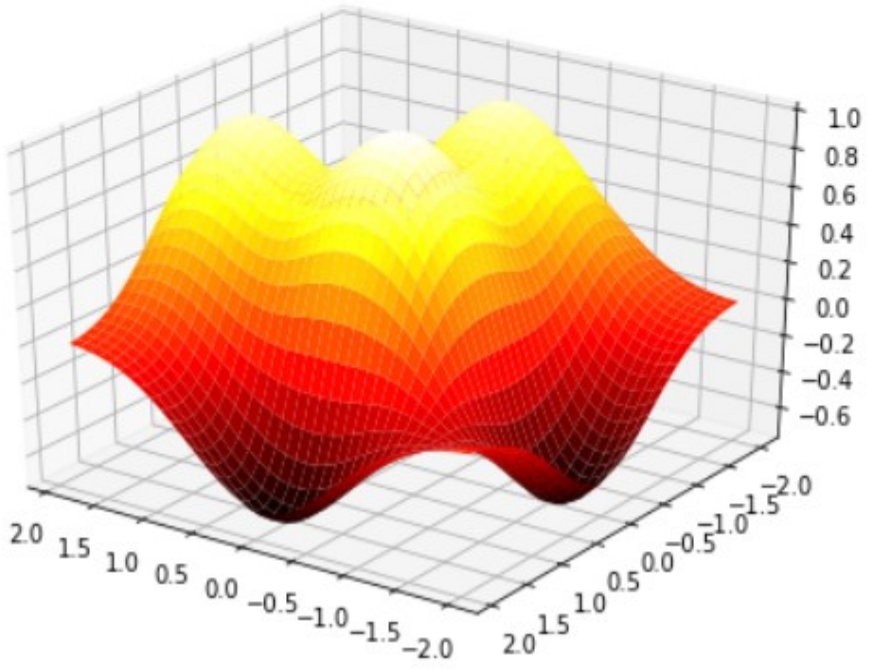
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x22609e4fac8>
```



Using mplot3d Toolkit for 3D Surfaces

- Use `cmap` kwarg to change the color. Rotate the surface by using the `view_init()` function.
- The `elev` kwarg adjust the height at which the surface is seen and the `azim` kwarg adjusts the angle of rotation of the surface.
- In this example, the 3D surface is rotated and observed from a higher viewpoint.

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-2.0,2.0,0.1)
Y = np.arange(-2.0,2.0,0.1)
X,Y = np.meshgrid(X,Y)
def f(x,y):
    return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
ax.plot_surface(X,Y,f(X,Y),rstride=1,cstride=1, cmap=plt.cm.hot)
ax.view_init(elev=30,azim=125)
```

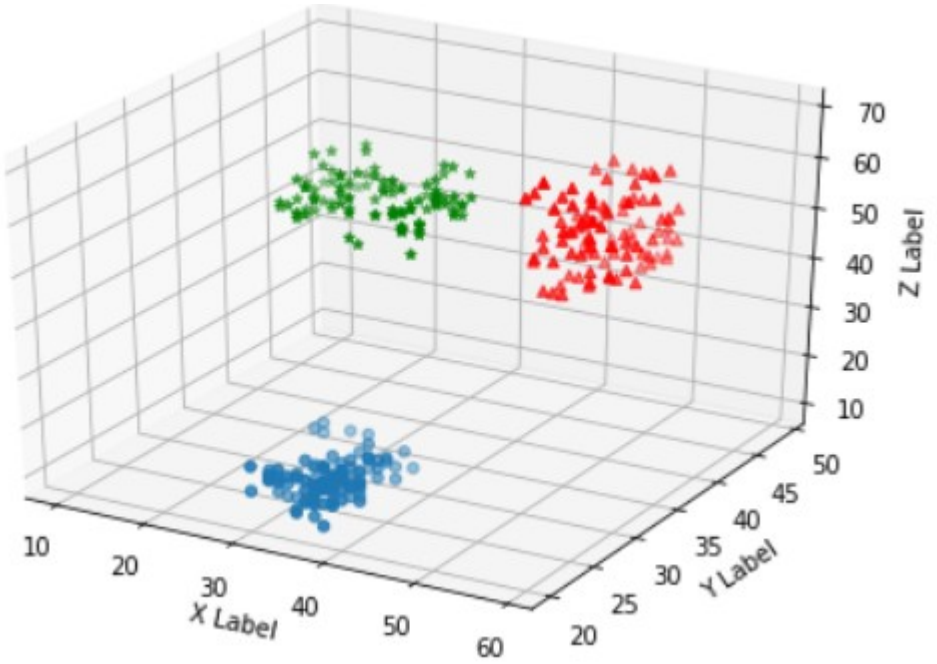


Scatter Plots in 3D

- The most used among all 3D views is the 3d scatter plot. With this type of visualisation, you can identify if the points follow particular trends and if they tend to cluster.

- We use the `scatter()` function as the 2D case but applied on the Axes3D object. By doing this, we can visualise different series all together in the same 3D representation.

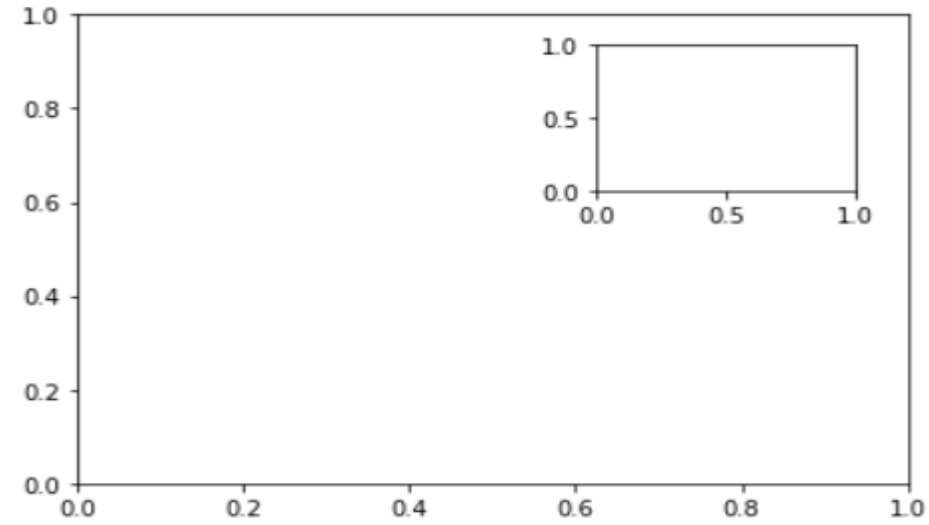
```
from mpl_toolkits.mplot3d import Axes3D
xs = np.random.randint(30,40,100)
ys = np.random.randint(20,30,100)
zs = np.random.randint(10,20,100)
xs2 = np.random.randint(50,60,100)
ys2 = np.random.randint(30,40,100)
zs2 = np.random.randint(50,70,100)
xs3 = np.random.randint(10,30,100)
ys3 = np.random.randint(40,50,100)
zs3 = np.random.randint(40,50,100)
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(xs,ys,zs)
ax.scatter(xs2,ys2,zs2,c='r',marker='^')
ax.scatter(xs3,ys3,zs3,c='g',marker='*')
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
```



Subplots Within Other subplots

- Since we are talking of frames (i.e. Axes objects), we need to separate the main Axes (i.e., the general chart) from the frame we want to add that will need another instance of Axes.
- To do this we use the `figure()` function to get the Figure object on which we define two different Axes objects using the `add_axes()` function

```
fig = plt.figure()
ax = fig.add_axes([0.1,0.1,0.8,0.8])
inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
```




```
fig = plt.figure()
ax = fig.add_axes([0.1,0.1,0.8,0.8])
inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
x1 = np.arange(10)
y1 = np.array([1,2,7,1,5,2,4,2,3,1])
x2 = np.arange(10)
y2 = np.array([1,3,4,5,4,5,2,6,4,3])
ax.plot(x1,y1)
inner_ax.plot(x2,y2)
```

```
[<matplotlib.lines.Line2D at 0x22627bec358>]
```