

**Name: Shruti Rajesh Gadre**

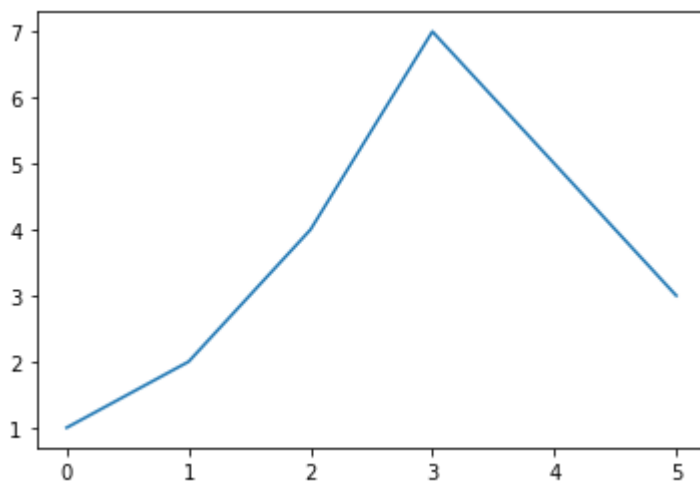
## **Task 7 - Create a Basic Data Visualization with Matplotlib**

### **Plotting the first graph**

First we need to import the `matplotlib` and `seaborn` library.

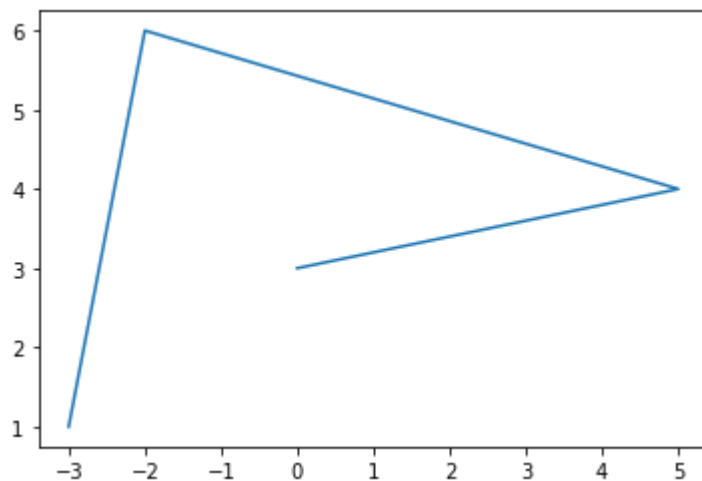
```
In [1]: import matplotlib.pyplot as plt  
import pandas as pd  
import numpy as np  
import seaborn as sb
```

```
In [2]: x = [1, 2, 4, 7, 5, 3]  
plt.plot(x)  
plt.show()
```



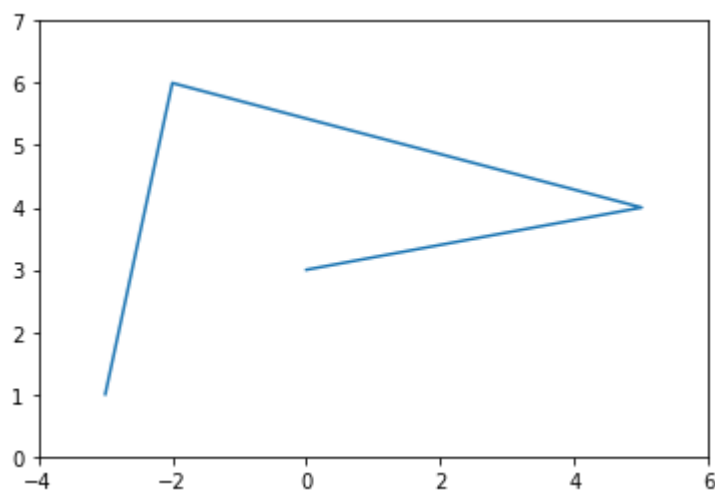
If the `plot` function is given one array of data, it will use it as the coordinates on the vertical axis, and it will just use each data point's index in the array as the horizontal coordinate. We can also provide two arrays: one for the horizontal axis `x`, and the second for the vertical axis `y`:

```
In [3]: plt.plot([-3, -2, 5, 0], [1, 6, 4, 3])  
plt.show()
```



The axes automatically match the extent of the data. If we would like to give the graph a bit more room, so let's call the `axis` function to change the extent of each axis `[xmin, xmax, ymin, ymax]`.

```
In [6]: plt.plot([-3, -2, 5, 0], [1, 6, 4, 3])  
plt.axis([-4, 6, 0, 7])  
plt.show()
```



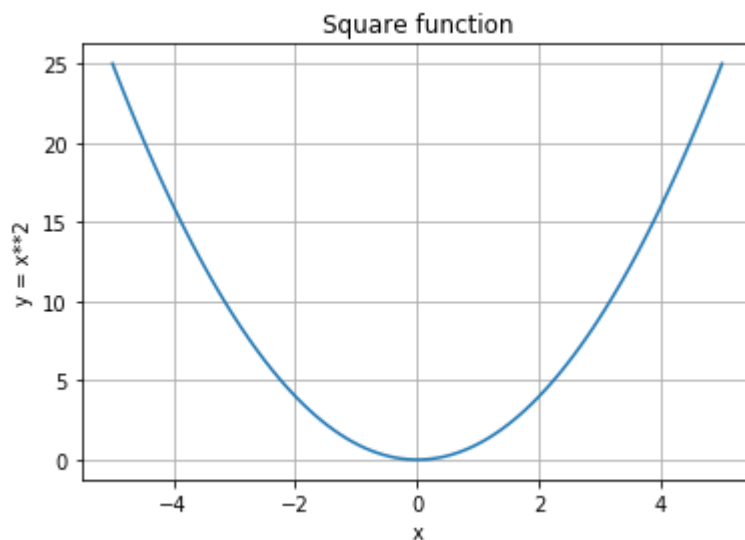
## Adding Title Labels and Grid

Adding a title, and x and y labels, and **drawing a grid**.

```
In [9]: x = np.linspace(-5, 5, 100)

y = x**2

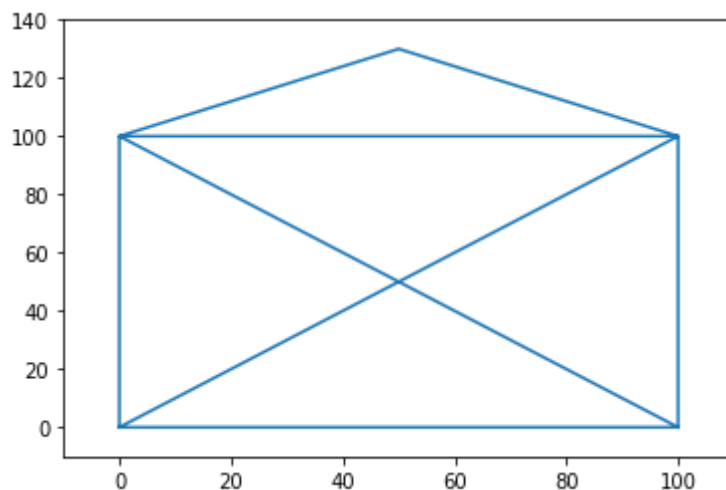
# Creating the plot
plt.plot(x, y)
plt.title("Square function")
plt.xlabel("x")
plt.ylabel("y = x**2")
plt.grid(True) # Adding Grid
plt.show()
```



## Line style and color

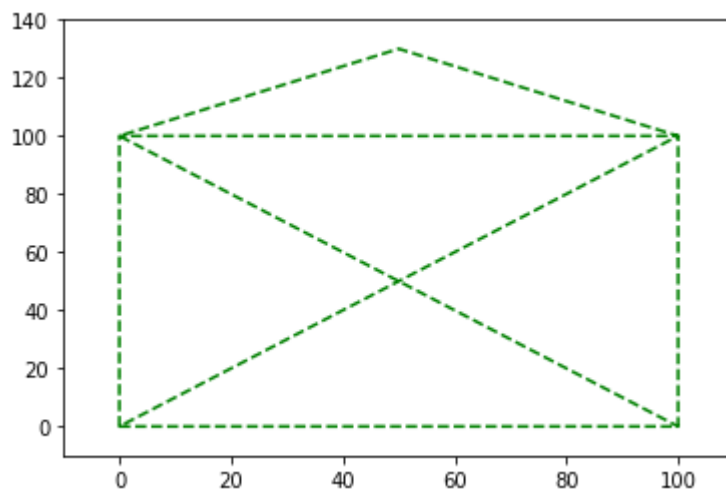
By default, matplotlib draws a line between consecutive points.

```
In [10]: plt.plot([0, 100, 100, 0, 0, 100, 50, 0, 100], [0, 0, 100, 100, 0, 100, 130, 100, 0])
plt.axis([-10, 110, -10, 140])
plt.show()
```



We can pass a 3rd argument to change the line's style and color. For example "g--" means "green dashed line".

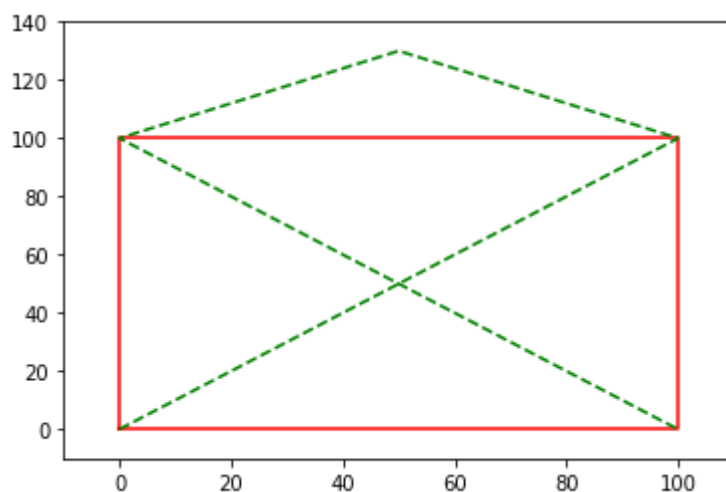
```
In [11]: plt.plot([0, 100, 100, 0, 0], [0, 0, 100, 100, 0], "g--")  
plt.plot([0, 100, 100, 0, 0], [100, 100, 0, 0, 100], "g--")  
plt.axis([-10, 110, -10, 140])  
plt.show()
```



We can plot multiple lines on one graph: just by passing `x1, y1, [style1], x2, y2, [style2], ...`

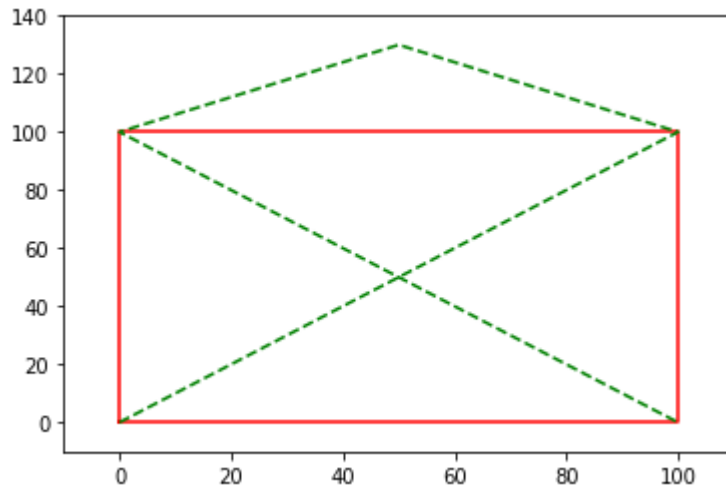
For example:

```
In [12]: plt.plot([0, 100, 100, 0, 0], [0, 0, 100, 100, 0], "r-", [0, 100, 50, 0, 100],  
[0, 100, 130, 100, 0], "g--")  
plt.axis([-10, 110, -10, 140])  
plt.show()
```



Or simply call `plot` multiple times before calling `show`.

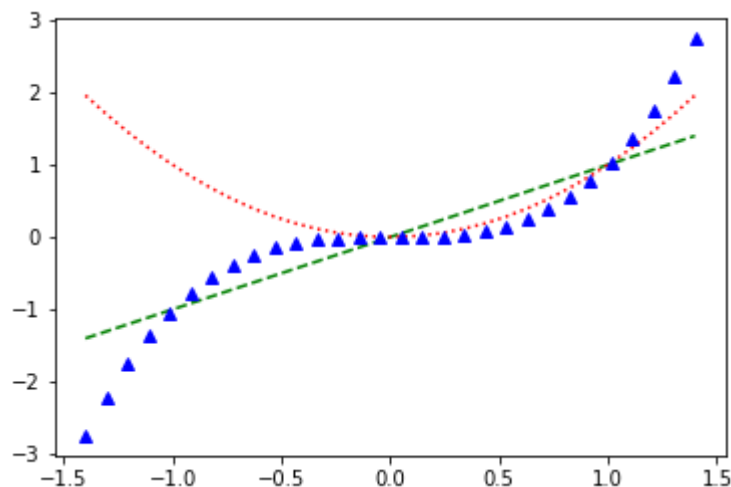
```
In [13]: plt.plot([0, 100, 100, 0, 0], [0, 0, 100, 100, 0], "r-") # line in red
plt.plot([0, 100, 50, 0, 100], [0, 100, 130, 100, 0], "g--") # line in green
plt.axis([-10, 110, -10, 140])
plt.show()
```



We can also draw simple points instead of lines. Here's an example with green dashes, red dotted line and blue triangles. Referred to [the documentation \(http://matplotlib.org/api/pyplot\\_api.html#matplotlib.pyplot.plot\)](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot) for the full list of style & color options.

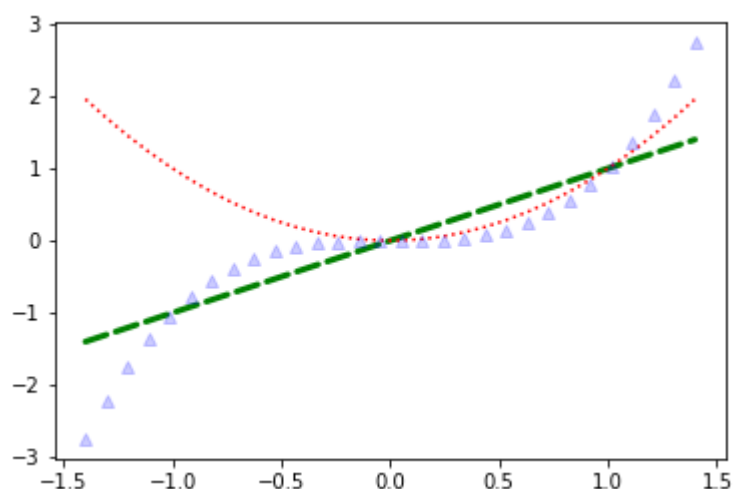
```
In [14]: x = np.linspace(-1.4, 1.4, 30)
print(x)
plt.plot(x, x, 'g--', x, x**2, 'r:', x, x**3, 'b^')
plt.show()
```

```
[-1.4      -1.30344828 -1.20689655 -1.11034483 -1.0137931  -0.91724138
 -0.82068966 -0.72413793 -0.62758621 -0.53103448 -0.43448276 -0.33793103
 -0.24137931 -0.14482759 -0.04827586  0.04827586  0.14482759  0.24137931
  0.33793103  0.43448276  0.53103448  0.62758621  0.72413793  0.82068966
  0.91724138  1.0137931   1.11034483  1.20689655  1.30344828  1.4       ]
```

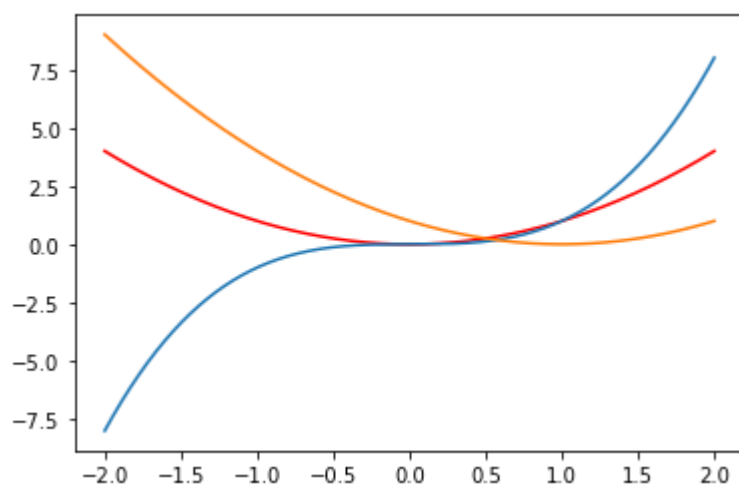


The plot function returns a list of `Line2D` objects (one for each line). We can set extra attributes on these lines, such as the line width, the dash style or the alpha level.

```
In [15]: x = np.linspace(-1.4, 1.4, 30)
line1, line2, line3 = plt.plot(x, x, 'g--', x, x**2, 'r:', x, x**3, 'b^')
line1.set_linewidth(3.0)
line1.set_dash_capstyle("round")
line3.set_alpha(0.2)
plt.show()
```



```
In [17]: x= np.linspace(-2, 2, 500)
plt.plot(x, x**2, 'r-', x, x**3, x, (x-1)**2)
plt.show()
```



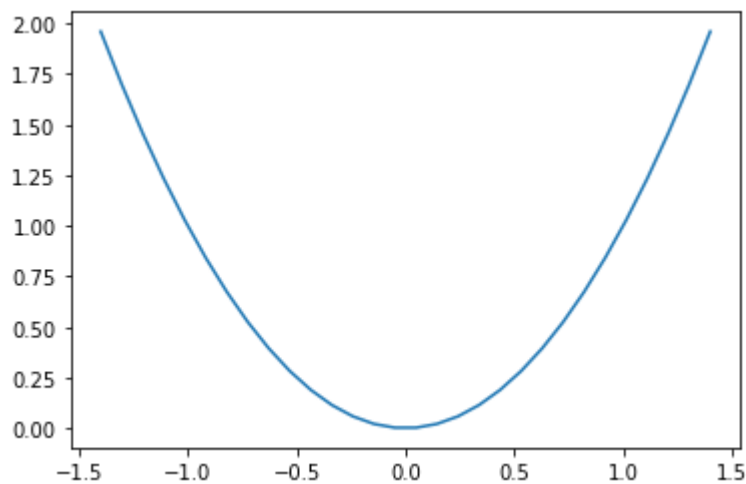
## Saving a figure

Saving a figure to disk is as simple as calling `savefig`

([http://matplotlib.org/api/pyplot\\_api.html#matplotlib.pyplot.savefig](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.savefig)) with the name of the file (or a file object).

The available image formats depend on the graphics backend we use.

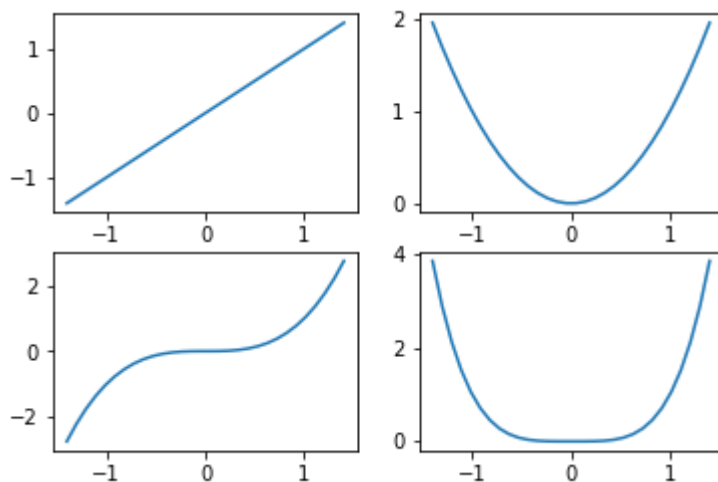
```
In [18]: x = np.linspace(-1.4, 1.4, 30)
plt.plot(x, x**2)
plt.savefig("my_square_function1.png", transparent=True, dpi=300)
```



## Subplots

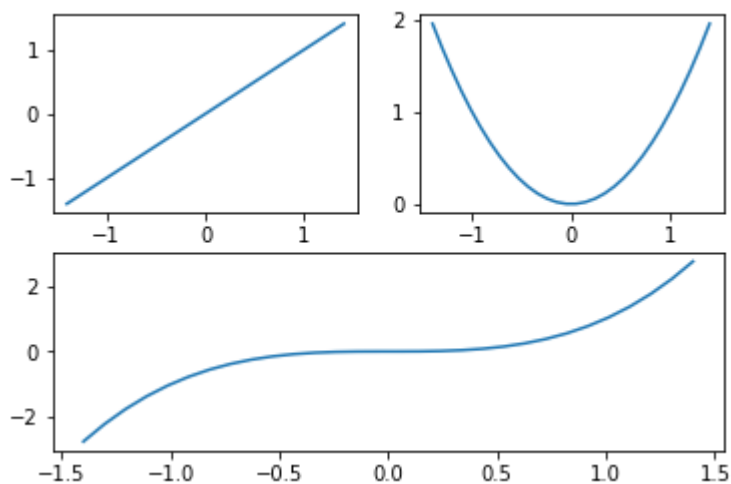
The subplots are organized in a grid. To create a subplot, just call the `subplot` function, and specify the number of rows and columns in the figure, and the index of the subplot we want to draw on (starting from 1, then left to right, and top to bottom). Note that pyplot keeps track of the currently active subplot, so when we call the `plot` function, it draws on the *active* subplot.

```
In [19]: x = np.linspace(-1.4, 1.4, 30)
plt.subplot(2, 2, 1) # 2 rows, 2 columns, 1st subplot = top left
plt.plot(x, x)
plt.subplot(2, 2, 2) # 2 rows, 2 columns, 2nd subplot = top right
plt.plot(x, x**2)
plt.subplot(2, 2, 3) # 2 rows, 2 columns, 3rd subplot = bottom left
plt.plot(x, x**3)
plt.subplot(2, 2, 4) # 2 rows, 2 columns, 4th subplot = bottom right
plt.plot(x, x**4)
plt.show()
```



- Note that `subplot(223)` is a shorthand for `subplot(2, 2, 3)`.

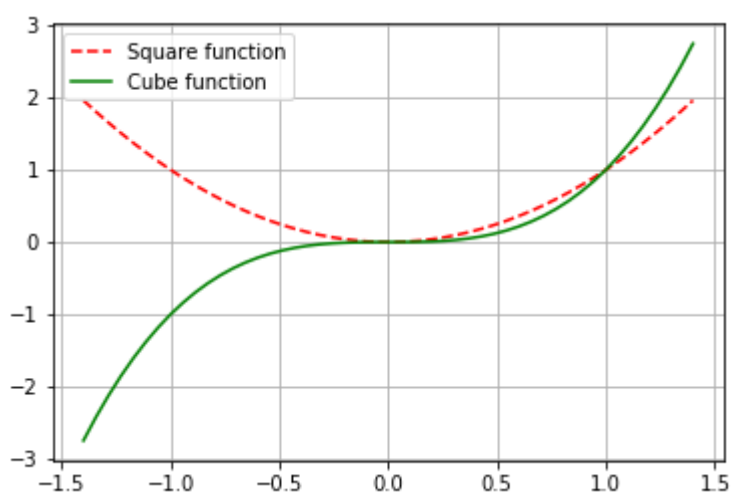
```
In [20]: plt.subplot(2, 2, 1) # 2 rows, 2 columns, 1st subplot = top left
plt.plot(x, x)
plt.subplot(2, 2, 2) # 2 rows, 2 columns, 2nd subplot = top right
plt.plot(x, x**2)
plt.subplot(2, 1, 2) # 2 rows, *1* column, 2nd subplot = bottom
plt.plot(x, x**3)
plt.show()
```



## Legends

The simplest way to add a legend is to set a label on all lines, then just call the `legend` function.

```
In [21]: x = np.linspace(-1.4, 1.4, 50)
plt.plot(x, x**2, "r--", label="Square function")
plt.plot(x, x**3, "g-", label="Cube function")
plt.legend(loc="best")
plt.grid(True)
plt.show()
```





# Ticks and tickers

The axes have little marks called "ticks". They are the small lines drawn at those locations, the labels drawn next to the tick lines, and "tickers" are objects that are capable of deciding where to place ticks. The default tickers typically do a pretty good job at placing ~5 to 8 ticks at a reasonable distance from one another.

But sometimes we need more control. Fortunately, matplotlib gives us full control over ticks. We can even activate minor ticks.

```

In [22]: x = np.linspace(-2, 2, 100)

plt.figure(1, figsize=(15,10))
plt.subplot(131)
plt.plot(x, x**3)
plt.grid(True)
plt.title("Default ticks")

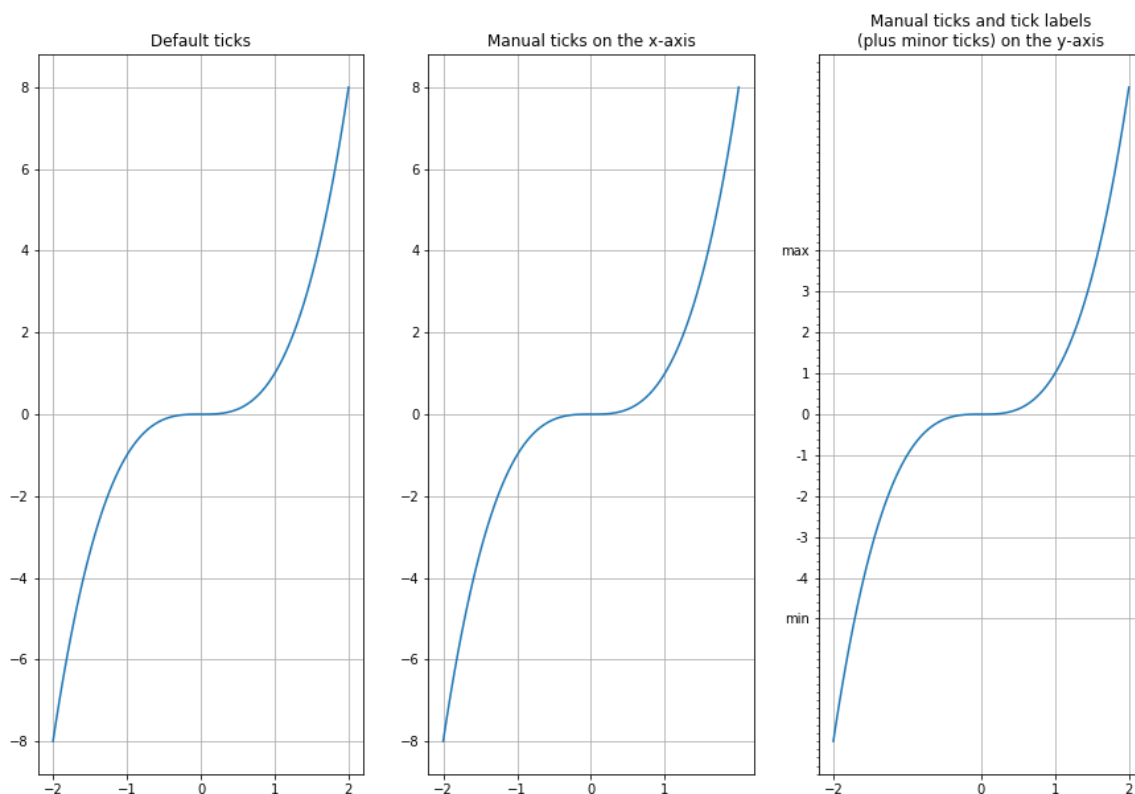
ax = plt.subplot(132)
plt.plot(x, x**3)
ax.xaxis.set_ticks(np.arange(-2, 2, 1))
plt.grid(True)
plt.title("Manual ticks on the x-axis")

ax = plt.subplot(133)
plt.plot(x, x**3)
plt.minorticks_on()
ax.tick_params(axis='x', which='minor', bottom=False)
ax.xaxis.set_ticks([-2, 0, 1, 2])
ax.yaxis.set_ticks(np.arange(-5, 5, 1))
ax.yaxis.set_ticklabels(["min", -4, -3, -2, -1, 0, 1, 2, 3, "max"])
plt.title("Manual ticks and tick labels\n(plus minor ticks) on the y-axis")

plt.grid(True)







plt.show()

```



# Data Measurement Levels

From a statistical perspective, there is more to variable types than the simple distinction between **numeric** and **categorical** data. There are, in fact, four so called data measurement levels, which define what the variable really means on what kind of mathematical operations can be applied to it.

- **Nominal Data**    The term “nominal” comes from Latin and can be translated as “being so in name only”. The only information carried by nominal data is the group an observation belongs to. An example of this could be color. You could encode yellow as 1 and blue as 2, but these numbers would have no specific value nor meaning. And such an encoding would not automatically make green equal to 1.5. When it comes to mathematical operations, there ain’t much you can do here. One can only compute the mode (the most frequent value) of a nominal variable. Other measures, such as the mean or the median, make no sense.
- **Ordinal Data**  As the name suggests, ordinal data have some order. It allows you to rank the values, such as for education level: elementary is less than high\_school, which in turn is less than university. The ordering allows us to compute the median: if our dataset has 100 examples of each education level, it is correct to say that the median education is high\_school. From the definition of the median, this means is that 50% of the examples have high\_school or elementary education, while the other 50% have high\_school or university education, which is correct and, maybe, an important insight. Calculating the mean, however, makes no sense for ordinal data. What is the average of a university and an elementary school anyway? A note for the mathematically-inclined reader: if you encode the levels of an ordinal variable with subsequent numbers, you can safely apply monotone transformations to it, such as taking the logarithm. This is because monotone transformations preserve the order, and the order is all that matters here.
- **Interval Data**  Interval data builds on top ordinal data. In addition to ordering the values, it also specifies that the intervals between subsequent values are the same. A good example of this is the temperature measured in degrees Celsius: the difference between 1 degree and 5 degrees is the same as between 20 and 24: it’s 4 degrees. Note that this was not the case for ordinal data: we cannot say that the difference between graduating from a high school versus from an elementary school only is the same as the one between a university and a high school. In the case of interval type variables, in addition to the mode and the median, computing the arithmetic mean also makes sense. You can also apply linear transformations to interval data.
- **Ratio Data**  Ratio data builds on top of interval data. The difference is that ratio type variables have a meaningful zero value. The examples are price, length, weight, amount of something, or temperature measured in Kelvin. The meaningful zero allows us to calculate ratios between two data points: we can say that 4 apples are twice as much as 2, or that rs 5 is half as expensive as rs 10. This was not the case for interval data: in the case of temperature measured in degrees Celsius, we cannot say 10 degrees is twice as warm as 5 degrees. Ratios make no sense for scales without a meaningful zero. Since we can take ratios of ratio type data, we can also apply scaling transformations, such as multiplication.

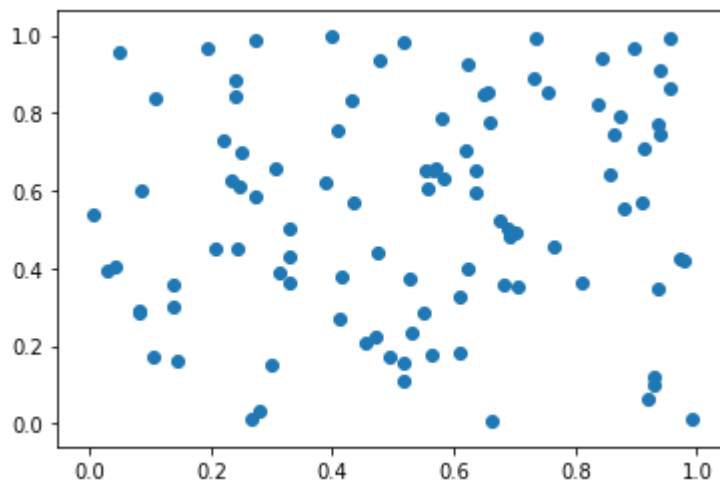
Data Measurement Type	Property	Central Tendency Measure	Allowed math	Example
NOMINAL	grouping	mode	-	color
ORDINAL	order	mode, median	monotone transformations	education level
INTERVAL	equal intervals	mode, median, mean	linear transformations	temperature in °C
RATIO	meaningful zero	mode, median, mean	scaling transformations	price

# Scatter plot

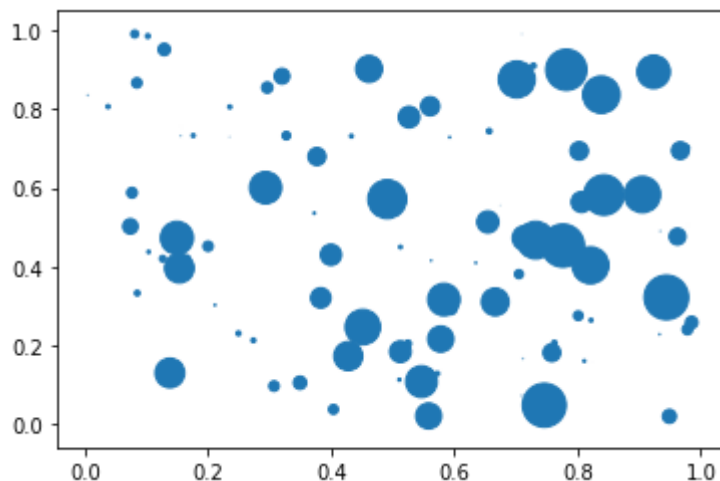
A **scatter plot** (also called a scatterplot, scatter graph, scatter chart, scattergram, or scatter diagram) is a type of plot or mathematical diagram using Cartesian coordinates to display values for typically two variables for a set of data. If the points are coded (color/shape/size), one additional variable can be displayed. The data are displayed as a collection of points, each having the value of one variable determining the position on the horizontal axis and the value of the other variable determining the position on the vertical axis.

To draw a scatter plot, simply provide the x and y coordinates of the points.

```
In [23]: from numpy.random import rand
x, y = rand(2, 100)
plt.scatter(x, y)
plt.show()
```



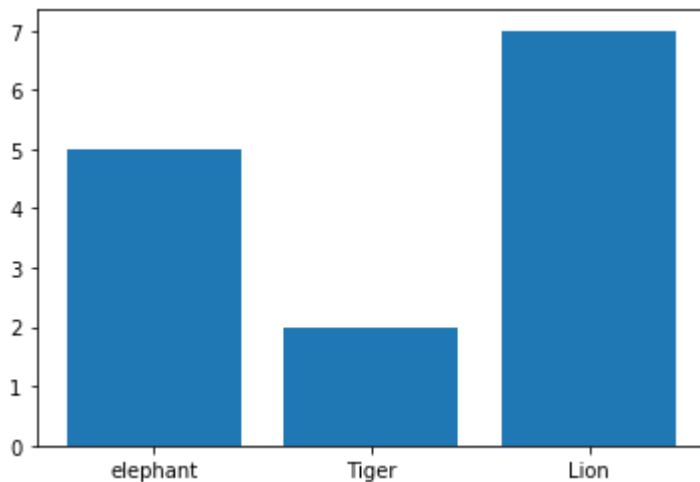
```
In [24]: x, y, scale = rand(3, 100)
scale = 500 * scale ** 5
plt.scatter(x, y, s=scale)
plt.show()
```



# Bar Chart

A bar chart or bar graph is a chart or graph that presents **categorical data** with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. A vertical bar chart is sometimes called a column chart.

```
In [25]: plt.bar(["elephant", "Tiger", "Lion"], [5, 2, 7])  
plt.show()
```



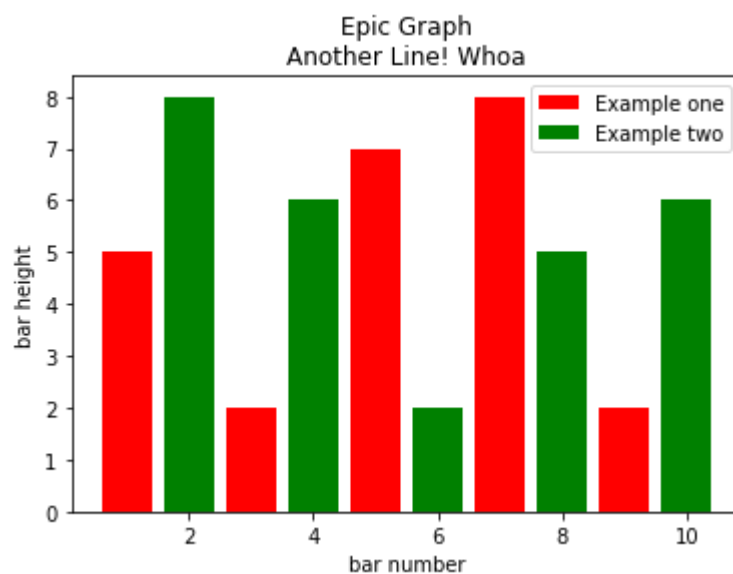
The `plt.bar` creates the bar chart for us. If you do not explicitly choose a **color**, then, despite doing multiple plots, all bars will look the same.

```
In [26]: plt.bar([1,3,5,7,9],[5,2,7,8,2], label="Example one",color='r')

plt.bar([2,4,6,8,10],[8,6,2,5,6], label="Example two", color='g')
plt.legend()
plt.xlabel('bar number')
plt.ylabel('bar height')

plt.title('Epic Graph\nAnother Line! Whoa')

plt.show()
```



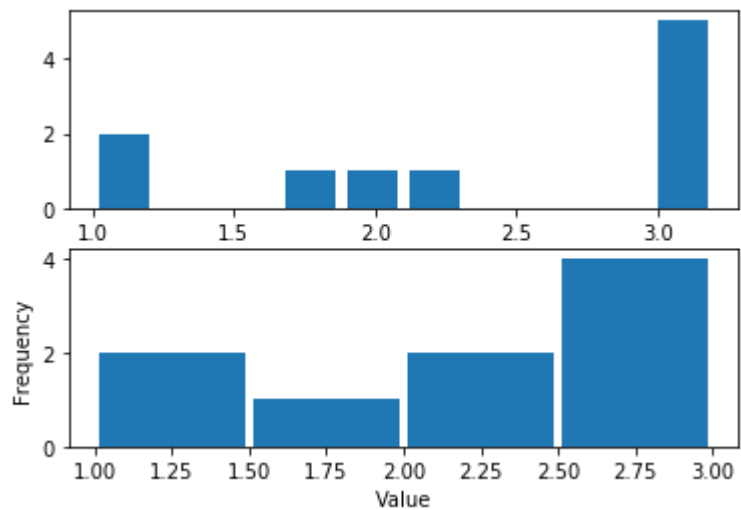
## Histograms

A histogram is a graphical representation that organizes a group of data points into user-specified ranges. Similar in appearance to a bar graph, the histogram condenses a data series into an easily interpreted visual by taking many data points and grouping them into logical ranges or bins.

```
In [27]: data = [1, 1.1, 1.8, 2, 2.1, 3.2, 3, 3, 3, 3]
plt.subplot(211)
plt.hist(data, bins = 10, rwidth=0.8)

plt.subplot(212)
plt.hist(data, bins = [1, 1.5, 2, 2.5, 3], rwidth=0.95)
plt.xlabel("Value")
plt.ylabel("Frequency")

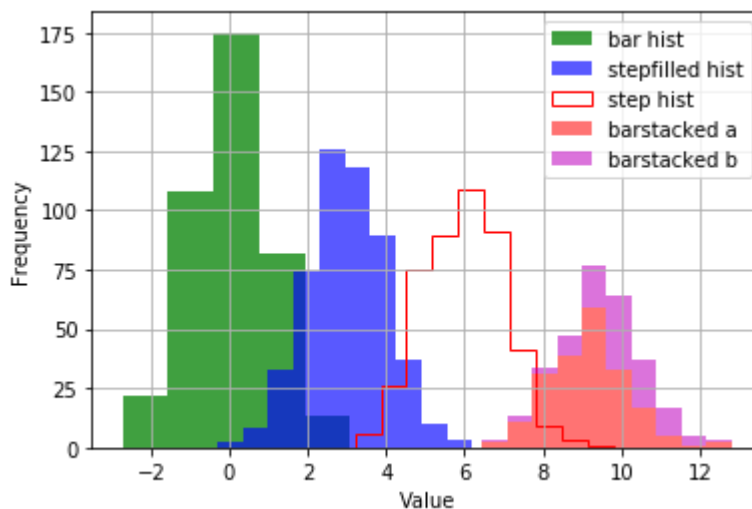
plt.show()
```



```
In [28]: data1 = np.random.randn(400)
data2 = np.random.randn(500) + 3
data3 = np.random.randn(450) + 6
data4a = np.random.randn(200) + 9
data4b = np.random.randn(100) + 10

plt.hist(data1, bins=5, color='g', alpha=0.75, label='bar hist') # default
                        histtype='bar'
plt.hist(data2, color='b', alpha=0.65, histtype='stepfilled', label='stepfi
                        lled hist')
plt.hist(data3, color='r', histtype='step', label='step hist')
plt.hist((data4a, data4b), color=('r','m'), alpha=0.55, histtype='barstacke
                        d', label=('barstacked a', 'barstacked b'))

plt.xlabel("Value")
plt.ylabel("Frequency")
plt.legend()
plt.grid(True)
plt.show()
```



## Line Chart

A line chart or line plot or line graph or curve chart is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments. It is a basic type of chart common in many fields. It is similar to a scatter plot except that the measurement points are ordered (typically by their x-axis value) and joined with straight line segments. A line chart is often used to visualize a trend in data over intervals of time – a time series – thus the line is often drawn chronologically. In these cases they are known as run charts.



```
In [29]: Year = [1920,1930,1940,1950,1960,1970,1980,1990,2000,2010]
Unemployment_Rate = [9.8,12,8,7.2,6.9,7,6.5,6.2,5.5,6.3]
plt.figure(dpi=200)
plt.plot(Year, Unemployment_Rate, color='red', marker='o')
plt.title('Unemployment Rate Vs Year', fontsize=14)
plt.xlabel('Year', fontsize=14)
plt.ylabel('Unemployment Rate', fontsize=14)
plt.grid(True)
plt.show()
```



# Report on Basic Data Visualization with Matplotlib

## Data Import and Preparation:

For this task, we loaded a sample dataset into our data analysis environment. The dataset contains information about monthly sales figures for a retail store.

## Matplotlib Installation:

Ensured that Matplotlib is installed in the Python environment using the command "pip install matplotlib."

## Visualization Type:

After reviewing the dataset, I have decided to create a line plot to visualize the trend using the Matplotlib graphs and plots.

## Documentation:

The purpose of this visualization was to provide a clear representation of the different plots and graphs present in Matplotlib.

## Testing and Validation:

Have double-checked the code and visualization to ensure accuracy and clarity. The plot accurately represents the provided data, and the visual elements are appropriately customized.

## Visualization Enhancements:

For further enhancements, I have experimented with different line styles, colors, and annotations to make the plots more visually appealing or informative.

## Conclusion

Matplotlib is a powerful and versatile library for data visualization in Python. It provides a wide range of tools and functions for creating various types of plots and charts, making it an essential tool for data analysts, scientists, and engineers. In this conclusion, we highlight key takeaways about Matplotlib:

### Flexibility and Customization:

Matplotlib allows users to customize virtually every aspect of a plot, including colors, markers, labels, axes, and more. This flexibility ensures that visualizations can be tailored to specific requirements and preferences.

### Diverse Plot Types:

Matplotlib supports a wide variety of plot types, from basic line and scatter plots to complex heatmaps, 3D plots, and geographic visualizations. This versatility makes it suitable for a broad range of data analysis tasks.

**Integration with Other Libraries:**

Matplotlib seamlessly integrates with other Python libraries such as NumPy, Pandas, and SciPy, making it a key component of the scientific Python ecosystem.

**Interactive Plotting:**

Matplotlib can be used in conjunction with interactive plotting libraries like Jupyter Notebook widgets or libraries like mpld3 to create interactive visualizations for exploration and presentation.

In conclusion, Matplotlib is a foundational library for data visualization in Python. Its capabilities extend from basic plotting to highly customized, publication-ready graphics. For anyone working with data in Python, Matplotlib is an essential tool to effectively communicate insights and findings through visual representations.