Visualization

Outline

- **Uses/necessity of matplotlib**
- **Anatomy**
  - Figure
- **plt.plot()**
  - Passing a single list
  - Passing 2 lists
  - Passing numpy arrays
- **plt.show()**
- **savefig**
- **Scale of axes**:
  - xticks
  - yticks
- **Styling and labeling**
  - xlabel
  - ylabel
  - title
  - color
  - Shape of plot
- **Setting range of axes**
  - xlim
  - ylim
- **Displaying legends**
  - plt.legend()
- **Multiple curves on same plot**
  - Adding plt.text()
  - plt.grid()
  - Short method of multiple curves
- **Different types of plots**:
  - Bar graph
    - Height/width/colour of bars
    - Multiple bar graphs on same plot
    - Shifting x-vals to prevent overlap
  - Histogram
    - Diff b/w bar and histogram
    - counts/bins
    - Changing number of bins
  - Scatter plot
    - About scatter plots
    - Multiple scatter plots in 1 plot
    - Color
    - Diff between plt.plot() and plt.scatter()
  - Pie charts
  - Drawing plot of sigmoid
    - Analysis
  - Subplots
    - Arguments: row, cols, pos
- **3D Graphs**
  - np.meshgrid()
  - Working of meshgrid
  - Plot_surface

General Greetings and Questions [00:00 - 00:02]

Discuss summary of the previous class [00:02 - 00:07]

Did anyone try Practice Questions and Assignment for previous lecture?

- **Clarify doubts** regarding practice questions and Assignment

Today's Agenda [00:07 - 00:09]

- In Today's lecture, we'll dive into **Visualization**
- We'll cover **Matplotlib** library
- We'll see some **interesting things** we can do **using Matlplotlib** to visualize our data

---

Installing Matplotlib [00:09 - 00:11]

In [ ]:

```
# %pip install matplotlib
```

Importing Matplotlib [00:11 - 00:12]

- You should be able to import matplotlib after installing it
- However we don't need to import the entire library but just is submodule `pyplot`
- We'll import `matplotlib.pyplot` as its **alias name** **`plt`**

What is `pyplot` though ?

- **`pyplot` is a module for visualization inside `matplotlib` library**

In [1]:

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

Question: How many of you have set-up plt now?

Now, why do we need to use Matplotlib? [00:12 - 00:15]

- It helps us to **visualize (plot) our data.**
- Understanding dataset in a **pictorial format**
- It's **extensively used in Exploratory Data Analysis**
- And also to **present peformance results of our models**

We now understand the use of matplotlib

Lets try to draw some visualisations to understand this better

Suppose we want to draw a curve passing through 3 points:

- (0, 3)
- (1, 5)
- (2, 9)

How can we draw this curve using matplotlib ?

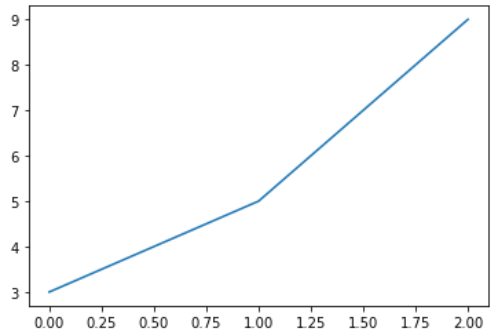
- By using the `plt.plot()` function
- We need to pass 2 arguments:
  1. x-axis values
  2. y-axis values
- For now lets create the lists and code it to see how it works

In [2]:

```
x_val = [0, 1, 2]
y_val = [3, 5, 9]
plt.plot(x_val, y_val)
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x2dcedf161c0>]
```

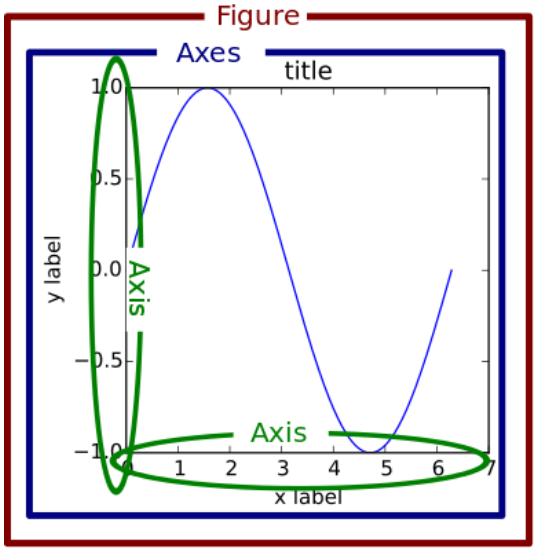


What can we observe from this plot ?

- `plt.plot()` automatically decided the scale of the plot
- It also prints the **type of object**, if you see:
  - **`matplotlib.lines.Line2D` is the type of the object**

Anatomy of Matplotlib

- There are **multiple underlying concepts and functions that you have used here without knowing**
- But later, when we move to advance plotting, it **might require us to know whats happening underneath**
-



So, lets explore anatomy of matplotlib a bit

What is a `Figure` ?

- It is the **overall window** or page that everything is drawn on.
- You can create multiple independent Figures.
- A Figure can have several other things in it, such as a suptitle, which is a centered title to the figure.
- You'll also find that you can add a legend and color bar, for example, to your Figure.
- To the figure you can add multiple Axes.

What are axes now ?

- It is the area on which the data is plotted with functions such as plot()
- It can have ticks, labels, etc. associated with it.
- So a figure can have multiple axes
- We will discuss more about this later in the lecture

What about the `Axis` now ?

- Each Axes has an x-axis and a y-axis
- They contain ticks, which have major and minor ticklines and ticklabels.
- There's also the axis labels, title, and legend to consider when you want to customize your axes

These are the major components of a matplotlib plot

As we move forward with this lecture you will find that matplotlib does a lot of the work underhood and requires us to just call its functions

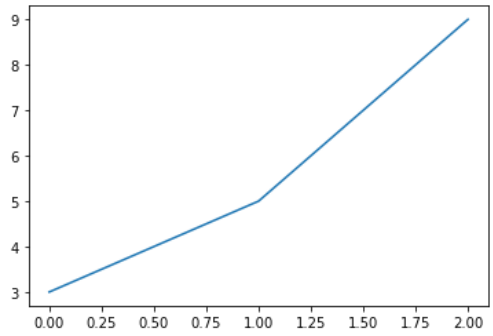
Now lets get back to the plot we drew

In [2]:

```
x_val = [0, 1, 2]
y_val = [3, 5, 9]
plt.plot(x_val, y_val)
```

Out[2]:

```
[<matplotlib.lines.Line2D at 0x7ff71827aca0>]
```



Notice from the plot we drew that x-axis values are same as indices of y-axis values

For eg:

- `3` in y_val is located at index `0`
- `5` in y_val is located at index `1` etc..

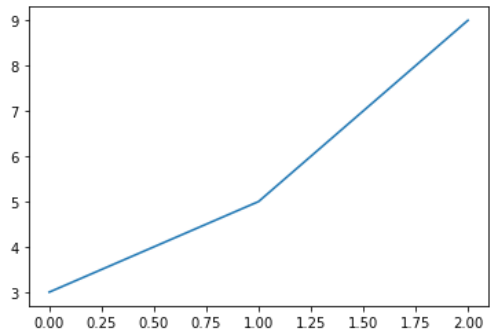
In such a case we can simply pass the y_val list only

In [3]:

```
plt.plot(y_val)
```

Out[3]:

```
[<matplotlib.lines.Line2D at 0x7ff6d803e340>]
```



What can we observe from this ?

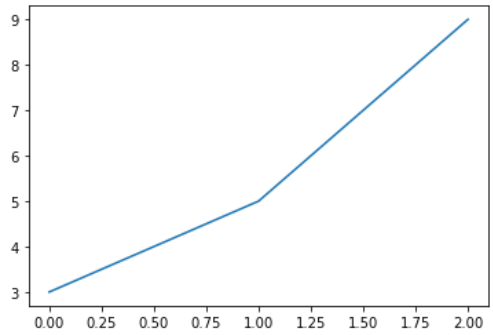
- The plot is same as the prev plot
- **Values we passed as a list** are **by default considered** as **y-values**
- By **default**, it has chosen corresponding **x-values as [0, 1, 2]**
- That's how `plot` works by default

And what if we don't want to print object type and just want the output figure in Jupyter Notebook,

- We can do this using the `plt.show()`

In [4]:

```
plt.plot([3, 5, 9])
plt.show()
```



Note: We should use plt.show() only when necessary. Why ?

- Multiple use within same script can lead to unpredictable results

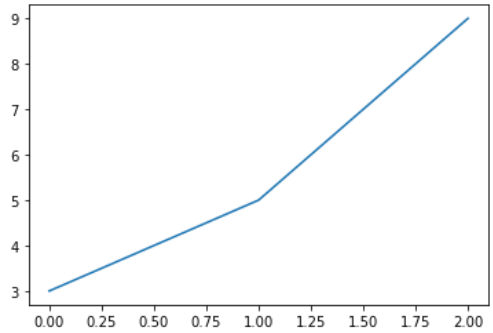
Now that we have seen how we can display a plot, what if we want to save it as an image in our local machine

How can we save plots ?

- Using the plt.savefig() command
- It takes the path to store the file as the input
- By default, the image extension used is .png

In [5]:

```
plt.plot(x_val, y_val)
plt.savefig("./figure")
```



Now, lets create a more complicated plot

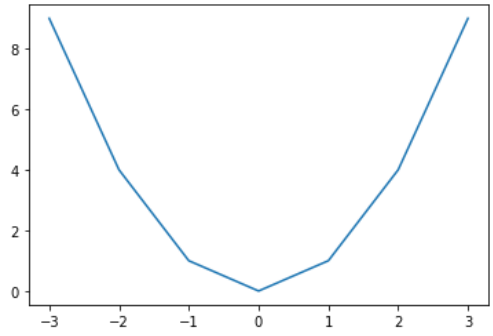
- Lets create a plot of $y = x^2$
- We know that it should come out as a parabola

How can we do this ?

- Again by creating lists of x_val and y_val

In [42]:

```
plt.plot([-3, -2, -1, 0, 1, 2, 3], [9, 4, 1, 0, 1, 4, 9])
plt.show()
```



What are the observations from this ?

- This curve closely resembles the parabolic shape
- But its not completely smooth

What can we do to improve this plot ?

- Increase no. of points to pass through it
- We can do so using numpy library
  - Recall how we can use it to create arrays

Lets code it to see how it works

In [4]:

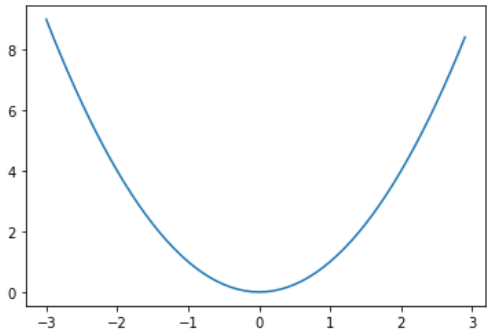
```
import numpy as np
```

Let's create some arrays to plot

- We will be using np.arange() for this
- Recall what this function does
  - Creates array with values in the given range
- We will use a step = 0.1 to create large no. of points

In [5]:

```
x = np.arange(-3, 3, step = 0.1)

# For each of these values, Let's say I want to plot its square

y = x ** 2

# Now let's plot the graph

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



- This is the curve for $y = x^2$
- It is much smoother and resembles parabola

Also observe the scale of the plot:

- y-axis: 2
- x-axis: 1

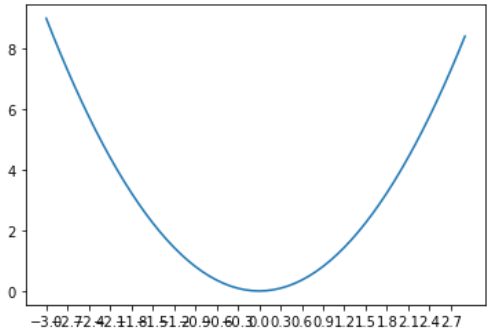
But our x-axis vals have a gap of 0.1 only (As set in linspace())

So how can we change the scale of an axis ?

- Using the xticks() or yticks() methods
- It takes as arguments:
  - List/array of axis vals
- For our plot we will set the scale of x-axis as 0.3
- Keeping the scale too small like 0.1 can lead to overlapping vals
- Lets see it in code now

In [8]:

```
x = np.arange(-3, 3, step = 0.1)

# For each of these values, Let's say I want to plot its square

y = x ** 2

# Now let's plot the graph

plt.plot(x, y)
plt.xticks(np.arange(-3, 3, step = 0.3))
plt.show()
```



Well we can't really see any val on x-axis

Why is this happening ?

- Because the ticks are overlapping with one another

What can we do to solve this problem ?

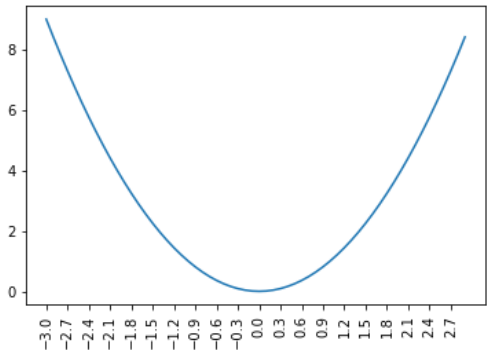
- Make the figsize bigger
- OR a simple method is to rotate the xticks by some angle

How can we rotate the xticks/yticks ?

- By using the `rotation` argument
- Lets see how it works

In [9]:

```
x = np.arange(-3, 3, step = 0.1)

# For each of these values, Let's say I want to plot its square

y = x ** 2

# Now let's plot the graph

plt.plot(x, y)
plt.xticks(np.arange(-3, 3, step = 0.3), rotation = 90)
plt.show()
```



Style and Labelling [00:25 - 00:57]

Now what if I want to show lables, titles, change styles, etc. on my plot?

- We can easily do that before we call `plt.show()`
- Because `plt.show()` gives the final display

How can we add the title ?

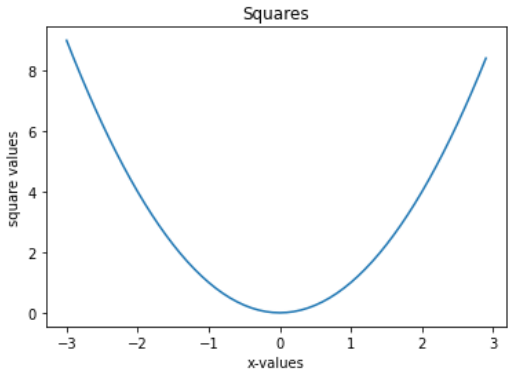
- Using plt.title() function

And what about the axis lables ?

- By using:
  - plt.xlabel(): x-axis
  - plt.ylabel(): y-axis

In [10]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y = x ** 2

# new
plt.title("Squares") # First, just type this much and run
plt.xlabel("x-values")
plt.ylabel("square values")

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



Now you can see the title and the axis labels

How are these values useful though ?

- Mention **meaning of values** on x and y axis in **lables**
- Mention the purpose of plot using **title**

Now what if we want to change the colour of the curve ?

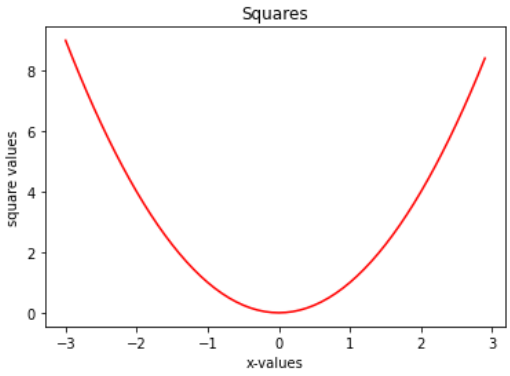
- `plt.plot()` contains an argument **color**
- It takes as argument a matplotlib color
- OR as string for some defined colours like:
  - black: `k` / `black`
  - red: `r` / `red` etc

- **But what all colours can we use ?**
  - Matplotlib provides a lot of colours
- Check docs for more colours

For now lets change our curve to red colour

In [48]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y = x ** 2

plt.title("Squares")
plt.xlabel("x-values")
plt.ylabel("square values")

plt.plot(x, y, color = 'r')  # new - List of colours is available in documentation - You can look it up later
plt.show()
```



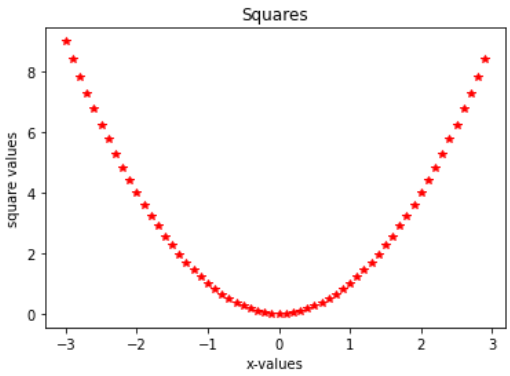
We can also change the shape of plot. How ?

- `plt.show()` has an argument **fmt** for format string
- By Changing Line to Points
  - `*` for start-shaped points,
  - `o` for circular points
  - `.` for smaller points
  - `v` for upside down triangle
  - `-` for single solid line
  - `--` for dashed line
  - ... and so on
- You can look it up in documentation later
- We can combine colour and shape we want in single string
- For eg: we want a red plot with star shape: String = "r*"

Lets change our plot to star shape

In [14]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y = x ** 2

plt.title("Squares")
plt.xlabel("x-values")
plt.ylabel("square values")

plt.plot(x, y, 'r*')  # new - *, o, ., v, -, -- for different styles
plt.show()
```



Now, lets say we only want the plot to include positive x_vals

How can we achieve this ?

- This requires changing the range of x-axis

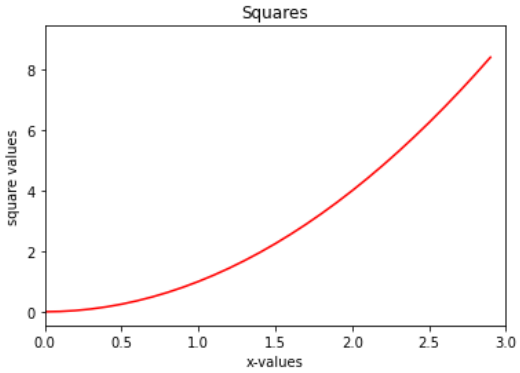
But how can we change the range of an axis in matplotlib ?

- So far we saw that **range of x and y axes were being decided by matplotlib automatically**
  - By default, Matplotlib will try to fit the graph in the best possible way
  - To change the range of these axes we can use:
    - `plt.xlim()` : x-axis
    - `plt.ylim()` : y-axis
  - These funcs take same 2 args:

    1. left: Starting point of range
    2. right: End point of range
  - Lets change range of our x-axis now to [0,3]

In [15]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y = x ** 2

plt.title("Squares")
plt.xlabel("x-values")
plt.ylabel("square values")

plt.xlim(left = 0, right = 3) # new

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

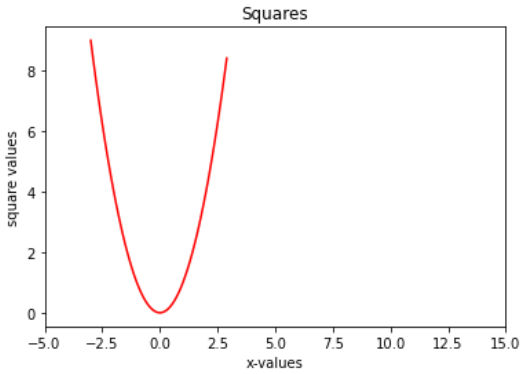


What can we observe from this ?

- Instead of starting at -3, x-axis **starts at 0**
- We can **pass any range we want** (**Check range with -5 to 15**)

In [16]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y = x ** 2

plt.title("Squares")
plt.xlabel("x-values")
plt.ylabel("square values")

plt.xlim(-5, 15) # new

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

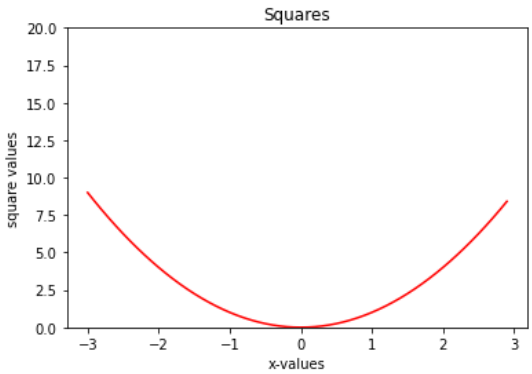


- Observe that in order to fit a wider range, it squeezed the curve

Similarly, we can change the `ylim`

In [12]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y = x ** 2

plt.title("Squares")
plt.xlabel("x-values")
plt.ylabel("square values")

plt.ylim(0, 20) # new

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



Quiz

In what order should you run the following commands in order to generate and display a plot?

1. plt.xlim(0, 10)
2. x = np.linspace(1, 10, 20)
3. plt.show()
4. plt.xlabel('X-Label')
5. plt.plot(x, label='Example Plot')

- 2-3-1-4-5
- 2-5-4-1-3
- 5-4-1-3-2
- 1-2-4-3-5

Ans: B

So far we have visualised a single plot to understand it

What if we want to compare it with some other plot like $y = x^3$?

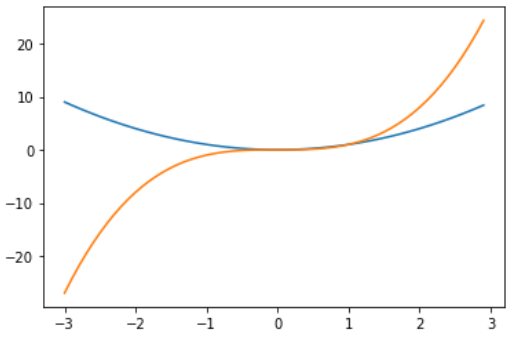
- To do so we will have to draw these plots in the same figure

But how can we do that ?

- By using multiple plt.plot() funcs before plt.show()
- Lets understand this through code

In [18]:

```
x = np.arange(-3, 3, step = 0.1)

y1 = x ** 2

y2 = x ** 3

plt.plot(x, y1)
plt.plot(x, y2)

plt.show()
```



What can we observe from this ?

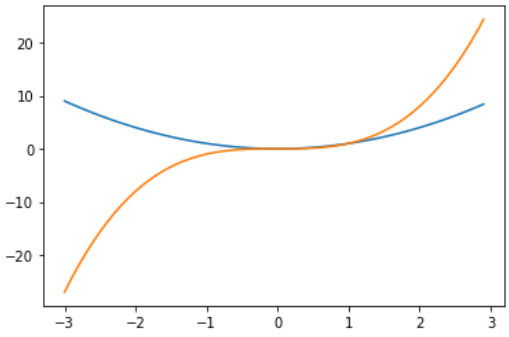
- Both the plots like in the same figure
- Matplotlib automatically created 2 plots with **different colors**

However how can we know which colour is of which plot ?

- plt.plot() has another argument **label** to do so
- We can simply set the label of each plot

In [49]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y1 = x ** 2

y2 = x ** 3

plt.plot(x, y1, label = "Squares")

plt.plot(x, y2, label = "Cubes")

plt.show()
```

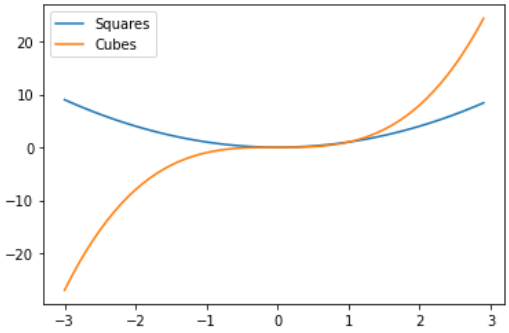


But why the legend has not been displayed?

- We **also need to write a command to display the legend**: `plt.legend()`

In [50]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y1 = x ** 2

y2 = x ** 3

plt.plot(x, y1, label = "Squares")

plt.plot(x, y2, label = "Cubes")
plt.legend()

plt.show()
```



Legends make more sense when we have multiple curves on the same graph

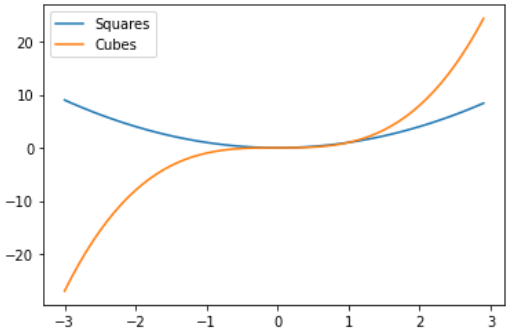
We can also pass these labels in plt.legend() as a list in the order plots are done

What does this mean ?

- We first draw the plot of $y = x^2$
- Then we draw plot of $y = x^3$
- So legends can be shown as `plt.legend(["Squares", "Cubes"])`

In [51]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y1 = x ** 2

y2 = x ** 3

plt.plot(x, y1)

plt.plot(x, y2)
plt.legend(["Squares", "Cubes"])

plt.show()
```



Question: Did everyone get this plot upto here?

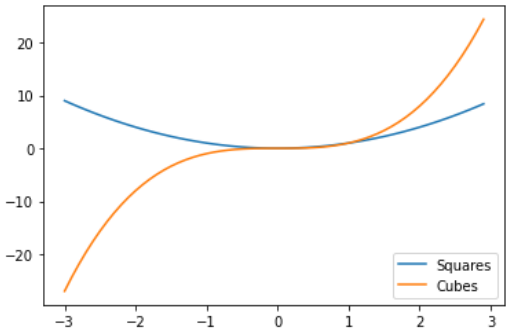
- [Resolve issues, if any]

But what if we want the legends to be in bottom-right corner ?

- Matplotlib automatically decides the best position for the legends
- But we can also change it using the `loc` parameter
- `loc` takes input as 1 of following strings:
  - upper center
  - upper left
  - upper right
  - lower right ... etc
- Lets see how it works

In [52]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y1 = x ** 2

y2 = x ** 3

plt.plot(x, y1)

plt.plot(x, y2)
plt.legend(["Squares", "Cubes"], loc = "lower right")

plt.show()
```



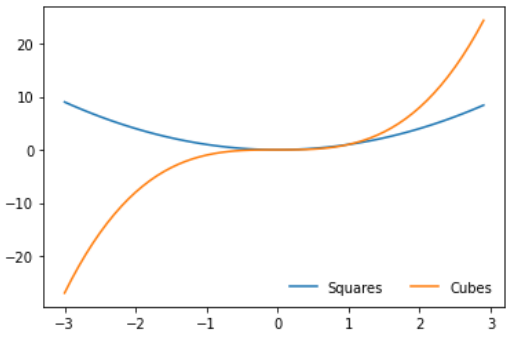
What if we want to add other stylings to legends ?

- Turns out we can do a lot more with the legends box
- For eg:
  - We can specify the number of rows/cols
    - Uses parameter `ncols` for this
    - The number of rows are then decided automaticallu
  - We can also decide if we want the box of legends to be displayed
    - Using the bool param `frameon`
  - More styling can be provided as well. You can refer the docs for further info

Lets code these 2 for now

In [23]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y1 = x ** 2

y2 = x ** 3

plt.plot(x, y1)

plt.plot(x, y2)
plt.legend(["Squares", "Cubes"], loc = "lower right", ncol = 2, frameon = False)

plt.show()
```



Now, notice that these plots are intersecting at some point

If we calculate it, the curves would intersect at points:

- x = 0
- x = 1

Lets say we want to highlight these points

How can we do that ?

- By adding text to these points
- Adding text "(0, 0)" and "(1, 1)" should be enough

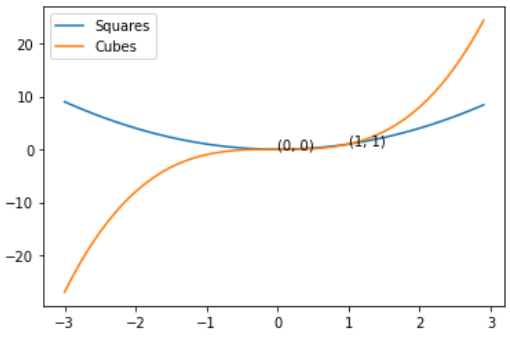
But how can we add text to points in a figure ?

- By using `plt.text()`
- Pass in the **x and y coordinates** over which we want text to appear
- Pass in the **text string**

Let's display the texts for intersection point of the 2 curves

In [24]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y1 = x ** 2

y2 = x ** 3

plt.plot(x, y1)

plt.plot(x, y2)
plt.legend(["Squares", "Cubes"])
plt.text(0, 0, "(0, 0)")
plt.text(1, 1, "(1, 1)")

plt.show()
```

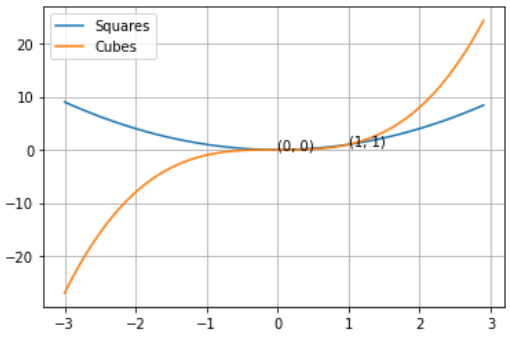


We can also show the grid as well in the background

- We can use **`plt.grid()`**
- We can also **pass in parameters inside `plt.grid()` to control its density, colour of grid lines, etc.**
- You can look it up later on how to customize the grid

In [25]:

```
# same code
x = np.arange(-3, 3, step = 0.1)

y1 = x ** 2

y2 = x ** 3

plt.plot(x, y1)

plt.plot(x, y2)
plt.legend(["Squares", "Cubes"])
plt.text(0, 0, "(0, 0)")
plt.text(1, 1, "(1, 1)")

plt.grid()  # new

plt.show()
```

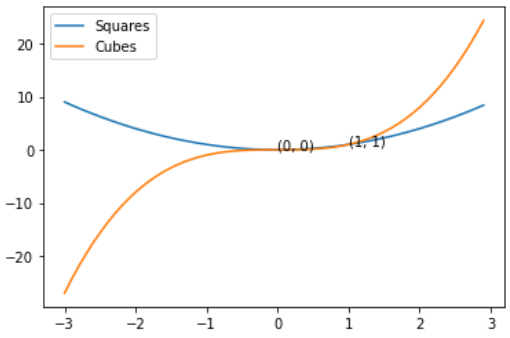


There's another short-hand notation for plotting multiple curves using same `plt.plot()` command

- Lets see how it works

In [26]:

```
plt.plot(x, y1, x, y2)

plt.legend(["Squares", "Cubes"])

plt.text(0, 0, "(0, 0)")
plt.text(1, 1, "(1, 1)")

plt.show()
```



Types of Graphs in Matplotlib [00:57 - 01:31]

So far we only studied line plots i.e plots that draw a line between 2 points

Now lets say you have 3 fruits:

- Fruit1 costs 10rs
- Fruit2 costs 20rs
- Fruit3 costs 5rs

How can we visualise this information now ?

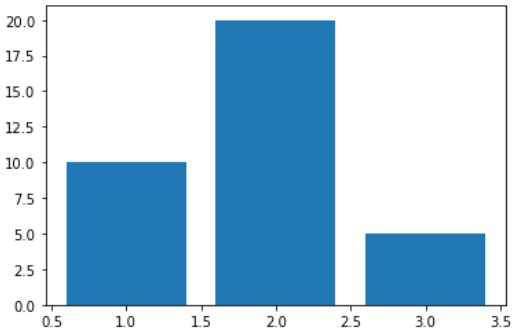
- This is where bar graph comes into picture

How can we draw a Bar Graph ?

- Using `plt.bar()`
- It takes:
  - x-axis values (1, 2, 3, for fruit1, 2, 3 in our case)
  - y-axis values: Heights of bars (10, 20, 5 in our case)
- Lets code it now

In [27]:

```
plt.bar([1, 2, 3], [10, 20, 5])
plt.show()
```



What can we infer from this plot ?

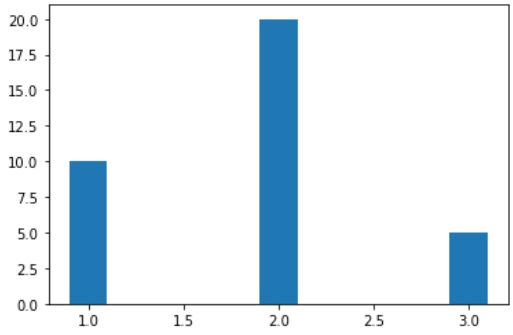
- We can see the diff in prices of the fruits
- The width of each bar is **1**
- Each bar is centered at its x-value

The bars are too wide though. Can we change their width ?

- By **default, width of bar is 1 unit**
- We can make the bars **wider or thinner by setting the `width` parameter**

In [28]:

```
# same code
plt.bar([1, 2, 3], [10, 20, 5], width=0.2) # new
plt.show()
```

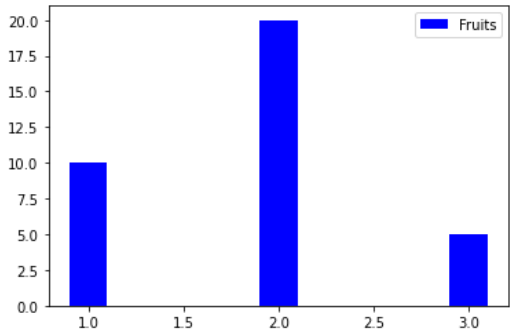


What about any additional styling to add to the bars ?

- We can **change colour of bars**
- We can **provide legends, etc.**

In [29]:

```
# same code
plt.bar([1, 2, 3], [10, 20, 5], width=0.2, color='b') # new
plt.legend(["Fruits"]) # new - Don't forget to put this when you mention a legend for graph
plt.show()
```



Lets say you went to another market and find that:

- Fruit1 costs 5rs
- Fruit2 costs 10rs
- Fruit3 costs 2.5rs

Now you want to visualise the diff in these prices from market 1

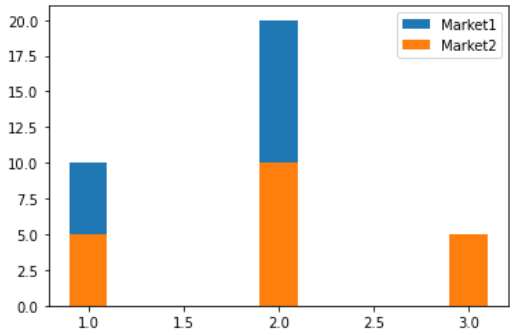
How can we do this ?

- By drawing multiple bar plots in the same figure
- Similar to how we drew multiple line plots in the same figure

Lets code it

In [30]:

```
# same code
plt.bar([1, 2, 3], [10, 20, 5], width=0.2, label='Market1')
plt.bar([1, 2, 3], [5, 10, 5], width=0.2, label='Market2') # new
plt.legend()
plt.show()
```



What can we observe from this ?

- The prices of fruit1 and fruit2 are double in market 2
- But we can't see price of fruit3 in market1

Why can't we see price of fruit3 from market1 ?

- Because of overlapping
- For `x=3` , the `market2` **bar** has **completely masked** the `market1` **bar**

How do we fix this problem?

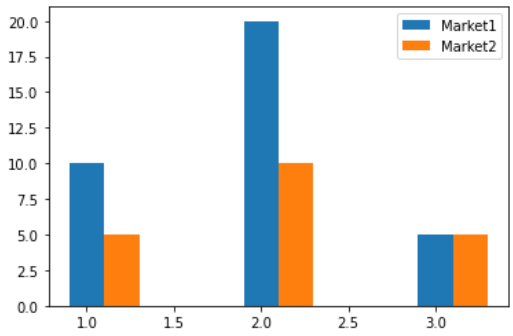
- We can **shift the x-values a little bit to left or right** for one of the bar graphs
- We can **pass [1.2, 2.2, 3.2] as x-values** for 2nd bar graph
- But its **so manual!!**

Is there a quicker way to do this?

- YES
- **Convert the x-values into a np array** and **apply shift**

In [31]:

```
# same code
plt.bar([1, 2, 3], [10, 20, 5], width=0.2, label='Market1')
plt.bar(np.array([1, 2, 3])+0.2, [5, 10, 5], width=0.2, label='Market2') # new
plt.legend()
plt.show()
```



This is how you can arrange the different bar graphs in same chart

Now lets look at some other usecase

Suppose we have data x = [1, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 2, 2]

Unique vals in x: 1, 2, 3

We want the visualise the frequency of each val in x

How can we do that ?

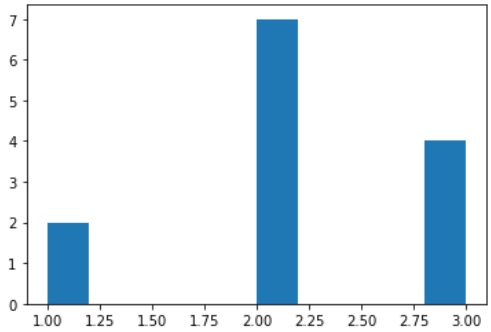
- Using histograms

What is a Histogram ?

- **Histograms are frequency charts**
- They tell us **how many times a value has occurred** in the data OR **how values are distributed**
- We can code it using `plt.hist()`
- We only need to pass x to the func and it will do the rest
- It returns 3 things:
  - count
  - bins
  - patches
  - We will discuss these later
- Lets code it now

In [32]:

```
vals = [1, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 2, 2]
plt.hist(vals)
plt.show()
```



What can we observe from this plot ?

- `1` appears two times
- `2` appears seven times
- `3` appears four times
- It has calculated the frequency of each value in data on its own
- It looks very similar to a bar chart

So what is the difference b/w Bar Chart and Histogram?

- In **Bar Graphs**, we were **manually passing the height of bars** for corresponding x-values
- In **Historgam**, we can just **pass in raw data**
- In histograms these rectangular columns are called **bins** or **buckets**
- It will **automatically determine the frequency of each value in data** and create the bar

Also did you notice, that the bins are not centered on `1`, `2`, `3` on x-axis?

- In **Bar Graph**, if you see, **bars are centred on their values on x-axis**
- But **NOT in Histogram, why??**
  - This is how histogram decides bins or buckets by default

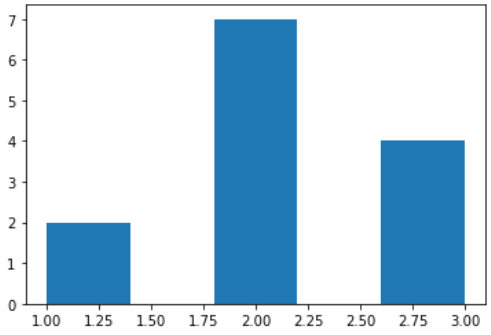
How can we change this behaviour ?

- Using the **bins** argument of `plt.hist()`
- It can take 3 types of inputs
- Lets take a look at them one by one

1. Integer
   - Specifies no. of bins in the plot

In [33]:

```
vals = [1, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 2, 2]
plt.hist(vals, bins = 5)
plt.show()
```



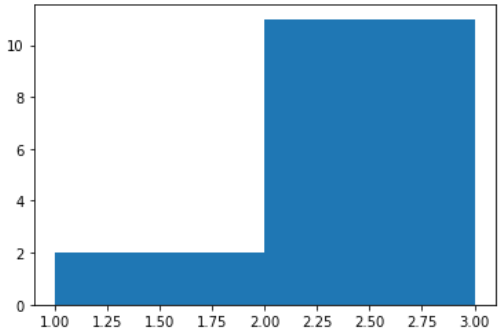
What can we observe from this ?

- The no. of bins obviuously remains 3
- But width and loc of these bins changes

What would happen if we set bins = 2 ?

In [34]:

```
vals = [1, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 2, 2]
plt.hist(vals, bins = 2)
plt.show()
```



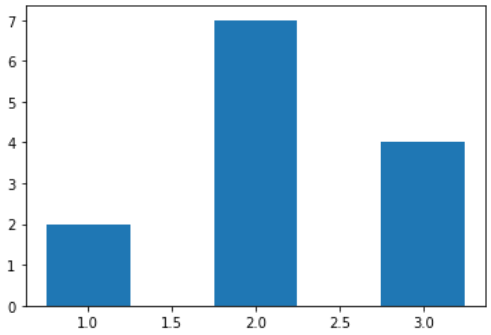
As you can see the 2nd and 3rd bin

The height of the merged bin = Max(Heights of bins being merged)

2. Sequence of numbers
   - Specifies the starting and ending points of bins
   - For eg: [1, 2, 3, 4, 5]
     - First bin: 1-2
     - Second bin: 2-3 etc
   - Lets see how it works

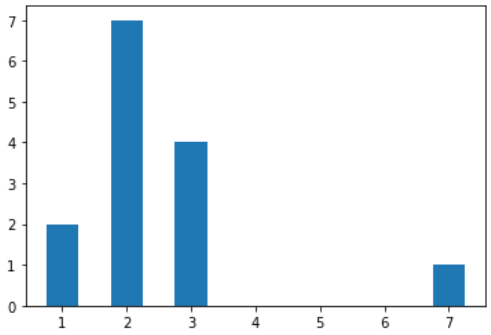
In [ ]:

```
vals = [1, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 2, 2]
plt.hist(vals, [0.75, 1.25, 1.75, 2.25, 2.75, 3.25])
plt.show()
```



In [ ]:

```
vals = [1, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 2, 2, 7]
plt.hist(vals, [0.75, 1.25, 1.75, 2.25, 2.75, 3.25, 6.75, 7.25])
plt.show()

```



The third method involves using matplotlib styled bins

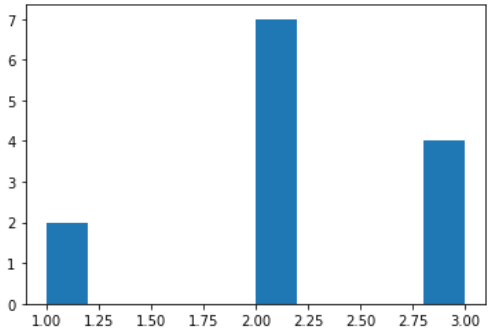
You can study more about them in docs

How can we save the bins ?

- We can **save the output from `plt.hist()` in variables**
- Let's check those values

In [ ]:

```
vals = [1, 2, 3, 1, 2, 3, 2, 3, 2, 3, 2, 2, 2]
count, bins, patches = plt.hist(vals)
plt.show()
```



In [ ]:

```
count
```

Out[39]:

```
array([2., 0., 0., 0., 0., 7., 0., 0., 0., 4.])
```

In [ ]:

```
bins
```

Out[40]:

```
array([1. , 1.2, 1.4, 1.6, 1.8, 2. , 2.2, 2.4, 2.6, 2.8, 3. ])
```

- We don't have to worry about `patches` at the moment
- It's not relevant

Now what do these `count` and `bins` mean?

If you see value of `bins`

- Histogram has created one bin from **1 to 1.2**
- Next bin from **1.2 to 1.4**
- Next one from **1.4 to 1.6** ... and so on upto **3.0**

By default, it has created 10 bins

- So, whatever values we pass in plt.hist()

- It divides the range of those values into 10 buckets

Now check the value of `count`

- It tells **how many values are there in each bin** (or bucket)
- So, we have **10 count values for 10 bins**

Question: What is the length of `count` ?

- 10
- No doubt there

Question: Now, What is the length of `bins` ?

- **10 + 1 = 11**
- **`bins` contain ranges**
- So, it has an **additional value - the last value of last bin range**

Did you understand it?

- There are **two occurences of value `1` in the first bin `1.0 - 1.2`**
- There is **nothing (no value) in the bin `1.2 - 1.4`**
- There are **seven occurences of value `2` in the bin `2.0 - 2.2`**

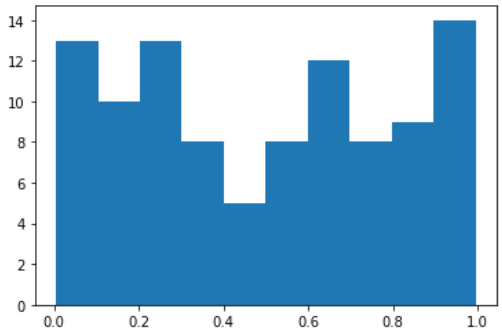
Now, Is the number of bins created by histogram fixed to 10?

- NO, We can change the **number of bins**
- We can also change **what those bins are**

Let's generate 100 random numbers b/w 0 and 1

In [54]:

```
vals = np.random.rand(100)
plt.hist(vals)
plt.show()
```

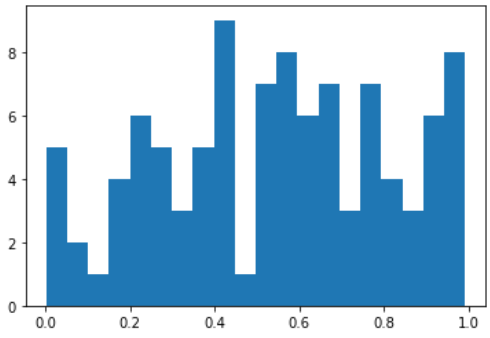


- As you can see, `plt.hist()` generates **10 bins by default**

Let's change the number of bins

In [55]:

```
# same code
vals = np.random.rand(100)
plt.hist(vals, 20) # Let's change the no. of bins to 20
plt.show()
```

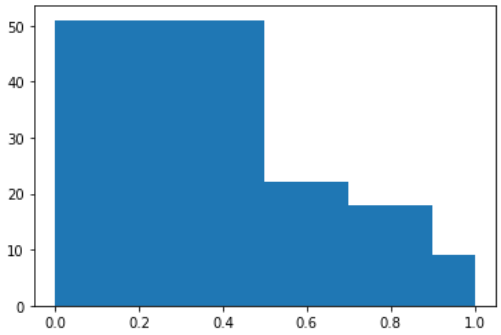


- Instead of specifying the no. of bins
- We can also **specify the exact bins we want**

Let's say we want the bins to be `[0-0.5, 0.5-0.7, 0.7-0.9, 0.9-1]`

In [59]:

```
vals = np.random.rand(100)
#plt.hist(vals, 20)
plt.hist(vals, [0,0.5,0.7,0.9,1])
plt.show()
```



Lets take another usecase now

Suppose we have a data of how many purchases are made from a store

Now you want to find the trend in the data

What does trend mean ?

- How the data changes i.e. does it increase/decrease during a certain interval

We can find these trends using visualisation

Whihch visualisation should we use ?

- Scatter plot

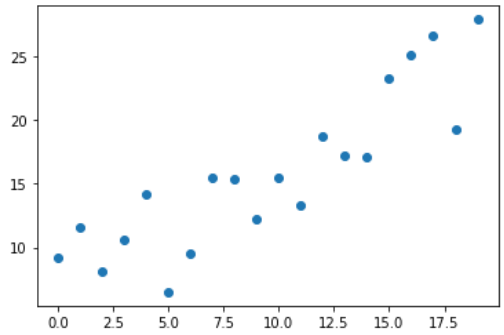
What is a Scatter Plot ?

- So, a scatter plot displays each (x, y) coordinate point separately
- The **points are scattered over the graph**

Let's generate 20 points and add some random numbers to them

In [ ]:

```
x = np.arange(0, 20)
y = np.linspace(1, 20, dtype=np.int32, num = 20) + np.random.rand(20)*10

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



What can you observe from this plot ?

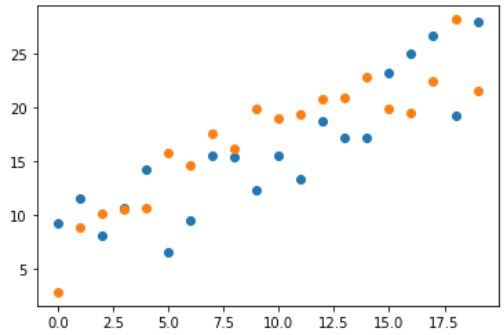
- The purchases increase and decreases in diff intervals
- But overall trend is **increasing**

Now lets say you have data of another shop and you want to compare it with the current one

To do so we would need to visualise the plots on the same figure

In [ ]:

```
x2 = np.arange(0, 20) # new
y2 = np.linspace(1, 20, dtype=np.int32, num = 20) + np.random.rand(20)*10 # new

plt.scatter(x, y)
plt.scatter(x2, y2)

plt.show()
```



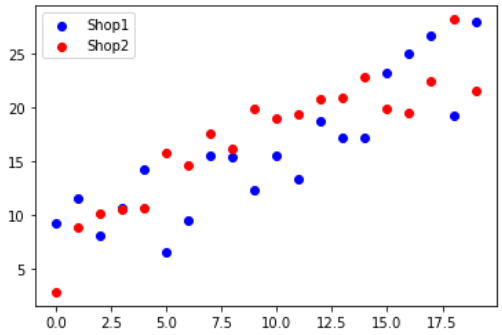
What can we infer from this plot ?

- Purchases in both shops can be compared in each point
- The **general trend is increasing** for both shops
- Also see that, **matplotlib has automatically chosen 2 different colours for the 2 Scatter Plots**

But we can **change colours manually as well** and **add legends**, just like we've been doing

In [ ]:

```
plt.scatter(x, y, color='b')
plt.scatter(x2, y2, color='r')
plt.legend(["Shop1", "Shop2"])

plt.show()
```

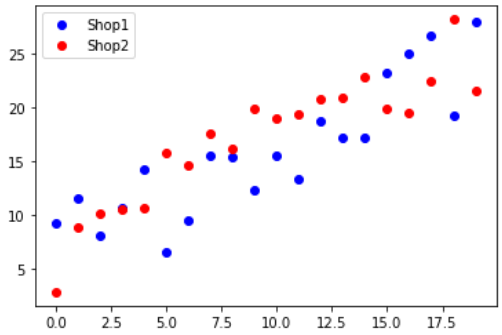


Now try to think if you can create a scatter plot using plt.plot()

- It seems we just have to change the styling of plots
- Lets see how it works

In [ ]:

```
plt.plot(x, y, 'bo')
plt.plot(x2, y2, 'ro')
plt.legend(["Shop1", "Shop2"])

plt.show()
```



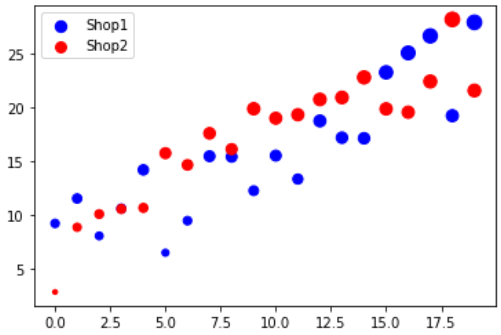
As you can see we got the same plot

So why do we need plt.scatter() ?

- In plt.scatter() we can handle properties of each point individually
- **Now what does this mean ?**
  - Lets say we want to make the size of every point proportional to its y-val
  - **Can we do this using plt.plot() ?**
    - No
  - But its possible through plt.scatter()
  - Lets see how

In [ ]:

```
plt.scatter(x, y, color = 'b', s = 4*abs(y))
plt.scatter(x2, y2, color = 'r', s = 4*abs(y2))
plt.legend(["Shop1", "Shop2"])

plt.show()
```



As you can see the size of points increases with y-val

But because plt.scatter() works on each point individually, it is slower

Hence in large datasets it is generally preferred to draw scatter plots using plt.plot()

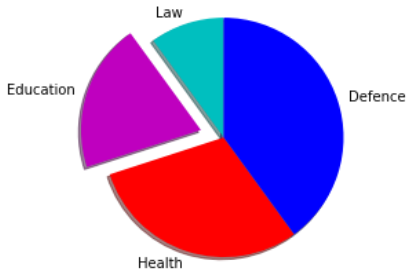
Pie charts

Finally Matplotlib also provide you with **Pie charts**

We won't go into too much details of it - We know what a Pie Chart is

In [ ]:

```
plt.pie([1,2,3,4],
        labels=['Law','Education','Health','Defence'],
        colors=['c','m','r','b'],
        startangle=90,
        shadow=True,
        explode=(0,0.2,0,0))
plt.show()
```



Question: [01:31 - 01:40]

Can you plot the graph for a function called sigmoid, which is defined as:

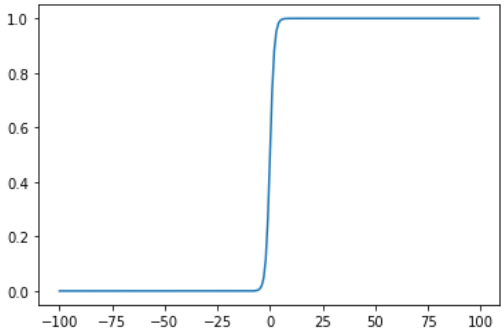
$$f(x) = \frac{1}{1+e^{-x}}$$

- **Hint**: Whenever you have to raise `e` to some power,
  - You can import `math`
  - And use `math.exp(5)`
  - This means e^5

Try doing this on your own

In [64]:

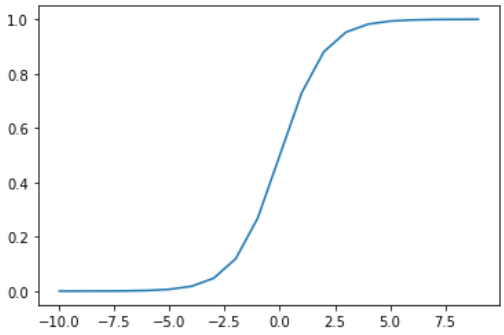
```
import math
import numpy as np

sigmoid = lambda x : 1/(1 + math.exp(-x))

# We don't know what curve looks like yet
# So, we'll start with a very broad range of x-values

x = np.arange(-100, 100)

# sigmoid function is not vectorized
# It only takes and returns a scaler value

# So, let's use List Comprehension

y = [sigmoid(i) for i in x]

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



- As you can see, **most of the curve** happens in the **central range**
- After that, it's just flat line
- So, **let's zoom in further in the central part of the curve**

In [ ]:

```
# same code
import math
import numpy as np

sigmoid = lambda x : 1/(1 + math.exp(-x))

x = np.arange(-10, 10) # new
y = [sigmoid(i) for i in x]

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



Observe a few things:

- **Mid-point** of curve is **(0, 0.5)**
- When **x increases, y approaches 1**
- When **x decreases, y approaches 0**

This is called Sigmoid or Logistic Function

- It will come handy later on

Subplots [01:40 - 01:52]

So far we have **shown only 1 figure** using `plt.show()`

We have plotted multiple curves, But on the same single graph

Now what if we want to plot multiple smaller figures at the same time?

- We will use **subplots**
- We'll **divide the figure into smaller plots**

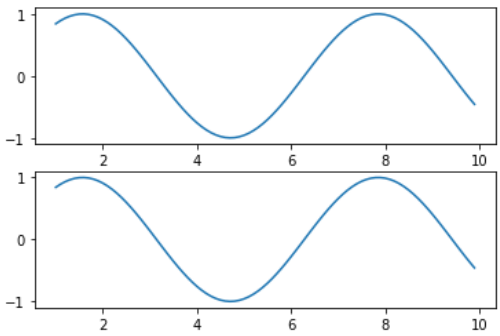
How can we achieve this ?

- Using `plt.subplots()` func
- It takes mainly 3 arguments:
  1. **No. of rows** we want to **divide our figure** into
  2. **No. of columns** we want to **divide our figure** into
- It returns 2 things:
  - Figure
  - Numpy Matrix of subplots

Let's see with an example

In [ ]:

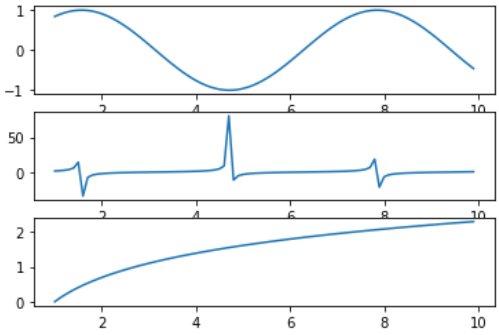
```
x = np.arange(1, 10, 0.1)
y = np.sin(x)

fig, ax = plt.subplots(2, 1)

ax[0].plot(x, y)

ax[1].plot(x, y)

plt.show()
```



Lets take another example

In [ ]:

```
x = np.arange(1, 10, 0.1)
y = np.sin(x)
y2 = np.tan(x)
y3 = np.log(x)

fig, ax = plt.subplots(3, 1)

ax[0].plot(x, y)

ax[1].plot(x, y2)

ax[2].plot(x, y3)

plt.show()
```



There is another method of creating subplots using the `plt.subplot()` func

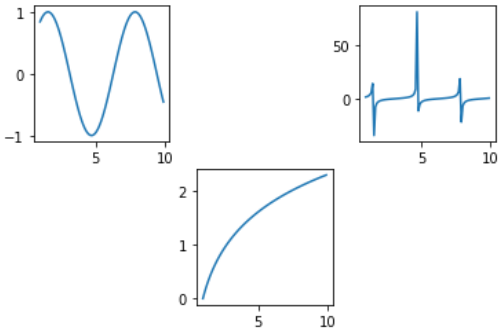
Step 1: Create a plt figure using plt.figure()

Step 2: Using `plt.subplot(row, col, pos)` create subplot from the figure

Step 3: Draw a curve on the subplot

In [ ]:

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

x = np.arange(1, 10, 0.1)
y = np.sin(x)
y2 = np.tan(x)
y3 = np.log(x)

plt.figure()

plt.subplot(2, 3, 1)
plt.plot(x, y)

plt.subplot(2, 3, 3)
plt.plot(x, y2)

plt.subplot(2, 3, 5)
plt.plot(x, y3)

plt.show()

```



We need to observe a few things here

1. The position/numbering starts from 1
2. It goes on row-wise from start of row to its finish
3. Empty subplots don't show any axes

If you look at the anaotmy of a plt figure, it also explains how there can be multiple axes within a figure (subplots)

Lets look at the pieces of code doing the same job but differently

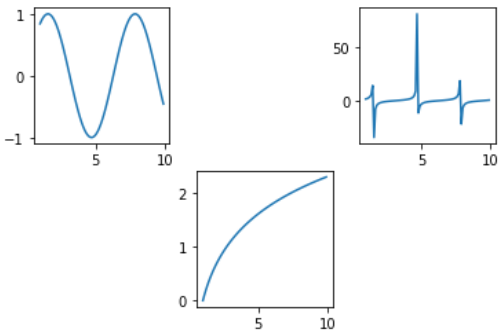
We saw this code earlier

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

x = np.arange(1, 10, 0.1)
y = np.sin(x)
y2 = np.tan(x)
y3 = np.log(x)

plt.figure()

plt.subplot(2, 3, 1)
plt.plot(x, y)

plt.subplot(2, 3, 3)
plt.plot(x, y2)

plt.subplot(2, 3, 5)
plt.plot(x, y3)

plt.show()
```

Lets write by using concept of `ax`

In [ ]:

```
fig = plt.figure()
ax231 = fig.add_subplot(231)
ax231.plot(x, y)
ax233 = fig.add_subplot(233)
ax233.plot(x, y2)
ax235 = fig.add_subplot(235)
ax235.plot(x, y3)
plt.show()
```



- We can of course use the clearner method which uses `plt`
- But if we are looking at some code, or doing some advance viz, we may require using `ax`

3-D Graphs [01:52 - 02:10]

- Matplotlib allows us to plot 3-D graphs

In [6]:

```
a = np.array([0, 1, 2, 3])
b = np.array([0, 1, 2])

a, b = np.meshgrid(a, b) # We'll use numpy's meshgrid
```

In [7]:

```
a
```

Out[7]:

```
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])
```

In [8]:

```
b
```

Out[8]:

```
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2]])
```

Notice here:

- `meshgrid()` allows you to **convert a 1-D array into 2-D**

How is `meshgrid()` working?

- It **takes dimensions of both arrays `a` and `b`**
  - Dimension of `a` is (4,)
  - Dimension of `b` is (3,)
- **Creates two 3×4 matrices**
- In **first matrix**, array of **`a` repeats 3 times** and gets **stacked vertically**
- In **second matrix**, array of **`b` repeats 4 times** and gets **stacked horizontally**

So, we get all possible combinations as coordinates

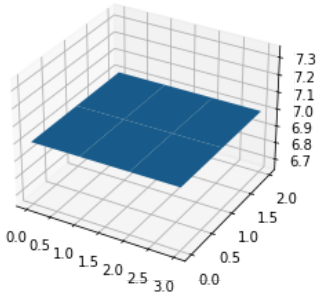
```
(0,0)  (1, 0)  (2, 0)  (3, 0)
(0,1)  (1, 1)  (2, 1)  (3, 1)
(0,2)  (1, 2)  (2, 2)  (3, 2)
```

- **Two 1-D arrays get converted into a meshgrid**

We can use the meshgrid to plot our 3-D graph

In [69]:

```
fig = plt.figure()  # Let's save plt.figure() in fig
ax = fig.add_subplot(projection='3d')
ax.plot_surface(a, b, np.array([[7]]) )  # 3rd argument is the elevation in 3-D we want
plt.show()
```



Observe that:

- **Range on the axes of our plane surface** is taken from **`a` and `b`**
  - **x-axis is represented by `a`** ---> 0 to 3

  - **y-axis is represented by `b`** ---> 0 to 2

- The **3rd dimension is the height from the floor we passed in**
  - **z-axis is represented by the elevation** we provided ---> 7

Question: Did everyone get this output?

Let's build a more complex 3-D plot

In [ ]:

```
a = np.arange(-1, 1, 0.005)  # I am going to use a very small step-size
b = a
a, b = np.meshgrid(a, b)


# Since `a` and `b` are some, this time we'll get a square matrix
```

In [ ]:

```
a.shape
```

Out[61]:

```
(400, 400)
```

In [ ]:

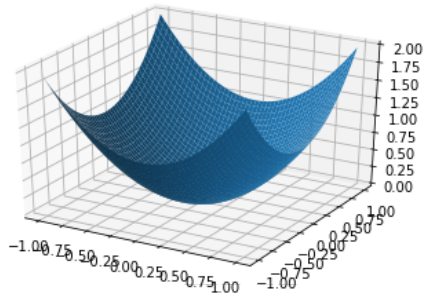
```
b.shape
```

Out[62]:

```
(400, 400)
```

In [ ]:

```
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(a, b, a**2 + b**2) # We'll NOT keep a constant z-axis elevation this time
plt.show()
```



Notice that:

- **Higher the values of `a` and `b`, higher will be z-axis value**
- At **`(a, b) = (0, 0)`**, `z` is **lowest at 0**
- The **dimensions of `a`, `b` and `a^2 + b^2` are all same**
  - 400×400
- That is why it is able to map all **(x, y, z) coordinates**

In [ ]:

```
(a**2).shape
```

Out[64]:

```
(400, 400)
```

In [ ]:

```
(a**2 + b**2).shape
```

Out[65]:

```
(400, 400)
```

This was all about Matplotlib for today

There are a lot of things to explore in Matplotlib

- You can explore other methods for performing different tasks on your own
- Or we'll come across many more things in future parts of the course

---

Final Q&A for Today's Lecture

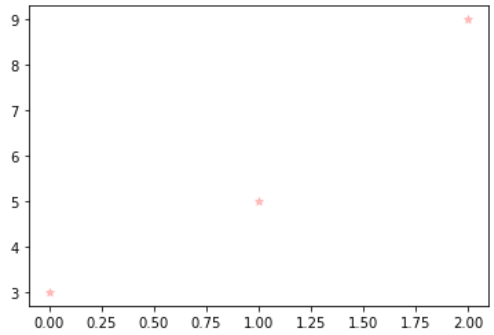
Hand-out seaborn as a supplementary (assign it as pre-read for next lecture)

Let students know about it too in the class to go through it

In [9]:

```
import matplotlib.pyplot as plt
x = [0, 1, 2]
y = [3, 5, 9]
plt.scatter(x,y, color = 'red', marker = '*', alpha = 0.2)
plt.text(x,y,label)

plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
C:\Users\SHELEN~1\AppData\Local\Temp/ipykernel_15340/1433141759.py in <module>
      3 y = [3, 5, 9]
      4 plt.scatter(x,y, color = 'red', marker = '*', alpha = 0.2)
----> 5 plt.text(x,y,label)
      6
      7 plt.show()

NameError: name 'label' is not defined
```



In [ ]:

```
1
```

In [ ]:

```
1
```

In [ ]:

```
1
```

In [ ]:

```
1
```

In [ ]:

```
1
```