**Matplotlib**

**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

**Installing Matplotlib**

# %pip install matplotlib

**Importing Matplotlib**

* 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**

import matplotlib.pyplot as plt

**Now, why do we need to use Matplotlib?**

* 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**

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 ?**

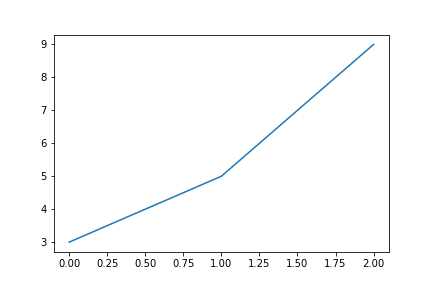
* By using the plt.plot() function
* We need to pass 2 arguments:
  1. x-axis values
  2. y-axis values
* For now lets create the lists and code it to see how it works

x\_val = [0, 1, 2]

y\_val = [3, 5, 9]

plt.plot(x\_val, y\_val)

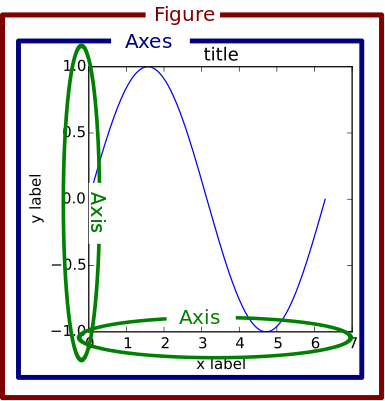
[<matplotlib.lines.Line2D at 0x7f38e56a9710>]



**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**

* 

**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

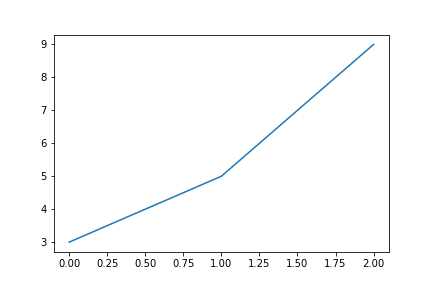
Now lets get back to the plot we drew

x\_val = [0, 1, 2]

y\_val = [3, 5, 9]

plt.plot(x\_val, y\_val)

[<matplotlib.lines.Line2D at 0x7f38e5195790>]



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

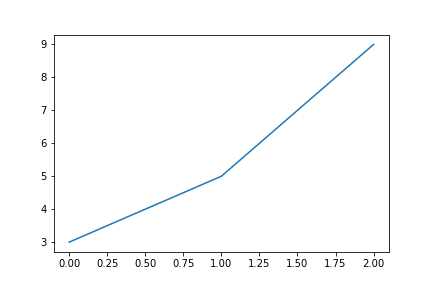
For eg:

* 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

plt.plot(y\_val)

[<matplotlib.lines.Line2D at 0x7f38e511c090>]



**What can we observe from this ?**

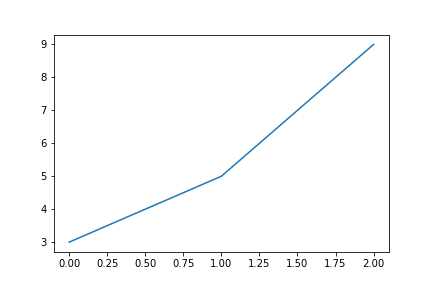
* The plot is same as the prev plot
* **Values we passed as a list** are **by default considered** as **y-values**
* By **default**, it has chosen corresponding **x-values as [0, 1, 2]**

**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()

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

plt.show()



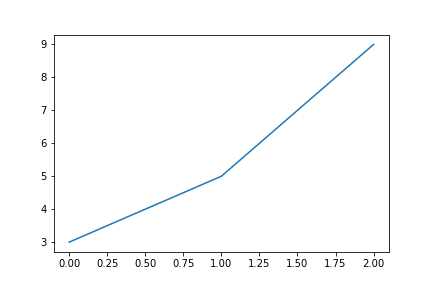
**Note: We should use plt.show() only when necessary. Why ?**

* Multiple use within same script can lead to unpredictable results

**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

plt.plot(x\_val, y\_val)



**Now, lets create a more complicated plot**

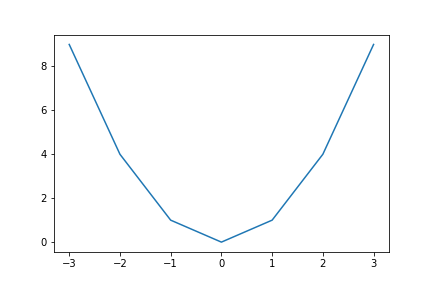
* Lets create a plot of y=x2y=x2
* We know that it should come out as a parabola

**How can we do this ?**

* Again by creating lists of x\_val and y\_val

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

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

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

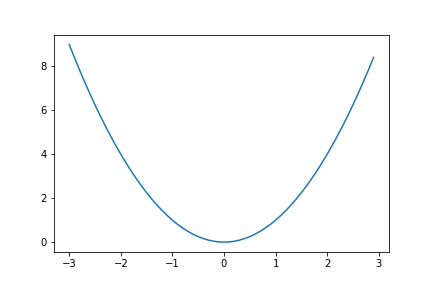
# For each of these values, Let's say I want to plot its square

y = x \*\* 2

# Now let's plot the graph

plt.plot(x, y)

plt.show()



* This is the curve for y=x2y=x2
* 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

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

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

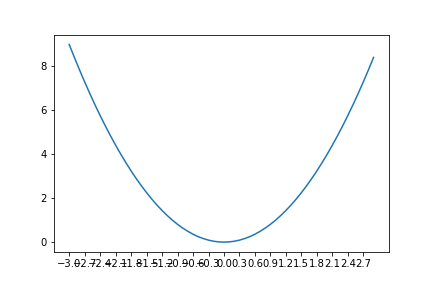
y = x \*\* 2

# 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

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

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

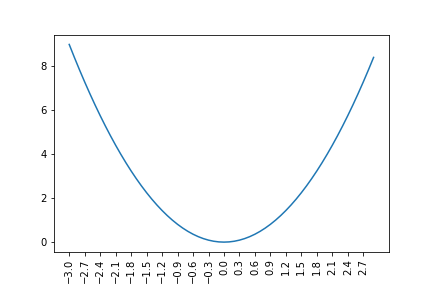
y = x \*\* 2

# 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**

**Now what if we 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

# same code

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

y = x \*\* 2

# new

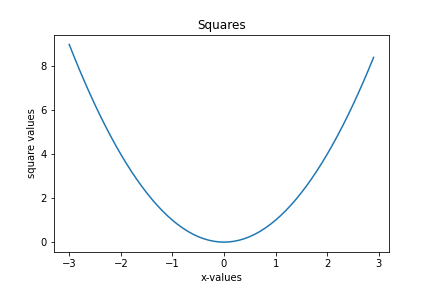
plt.title("Squares") # First, just type this much and run

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

# same code

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

y = x \*\* 2

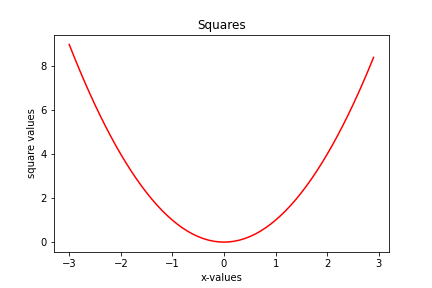
plt.title("Squares")

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

# same code

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

y = x \*\* 2

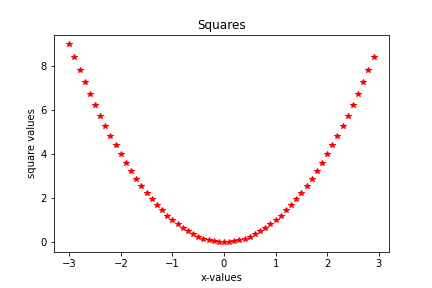
plt.title("Squares")

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

# same code

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

y = x \*\* 2

plt.title("Squares")

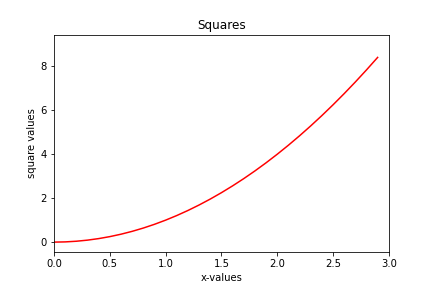
plt.xlabel("x-values")

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**

# same code

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

y = x \*\* 2

plt.title("Squares")

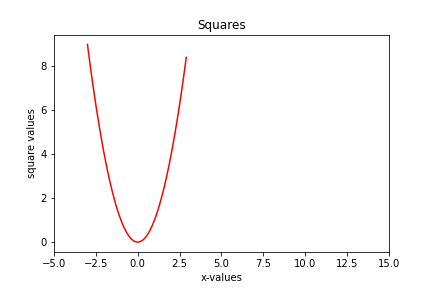
plt.xlabel("x-values")

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**

# same code

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

y = x \*\* 2

plt.title("Squares")

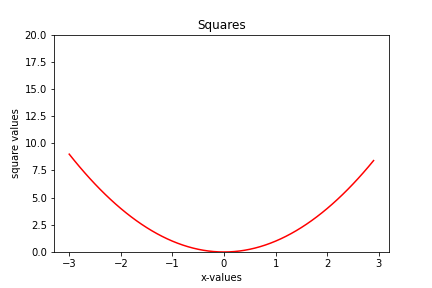
plt.xlabel("x-values")

plt.ylabel("square values")

plt.ylim(0, 20) # new

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

plt.show()



**What if we want to compare it with some other plot like**y=x3y=x3**?**

* 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()

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

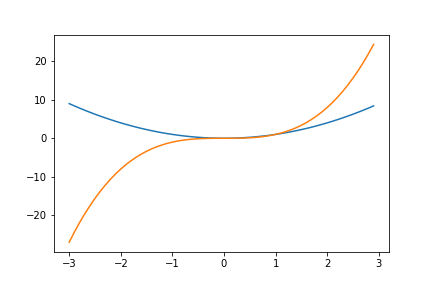
y1 = x \*\* 2

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

# same code

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

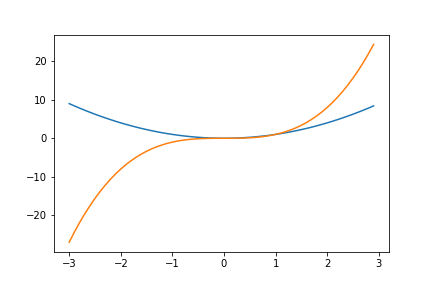
y1 = x \*\* 2

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()

# same code

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

y1 = x \*\* 2

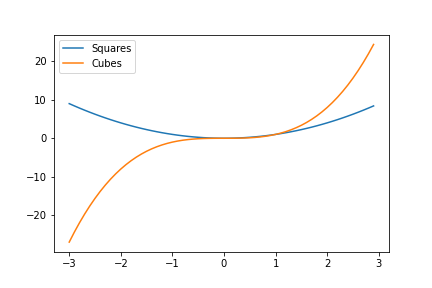
y2 = x \*\* 3

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=x2y=x2
* Then we draw plot of y=x3y=x3
* So legends can be shown as plt.legend(["Squares", "Cubes"])

# same code

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

y1 = x \*\* 2

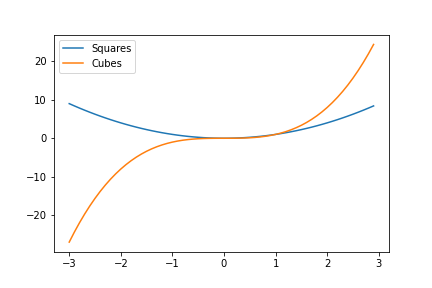
y2 = x \*\* 3

plt.plot(x, y1)

plt.plot(x, y2)

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

plt.show()



**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

# same code

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

y1 = x \*\* 2

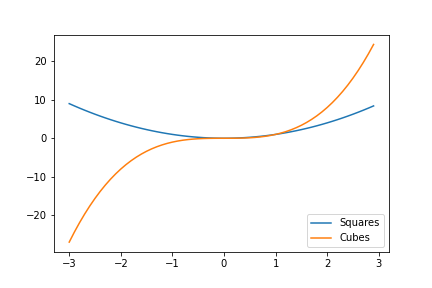
y2 = x \*\* 3

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

# same code

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

y1 = x \*\* 2

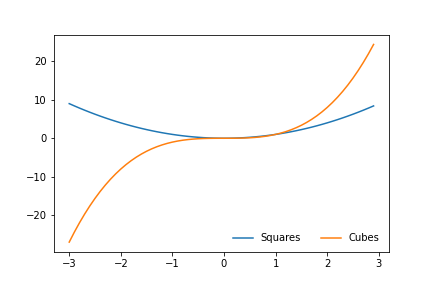
y2 = x \*\* 3

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

# same code

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

y1 = x \*\* 2

y2 = x \*\* 3

plt.plot(x, y1)

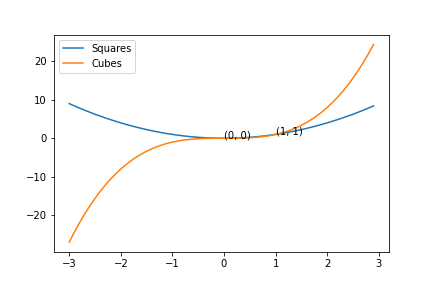
plt.plot(x, y2)

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

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

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

plt.show()



**We can 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

# same code

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

y1 = x \*\* 2

y2 = x \*\* 3

plt.plot(x, y1)

plt.plot(x, y2)

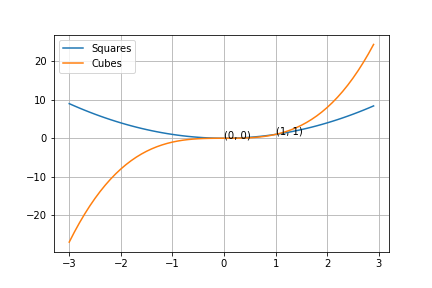
plt.legend(["Squares", "Cubes"])

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

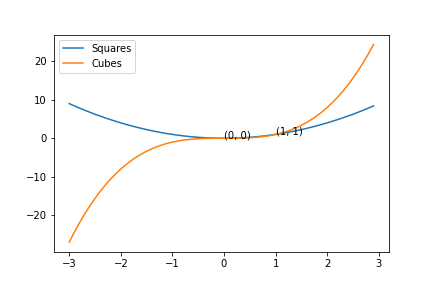
plt.plot(x, y1, x, y2)

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

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

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

plt.show()



**Types of Graphs in Matplotlib**

Now lets say you have 3 fruits:

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

**How can we visualise this information now ?**

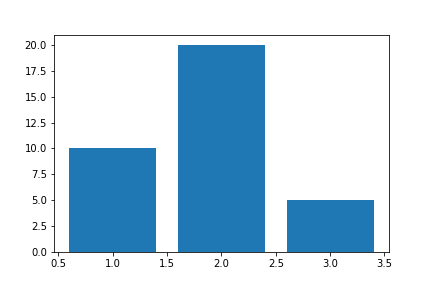
* This is where bar graph comes into picture

**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

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

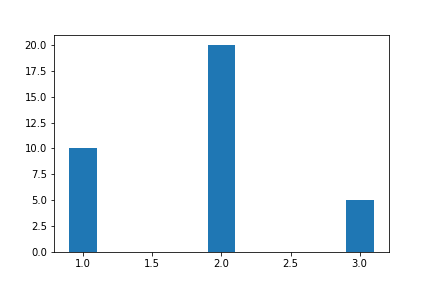
**The bars are too wide though. Can we change their width ?**

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

# 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 ?**

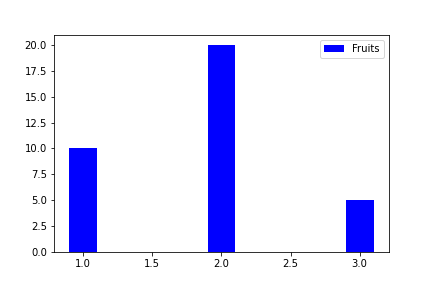
* We can **change colour of bars**
* We can **provide legends, etc.**

# 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

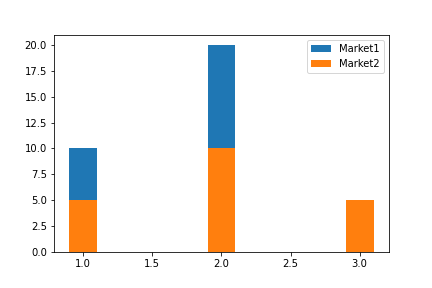
# same code

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**

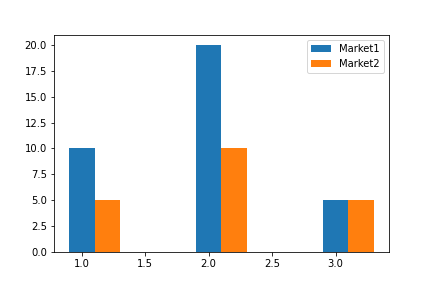
# same code

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

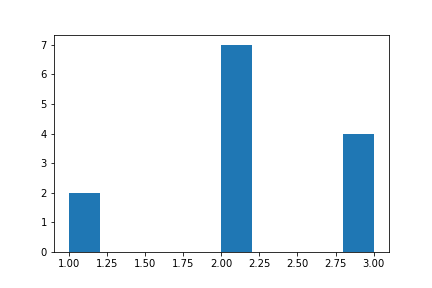
**What is a Histogram ?**

* **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

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 ?**

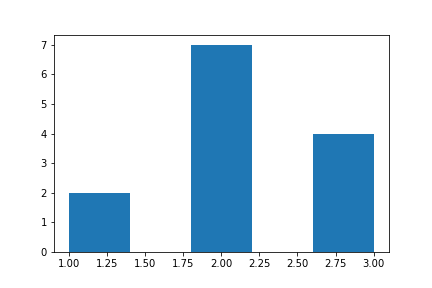
* Using the **bins** argument of plt.hist()
* It can take 3 types of inputs
* Lets take a look at them one by one

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

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 ?**

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)

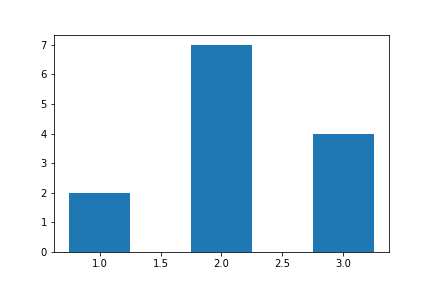
1. Sequence of numbers

* 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

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])

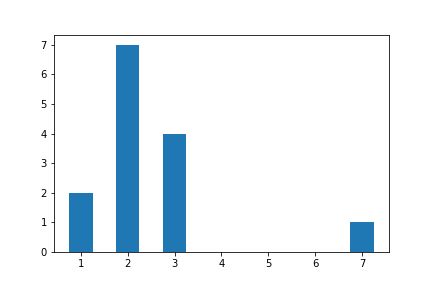
plt.show()



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

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**

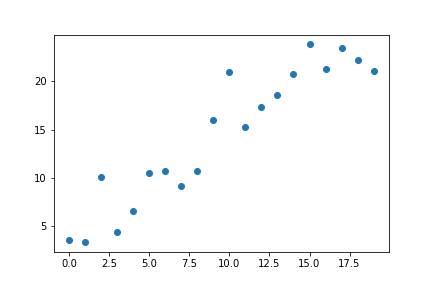
**Let’s generate 20 points and add some random numbers to them**

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

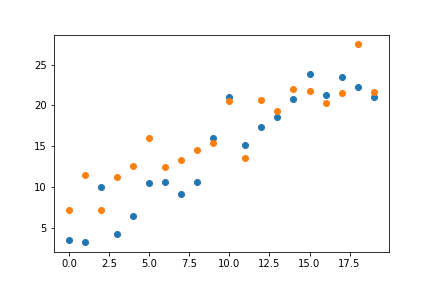
x2 = np.arange(0, 20) # new

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**

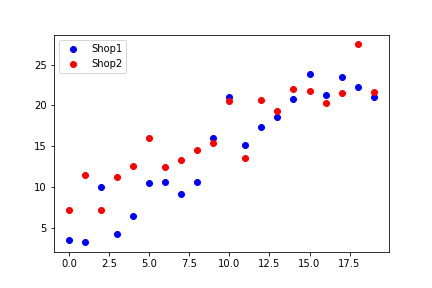
But we can **change colours manually as well** and **add legends**, just like we’ve been doing

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()**

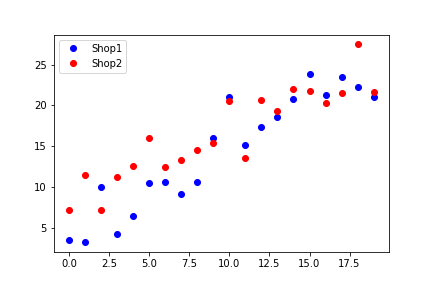
* It seems we just have to change the styling of plots
* Lets see how it works

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() ?**

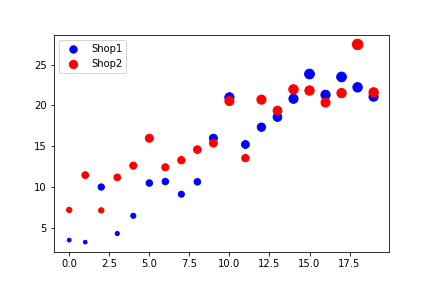
* In plt.scatter() we can handle properties of each point individually
* **Now what does this mean ?**
  + Lets say we want to make the size of every point proportional to its y-val
  + **Can we do this using plt.plot() ?**
    - No
  + But its possible through plt.scatter()
  + Lets see how

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

plt.pie([1,2,3,4],

labels=['Law','Education','Health','Defence'],

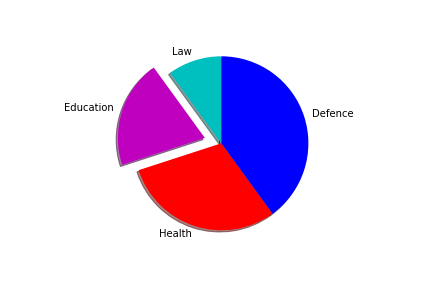
colors=['c','m','r','b'],

startangle=90,

shadow=True,

explode=(0,0.2,0,0))

plt.show()



**Subplots**

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:
  1. Figure
  2. Numpy Matrix of subplots

Let’s see with an example

x = np.arange(1, 10, 0.1)

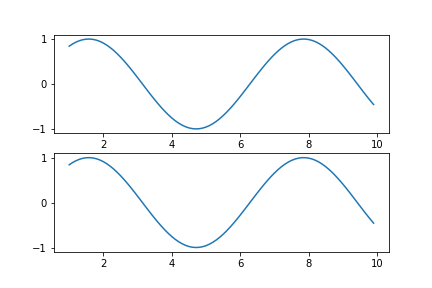
y = np.sin(x)

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

ax[0].plot(x, y)

ax[1].plot(x, y)

plt.show()



Lets take another example

x = np.arange(1, 10, 0.1)

y = np.sin(x)

y2 = np.tan(x)

y3 = np.log(x)

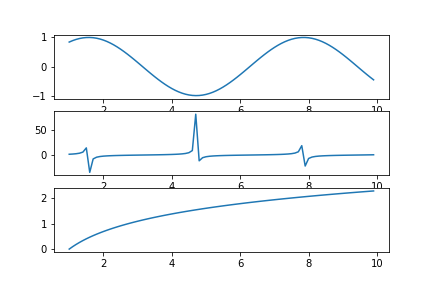
fig, ax = plt.subplots(3, 1)

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

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)

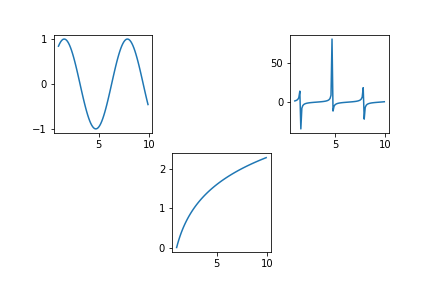
plt.subplot(2, 3, 3)

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

fig = plt.figure()

ax231 = fig.add\_subplot(231)

ax231.plot(x, y)

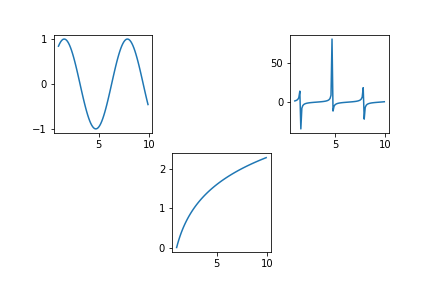
ax233 = fig.add\_subplot(233)

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**

* Matplotlib allows us to plot 3-D graphs

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

a

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

[0, 1, 2, 3],

[0, 1, 2, 3]])

b

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**3×43×4**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**

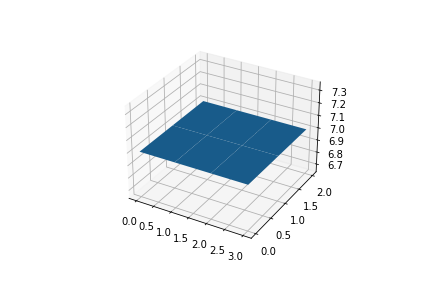
**We can use the meshgrid to plot our 3-D graph**

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

**Let’s build a more complex 3-D plot**

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

a.shape

(400, 400)

b.shape

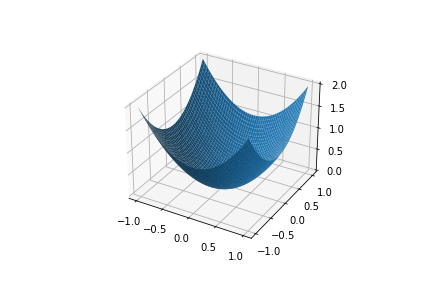
(400, 400)

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**
  + 400×400400×400
* That is why it is able to map all **(x, y, z) coordinates**

(a\*\*2).shape

(400, 400)

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

(400, 400)

**This is 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

Last changed by

**Scaler Topics**Scaler Topics

 220

**Seaborn**

**Content**

* **Introduction**
* **Installing and Importing**
* **Loading Dataset using seaborn**
* **Plots using Seaborn**
  + Histogram
  + KDE Plot
  + Scatterplot
  + Joint Plot
  + Pair Plot
* **Categorical Plots**
  + Count Plot
  + Box Plot
  + Voilin Plot
* **Finding correlations among attributes**
  + corr()
  + Heat Map
* **Choosing right visualization for a given purpose**
* **Use Case: Visualizing Tips Dataset**
* **Challenge: Visualize Titanic Dataset**

**Introduction**

* **Seaborn** is also a Python **Data Visualisation Library**
* Just **like Matplotlib**
* Infact, it is **built on top of matplotlib**
* It uses **Pandas DataFrame** to work with datasets

**Installing Seaborn**

**You must know this by now**

Using !pip install command

!pip install seaborn

Requirement already satisfied: seaborn in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (0.11.2)

Requirement already satisfied: pandas>=0.23 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from seaborn) (1.3.4)

Requirement already satisfied: numpy>=1.15 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from seaborn) (1.21.2)

Requirement already satisfied: scipy>=1.0 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from seaborn) (1.7.1)

Requirement already satisfied: matplotlib>=2.2 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from seaborn) (3.5.0)

Requirement already satisfied: pillow>=6.2.0 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from matplotlib>=2.2->seaborn) (8.4.0)

Requirement already satisfied: packaging>=20.0 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from matplotlib>=2.2->seaborn) (21.3)

Requirement already satisfied: pyparsing>=2.2.1 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from matplotlib>=2.2->seaborn) (3.0.4)

Requirement already satisfied: cycler>=0.10 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from matplotlib>=2.2->seaborn) (0.11.0)

Requirement already satisfied: fonttools>=4.22.0 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from matplotlib>=2.2->seaborn) (4.25.0)

Requirement already satisfied: python-dateutil>=2.7 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from matplotlib>=2.2->seaborn) (2.8.2)

Requirement already satisfied: kiwisolver>=1.0.1 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from matplotlib>=2.2->seaborn) (1.3.1)

Requirement already satisfied: pytz>=2017.3 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from pandas>=0.23->seaborn) (2021.3)

Requirement already satisfied: six>=1.5 in /home/aryanj/anaconda3/envs/tf-gpu/lib/python3.9/site-packages (from python-dateutil>=2.7->matplotlib>=2.2->seaborn) (1.16.0)

**Importing Seaborn**

* You should be able to import Seaborn after installing it
* We’ll import seaborn as its **alias name sns**

import seaborn as sns

import numpy as np

import matplotlib.pyplot as plt

**Loading Dataset using seaborn**

* Seaborn contains some **datasets in-built within the library itself**
* We can use its **load\_dataset()** function to **read the data**
* Works just like Pandas’s read\_csv()

**Have you heard of the Iris Dataset?**

* It’s like the **“Hello World” of datasets**
* Mostly used by **beginners** as a practice
* It’s the dataset about **3 species of Iris flower**
* Feel free to read more about the dataset here:

<https://en.wikipedia.org/wiki/Iris_flower_data_set>

**Let’s load and read Iris Dataset using seaborn**

iris = sns.load\_dataset('iris')

**Now, Let’s check what all is there in the dataset**

type(iris)

pandas.core.frame.DataFrame

iris.head()

|  | **sepal\_length** | **sepal\_width** | **petal\_length** | **petal\_width** | **species** |
| --- | --- | --- | --- | --- | --- |
| **0** | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| **1** | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| **2** | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| **3** | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| **4** | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

**As you can see:**

* The iris dataset is **loaded as a Pandas DataFrame**
* The iris dataset has **5 columns (attributes)**
  1. sepal\_length
  2. sepal\_width
  3. petal\_length
  4. petal\_width
  5. species

**We’ll explore these 5 variables (features) one by one**

* The last attribute **species** tells us which **species** the flower is **based on those measurements**

**Let’s check it out in a bit more detail**

iris['species'].unique()

array(['setosa', 'versicolor', 'virginica'], dtype=object)

**So, the flowers in dataset can belong to one of the 3 categories**

1. setosa
2. versicolor
3. virginica

* However, we will not go into the details of the dataset

**The purpose of this lecture is to see the functionality of seaborn**

* Feel free to explore the iris dataset more on your own

**Let’s plot our first graph using seaborn for the variable petal\_length**

iris['petal\_length']

0 1.4

1 1.4

2 1.3

3 1.5

4 1.4

...

145 5.2

146 5.0

147 5.2

148 5.4

149 5.1

Name: petal\_length, Length: 150, dtype: float64

**Histogram**

* We want to check the **distribution of this variable petal\_length**
* So, we use Histogram
* For **Scatter Plot**, we need **2 variables** - one on x-axis and one on y-axis
* For **Bar Chart**, we need a **categorical variable**
  + Like if we want to count how many employees in a company have Bachelors, Masters or PhD.

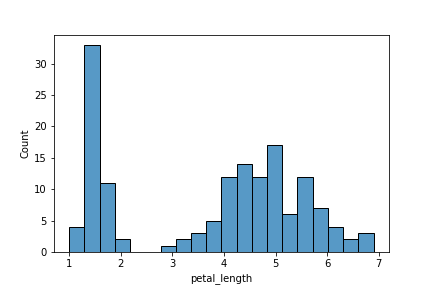
**Remember histogram from the matplotlib lecture?**

* We can use seaborn’s histplot() to plot a histogram
* We can also set the number of bins we want in our plot

**Let’s plot the distribution for the petal\_length column**

sns.histplot(iris['petal\_length'], bins= 20)

plt.show()



**As you can see here**

* We got a **histogram for petal\_length attribute**
* **More than 30 flowers** have petal\_length **between approx 1.25 and 1.5**
* **No flowers** have petal\_length between approx **2.25 and 2.75**
* … and so on. You can make similar other observations

See how neatly each bar is separated from the other

* Like we mentioned seaborn is built on top of matplotlib
* It has **some features** (not all) that are an **enhancement** over matplotlib.pyplot

**Now, let’s check out another plot using seaborn**

**Kernel Density Estimate (KDE) Plot**

* A KDE plot is a method for visualizing the distributions
* Just like histogram
* But instead of bars, KDE represents data using a **continuous probability density curve**
* You can check documentation for more details

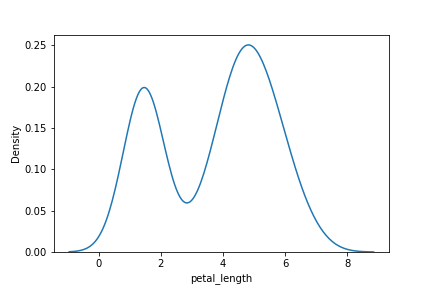
**Now, Why do we even need KDE plots?**

* Compared to histogram, KDE produces a plot which is **less cluttered** and **more interpretable**
* Especially when drawing multiple distributions.

**Let’s plot KDE using seaborn's kdeplot**

sns.kdeplot(iris['petal\_length'])

plt.show()



**As you can see**

* We got a **curve** instead of bars
* **y-axis** has the **probabilities** instead of actual count

**It gives the same information as histogram, but in a smoother way**

* Probability of flowers having petal\_length between approx 1.25 and 1.5 is high (~ 0.2)
* Probability of flowers having petal\_length between approx 2.25 and 2.75 is low (~ 0.05)

**Scatterplot**

**Remember scatterplot from Matplotlib?**

**Scatter Plot is mostly used to visualize relationship b/w 2 variables (Bi-Variate Analysis)**

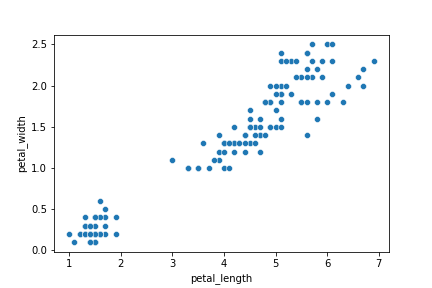
* A scatter plot displays each (x, y) coordinate point separately
* The points are scattered over the graph

**Let’s plot scatterplot now using seaborn**

* We will take **petal\_length as x-coordinates** and **petal\_width as y-coordinates**
* These (x, y) pairs of points will be plotted as a scatter plot

sns.scatterplot(x= iris['petal\_length'], y = iris['petal\_width'])

plt.show()



**Did you notice?**

* We did not have to specifically mention plt.xlabel() or plt.ylabel()
* **Seaborn automatically labelled axes for us**, unlike matplotlib.

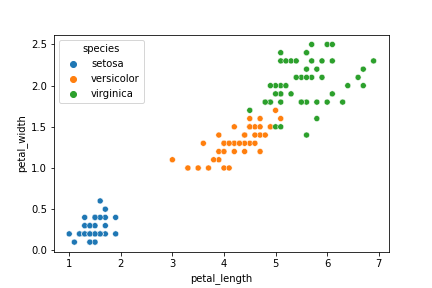
**There’s one more interesting thing we can do with seaborn’s scatterplot**

* We can visualize different species in different colours

**Let’s see how we can do it**

sns.scatterplot(x ='petal\_length', y ='petal\_width' , data= iris, hue='species')

plt.show()



* So, we provide our iris dataset as **input data parameter**
* We **set the hue** to species column

**Each different species is plotted in a different colour**

* The default colours are chosen automatically by the plot
* Check the **top left corner**
  + **Colour Legends** for each category are also shown automatically

**Now, What if we had to draw the same Scatter Plot using Matplotlib?**

* We’d have to separate the datasets for the 3 flowers first
* Then, we’d have to write scatterplot code for each individual dataset

**Let’s try and draw the same Scatter Plot using Matplotlib and compare it with Seaborn**

setosa = iris[iris['species'] == 'setosa']

versicolor = iris[iris['species'] == 'versicolor']

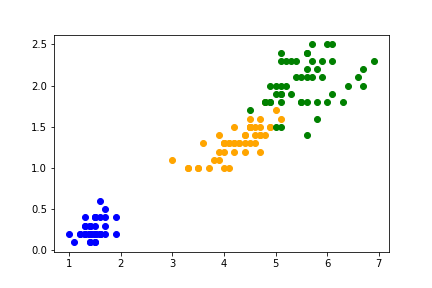
virginica = iris[iris['species'] == 'virginica']

plt.scatter(x=setosa['petal\_length'], y=setosa['petal\_width'], c = 'blue')

plt.scatter(x=versicolor['petal\_length'], y=versicolor['petal\_width'], c = 'orange')

plt.scatter(x=virginica['petal\_length'], y=virginica['petal\_width'], c = 'green')

plt.show()



* As you can see, we got the same plot.
* But with seaborn, the **code is just so much simpler and smaller**
* That’s the **convenience** we have with seaborn

**Let’s see a few more plots that we can visualize using seaborn**

**Joint Plot**

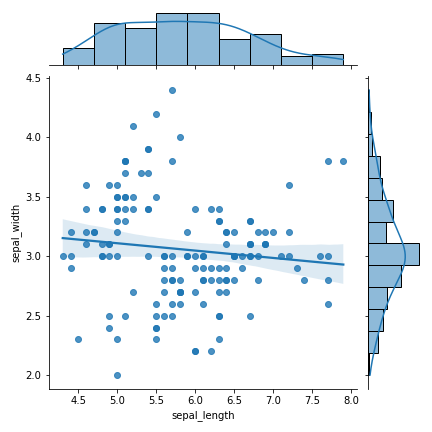
* It draws a plot of two variables
* It shows scatter, histogram and KDE graphs in the same plot.

**Let’s check it out**

* We will take **sepal\_length as x-coordinates** and **sepal\_width as y-coordinates**
* Again, we will pass iris dataset as **input data parameter**
* We can select from different values for **parameter kind** and it **will plot accordingly**
  + “scatter” | “kde” | “hist” | “hex” | “reg” | “resid”
* We will set **parameter kind** to **'reg'** here

sns.jointplot(x= 'sepal\_length', y = 'sepal\_width', data= iris, kind='reg')

plt.show()



**As we can see here:**

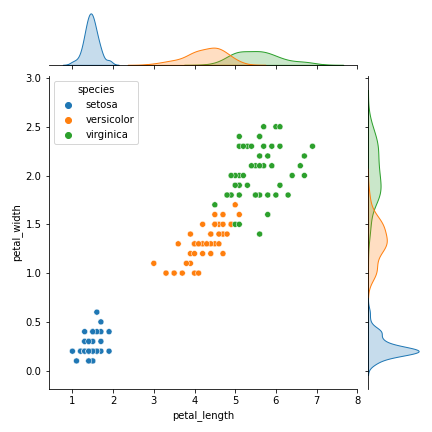
* jointplot plots **scatter, histogram and KDE in the same graph** when we set **kind=reg**
* Scatter shows the **scattering of (sepal\_length, sepal\_width) pairs as (x, y) points**
* Histogram and KDE shows the separate distributions of sepal\_length and sepal\_width in the data

**We can also add hue to Joint Plot**

* Let’s check how the 3 species of flowers are distributed in terms of petal\_length and petal\_width
* We’ll take **petal\_length** as **x-coordinates** and **petal\_width** as **y-coordinates**

sns.jointplot(x= 'petal\_length', y = 'petal\_width', data= iris, hue='species')

plt.show()



**Pair Plot**

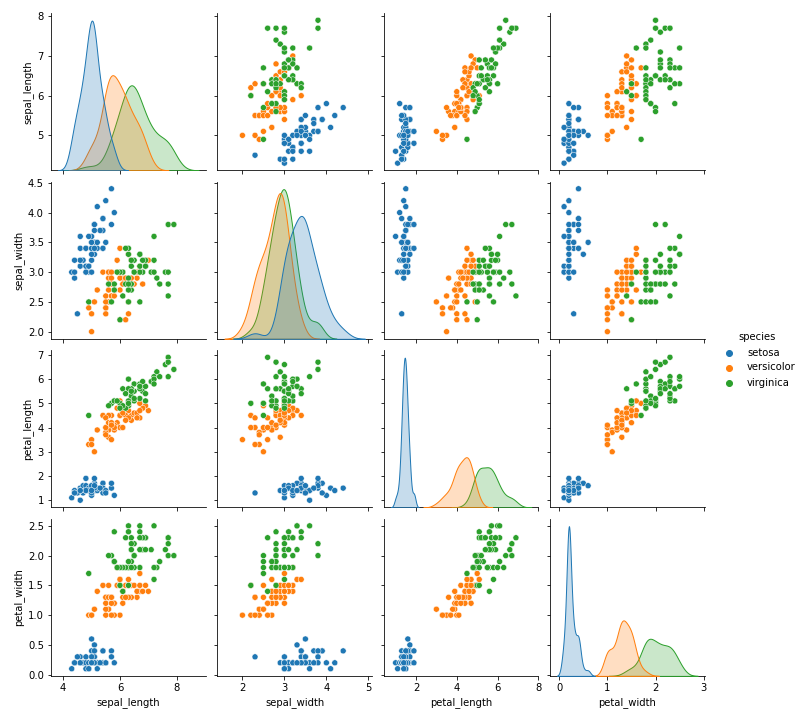
* pairplot() in seaborn creates a **grid of Axes by default**
* Each numeric attribute in data is shared across the y-axes across a single row and the x-axes across a single column.
* It displays a **scatterplot between each pair of attributes in the data** with different **hue** for each category

**Since, the diagonal plots belong to same attribute at both x and y axis, they are treated differently**

* A univariate distribution plot is drawn to show the marginal distribution of the data in each column.

sns.pairplot(data = iris, hue= 'species')

plt.show()



**Notice that:**

* It is **like a scatterplot of iris with hue='species'**
* But the **scatter is plotted between every pair of attributes**
* **Colour Legends** for each species category are given on **right side**
* It shows **relation between each pair of attributes**

**Diagonal plots are different from scatterplots**

* Because x and y axis have same attribute
* Diagonal plots show a univariate curve category-wise for each attribute

**It is also possible to show a subset of variables or plot different variables on the rows and columns**

* Feel free to experiment this on your own

**Categorical Plots**

**Now, we’ll see some Categorical Plots using seaborn**

* Categorical Plots are the **plots based on categories**

**Do we have a categorical variable in our dataset?**

* **species in the case of iris dataset**

**Which plots, in general, do you think are suitable to visualize categorical variables?**

* Bar Plots
* Because we want to see the **frequency/count of data points belonging to each category** of that categorical variable

**Count Plot**

* It’s like a bar plot
* It plots a simple bar graph displaying **count of datapoints (rows) belonging to each category**
* We provide **column label on x-axis**
* **Count of data** (no. of flowers) **belonging to each category** in the column is **on y-axis**

sns.countplot(x = 'species', data = iris)

plt.show()



**What do you notice?**

**How many flowers in dataset belong to each species?**

* **All the 3 species** - setosa, versicolor and virginica have the **same number of flowers**
* There are **50 flowers belonging to each species in the Iris Dataset**

**Box Plot**

* It draws a box plot to show distributions with respect to categories.

**But what exactly is a Box Plot?**

* A box plot or **box-and-whisker plot** shows the **distribution of quantitative data** in a way that **facilitates comparisons between attributes** or **across levels** of a categorical attribute.
* The **box** shows the **quartiles** of the dataset
* While the **whiskers** extend to show the **rest of the distribution**
* Except for points that are determined to be “outliers” using a method that is a function of the **inter-quartile range**.

**Let’s start with understanding what’s a quartile**

* Box plot shows distribution of numerical data and skewness through displaying the **data percentiles**, called **quartiles**

**Box plots show the five-number summary of data:**

1. Minimum score,
2. first (lower) quartile
3. Median
4. Third (upper) quartile
5. maximum score

**Minimum Score**

* It is the **lowest value**, excluding outliers
* It is shown at the **end of bottom whisker**

**Lower Quartile**

* **25% of values fall below the lower quartile value**
* It is also known as the **first quartile**.

**Median**

* Median marks the **mid-point of the data**
* It is shown by the **line that divides the box into two parts**
* It is sometimes known as the **second quartile**.
* **Half the scores are greater than or equal to this value and half are less**.

**Upper Quartile**

* **75% of the values fall below the upper quartile value**
* It is also known as the **third quartile**.
* So, **25% of data are above this value**.

**Maximum Score**

* It is the **highest value**, excluding outliers
* It is shown at the **end of upper whisker**.

**Whiskers**

* The upper and lower whiskers represent **values outside the middle 50%**
* That is, the **lower 25% of values** and the **upper 25% of values**.

**Interquartile Range (or IQR)**

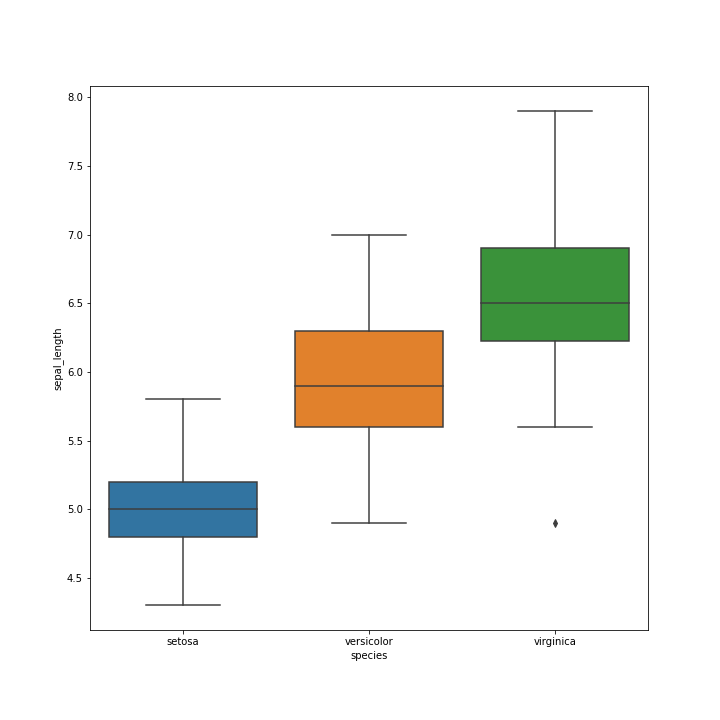
* This is the box plot showing the **middle 50% of scores**
* It is the **range between the 25th and 75th percentile**.

**Now, Let’s plot a box plot to check variation of sepal\_length among the 3 species of iris**

plt.figure(figsize=(10,10))

sns.boxplot(x = 'species', y = 'sepal\_length', data = iris)

plt.show()



**As you can see:**

* **Different species have different range of sepal\_length**

**Each species’ box plot shows:**

* the **lowest sepal\_length** in data for that species
* the **25th percentile (lower quartile) value of sepal\_length** for that species
* the **median sepal\_length** in data for that species
* the **75th percentile (upper quartile) value of sepal\_length** for that species
* the **highest sepal\_length** in data for that species

**Whiskers show the**

* **sepal\_length outside the middle 50% of values**
* The **lower 25% of sepal\_length** and the **upper 25% of sepal\_length**.

**Voilin Plot**

* Its a **combination of Box Plot and Distribution Plot**.
* It works similar to a box and whisker plot.
* It shows the distribution of quantitative data across several levels of categorical attribute such that those distributions can be compared.

**How is it different from box plot?**

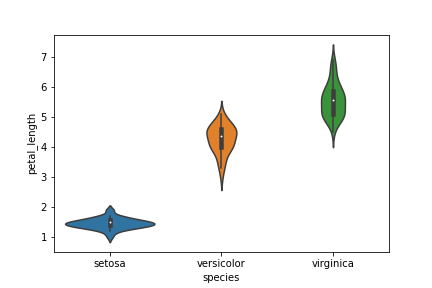
* In a box plot, all of the plot components correspond to actual datapoints
* Whereas, the violin plot features a KDE of the underlying distribution.

**This can be an effective and attractive way to show multiple distributions of data at once**

**Let’s draw a Violin Plot for petal\_length for the 3 species of iris**

sns.violinplot(x = 'species', y = 'petal\_length', data = iris)

plt.show()



**Observe:**

* Inside violin, you will see a boxplot.
* Left and right side represents the **distributions of petal\_length for each species**.

**Finding correlations among attributes**

* We can find the level of correlation b/w different attributes (variables)

**But what exactly is a correlation?**

* Two variables are correlated when **they change in same/opposite direction**

**We can check coefficient of correlation using corr()**

iris.corr()

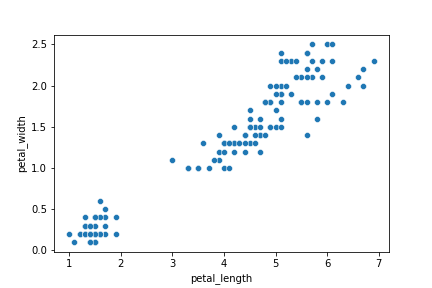
|  | **sepal\_length** | **sepal\_width** | **petal\_length** | **petal\_width** |
| --- | --- | --- | --- | --- |
| **sepal\_length** | 1.000000 | -0.117570 | 0.871754 | 0.817941 |
| **sepal\_width** | -0.117570 | 1.000000 | -0.428440 | -0.366126 |
| **petal\_length** | 0.871754 | -0.428440 | 1.000000 | 0.962865 |
| **petal\_width** | 0.817941 | -0.366126 | 0.962865 | 1.000000 |

* Higher the **MAGNITUDE** of coefficient of correlation, more the variables are **correlated**
* The **sign just determines the direction of change**
  + + means increase in value of one variable causes increase in value of other variable
  + - means increase in value of one variable causes decrease in value of other variable, and vice versa

**As you can see, petal\_length and petal\_width have the highest correlation coeff of 0.96**

sns.scatterplot(x= 'petal\_length', y= 'petal\_width', data = iris)

plt.show()



* When petal\_length increases, petal\_width also increases

**But Remember**

**Correlation does NOT mean Causation**

* We cannot conclude that change in values of a variable is causing change in values of other variable

**Now, Let’s look at a way to visualize correlation among variables**

**Heat Map**

* A heat map plots rectangular data as a color-encoded matrix.
* **Stronger the colour, stronger the correlation b/w the variables**

**Let’s plot a Heat Map using correlation coefficient matrix generated using corr()**

sns.heatmap(iris.corr(), cmap= "Blues", annot=True)

plt.show()



* **annot=True** is for writing correlation coeff inside each cell

**You can change the colours of cells in Heat Map if you like**

* There are a lot of options available!

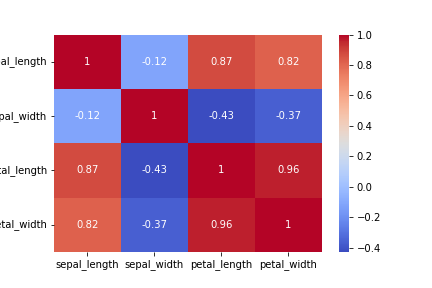
print(plt.colormaps())

['magma', 'inferno', 'plasma', 'viridis', 'cividis', 'twilight', 'twilight\_shifted', 'turbo', 'Blues', 'BrBG', 'BuGn', 'BuPu', 'CMRmap', 'GnBu', 'Greens', 'Greys', 'OrRd', 'Oranges', 'PRGn', 'PiYG', 'PuBu', 'PuBuGn', 'PuOr', 'PuRd', 'Purples', 'RdBu', 'RdGy', 'RdPu', 'RdYlBu', 'RdYlGn', 'Reds', 'Spectral', 'Wistia', 'YlGn', 'YlGnBu', 'YlOrBr', 'YlOrRd', 'afmhot', 'autumn', 'binary', 'bone', 'brg', 'bwr', 'cool', 'coolwarm', 'copper', 'cubehelix', 'flag', 'gist\_earth', 'gist\_gray', 'gist\_heat', 'gist\_ncar', 'gist\_rainbow', 'gist\_stern', 'gist\_yarg', 'gnuplot', 'gnuplot2', 'gray', 'hot', 'hsv', 'jet', 'nipy\_spectral', 'ocean', 'pink', 'prism', 'rainbow', 'seismic', 'spring', 'summer', 'terrain', 'winter', 'Accent', 'Dark2', 'Paired', 'Pastel1', 'Pastel2', 'Set1', 'Set2', 'Set3', 'tab10', 'tab20', 'tab20b', 'tab20c', 'magma\_r', 'inferno\_r', 'plasma\_r', 'viridis\_r', 'cividis\_r', 'twilight\_r', 'twilight\_shifted\_r', 'turbo\_r', 'Blues\_r', 'BrBG\_r', 'BuGn\_r', 'BuPu\_r', 'CMRmap\_r', 'GnBu\_r', 'Greens\_r', 'Greys\_r', 'OrRd\_r', 'Oranges\_r', 'PRGn\_r', 'PiYG\_r', 'PuBu\_r', 'PuBuGn\_r', 'PuOr\_r', 'PuRd\_r', 'Purples\_r', 'RdBu\_r', 'RdGy\_r', 'RdPu\_r', 'RdYlBu\_r', 'RdYlGn\_r', 'Reds\_r', 'Spectral\_r', 'Wistia\_r', 'YlGn\_r', 'YlGnBu\_r', 'YlOrBr\_r', 'YlOrRd\_r', 'afmhot\_r', 'autumn\_r', 'binary\_r', 'bone\_r', 'brg\_r', 'bwr\_r', 'cool\_r', 'coolwarm\_r', 'copper\_r', 'cubehelix\_r', 'flag\_r', 'gist\_earth\_r', 'gist\_gray\_r', 'gist\_heat\_r', 'gist\_ncar\_r', 'gist\_rainbow\_r', 'gist\_stern\_r', 'gist\_yarg\_r', 'gnuplot\_r', 'gnuplot2\_r', 'gray\_r', 'hot\_r', 'hsv\_r', 'jet\_r', 'nipy\_spectral\_r', 'ocean\_r', 'pink\_r', 'prism\_r', 'rainbow\_r', 'seismic\_r', 'spring\_r', 'summer\_r', 'terrain\_r', 'winter\_r', 'Accent\_r', 'Dark2\_r', 'Paired\_r', 'Pastel1\_r', 'Pastel2\_r', 'Set1\_r', 'Set2\_r', 'Set3\_r', 'tab10\_r', 'tab20\_r', 'tab20b\_r', 'tab20c\_r', 'rocket', 'rocket\_r', 'mako', 'mako\_r', 'icefire', 'icefire\_r', 'vlag', 'vlag\_r', 'flare', 'flare\_r', 'crest', 'crest\_r']

sns.heatmap(iris.corr(), cmap= "coolwarm", annot=True)

plt.savefig("plot13")

plt.show()



**Choosing right visualization for a given purpose**

* There’s a whole bunch of charts and plots we’ve seen
  + Bar chart
  + Historgam
  + Box Plot
  + Violin Plot
  + Scatterplot
  + Count Plot
  + Heat Map
  + … and so on
* But we always need to **select the right plot for every purpose**

**What is the right chart to use for a given problem?**

* We need to decide the right chart/plot to use for a dataset at-hand
* We can’t just blindly use any chart for any data that’s available to us
* There are certain **thumb rules** that we need to consider

**Now, What exactly is the process of selecting the right chart?**

* First you need to look at what is the type of variable you’re dealing with

**Let’s divide this step in into:**

1. 1-Dimensional
2. 2-Dimensional
3. Multi-Dimensional

**1-Dimensional Visualization**

**Which one will we choose when we just want to analyze 1 variable?**

* 1-D
* We look at only 1 variable at a time

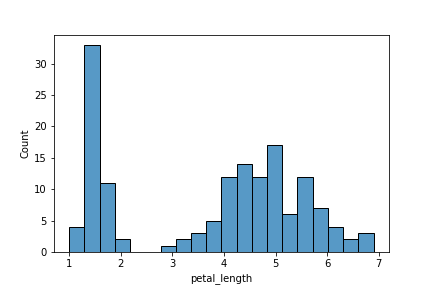
**Now, Which plot to use if the variable is continuous (numeric)?**

* **Histogram** —> To see **distribution** of that continuous variable
* **Box Plot** —> To see the **inter-quartile range of values** of that continuous variable

**For example: What we saw with petal\_length above:**

sns.histplot(iris['petal\_length'], bins= 20)

plt.show()



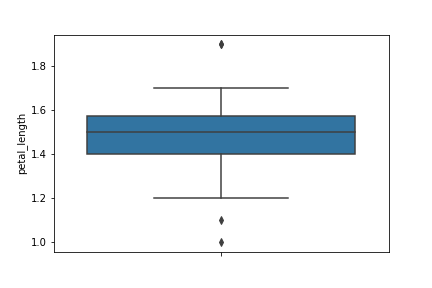
**Let’s say we want to visualize how petal\_length varies for just setosa species**

* We can utilize the **Box Plot** here we saw earlier

setosa = iris[iris['species'] == 'setosa']

sns.boxplot(y = 'petal\_length', data = setosa)

plt.show()



**What if the variable is categorical?**

* Categorical variable will have **discrete unique values**
* **Bar Chart / Count Plot** —> To see **how many datapoints belong to each category**
* **Pie Chart** —> To see **ratio (%age) of datapoints** belonging to each category

**For example: Bar/Count Plot we saw above to check no. of flowers belonging to each species:**

sns.countplot(x = 'species', data = iris)

plt.show()



**We can also check the proportions / %ages of flowers belonging to each species using a Pie Chart:**

* **Seaborn doesn’t have a direct function to create a pie chart**
* So, we’ll use **matplotlib** here

n\_setosa = iris[iris['species'] == 'setosa'].shape[0] # We are getting the number of datapoints in each species

n\_versicolor = iris[iris['species'] == 'versicolor'].shape[0]

n\_virginica = iris[iris['species'] == 'virginica'].shape[0]

data = [n\_setosa, n\_versicolor, n\_virginica]

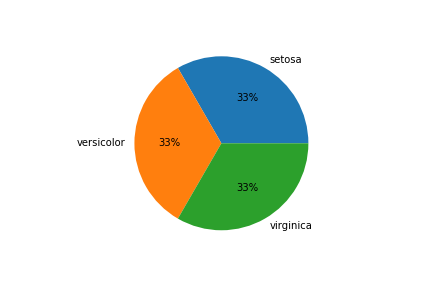
labels=['setosa','versicolor','virginica']

plt.pie(data,

labels=labels,

autopct='%.0f%%') # To show the portions in %ages

plt.show()



**2-Dimensional Visualization**

**Now, What if we want to analyze 2 variables at a time?**

* We usually do this to check the relationship b/w 2 variables
* Its a **Bi-Variate Analysis**

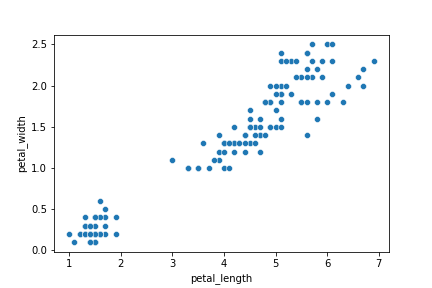
**Now, Which chart will we use if both variables are continuous (numeric)?**

* **Scatter Plot** —> To see how the 2 continuous variables are **dependent on each other or vary with each other**
* **Line Chart** —> To see the **approximate relationship (dependency) b/w the 2 variables** represented by a line

**For example: When we analyzed how petal\_length and petal\_width vary with each other**

sns.scatterplot(x= 'petal\_length', y= 'petal\_width', data = iris)

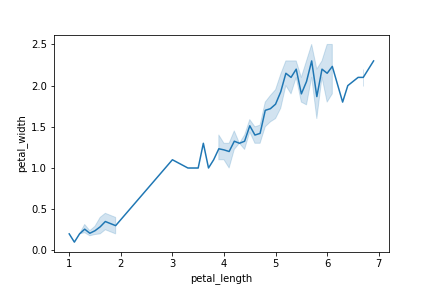
plt.show()



**We can also use Line Chart to get an approximate relationship b/w petal\_length and petal\_width**

sns.lineplot(x= 'petal\_length', y= 'petal\_width', data = iris)

plt.show()



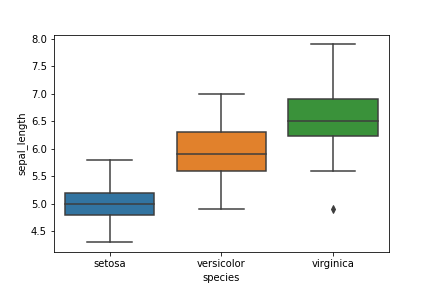
**Now, Which chart to use when we have 1 continuous and 1 discrete variable?**

* You should know this by now!
* **Box Plot** —> To see **distribution of numeric variable across each category of categorical variable**

**For example: When we checked above how sepal\_length varies for each species of iris**

sns.boxplot(x = 'species', y = 'sepal\_length', data = iris)

plt.show()



* It helps us answer questions like **“is there a change/difference in sepal\_length for different species?”**
* Box Plot allows these types of **comparisons of a numerica variable among different categories**

**Now, What if both our variables are categorical?**

* We can use a Bar Chart (Stacked or Dodged)

**For example:**

* Let’s say we have some employees data of a company
* We want to check **how many employees are there in each department - Engg, HR, Operations, Sales**
* Now, we want to check **how many Males and Females are there in each department**
* A **Stacked/Dodged Bar Chart can be appropriately used here**

**Multi-Dimensional Visualization**

**Now, Let’s talk about multi-dimensional visualization**

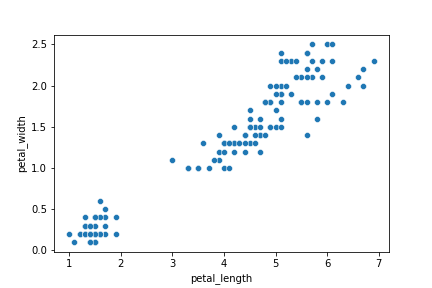
* It’s not very straightforward
* But if plotted in a **constructive way**, a multi-dimensional plot can reveal a lot of useful information

**Let’s start with 2 dimensions, i.e., 2 variables, and then we’ll add more dimensions to the plot**

**Do you remember the Scatter Plot b/w petal\_length and petal\_width?**

sns.scatterplot(x= 'petal\_length', y= 'petal\_width', data = iris)

plt.show()



We can make use of

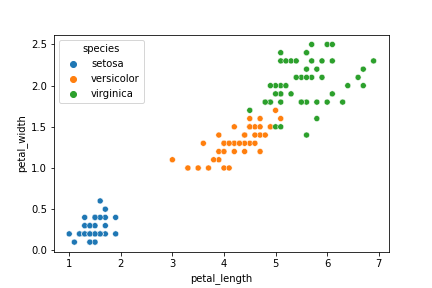
* Colour
* Shape
* Size

to add more dimensions to our Scatter Plot

We have already used “colour” when we plotted Scatter Plot b/w petal\_length and petal\_width and added species as hue to the Scatter Plot

sns.scatterplot(x ='petal\_length', y ='petal\_width' , data= iris, hue='species')

plt.show()



* Different species are now represented by different colours
* We had a 2-D Scatter Plot b/w petal\_length and petal\_width
* The, we added **species as a 3rd dimension** using the **colour**

**Now, It’s a multi-dimensional plot**

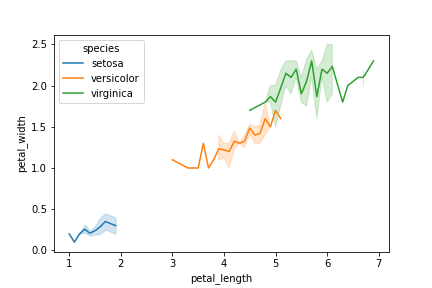
**By looking at it, Can you now tell “which points belong to which species of iris”?**

* You can also tell what’s the **range (clusters) of petal\_length and petal\_width** of each species

**We can also add this dimension of species in our Line Plot we saw earlier:**

sns.lineplot(x= 'petal\_length', y= 'petal\_width', data = iris, hue='species')

plt.show()



* Same way we can add more dimensions to the same plot using **shape** and **size** to both Scatter Plots
* We’ll cover them when we come to **Exploratory Data Analysis (EDA)**
* For now, just remember that we can always **constructively plot a multi-dimensional plot to reveal important information from our data**

**Visualizing Tips Dataset**

* Let’s move on to visualizing another dataset from seaborn library
* **Tips Dataset**
* Its some tipping data where one waiter recorded information about each tip he/she received over a period of a few months working in one restaurant

**Let’s go ahead and load the data**

tips = sns.load\_dataset('tips')

**Let’s see what all the tips data includes**

tips.shape

(244, 7)

tips.head()

|  | **total\_bill** | **tip** | **sex** | **smoker** | **day** | **time** | **size** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 16.99 | 1.01 | Female | No | Sun | Dinner | 2 |
| **1** | 10.34 | 1.66 | Male | No | Sun | Dinner | 3 |
| **2** | 21.01 | 3.50 | Male | No | Sun | Dinner | 3 |
| **3** | 23.68 | 3.31 | Male | No | Sun | Dinner | 2 |
| **4** | 24.59 | 3.61 | Female | No | Sun | Dinner | 4 |

**It has 7 columns**

* How much was the total bill
* How much did the customer tip
* Was the customer male or female
* Was the customer a smoker or not
* What day was it
* What meal time was it
* size represents no. of people in the party

**Now, we’ll answer a few questions related to dataset using visualization through seaborn**

**Can you visualize in how many cases the customer was male and in how many cases the customer was female?**

sns.countplot(x = tips['sex'])

plt.show()



**What all days does the dataset has?**

tips['day'].unique()

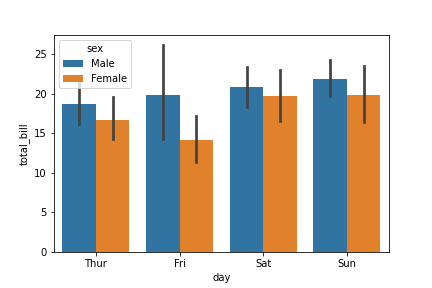
['Sun', 'Sat', 'Thur', 'Fri']

Categories (4, object): ['Thur', 'Fri', 'Sat', 'Sun']

**Can you tell out of all the days which day gets the highest bill amount for both male and female customers?**

sns.barplot(x = 'day', y = 'total\_bill', data=tips, estimator=np.mean, hue = 'sex')

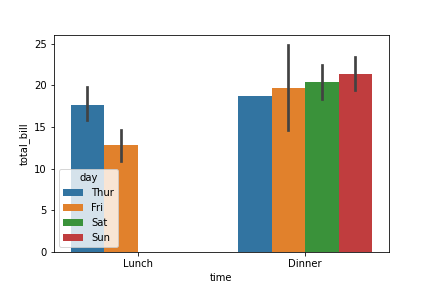
plt.show()



**Can you tell when do people prefer coming to restaurant more - lunch time or dinner time? And on what days?**

sns.barplot(x = 'time', y = 'total\_bill', data = tips, hue = 'day')

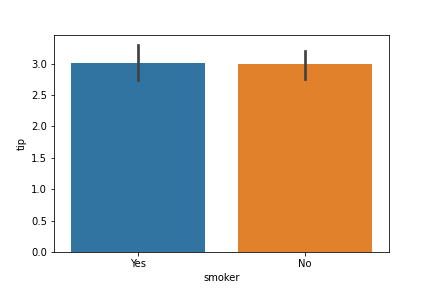
plt.show()



**Can you tell whether people who smoke give more tip or not?**

sns.barplot(x = 'smoker', y = 'tip', data = tips)

plt.show()

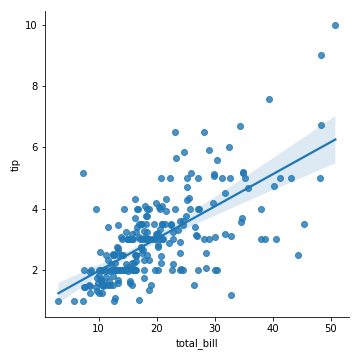


**How are total\_bill and tip related? Is there a correlation b/w them?**

* We can use lmplot()

sns.lmplot(x = 'total\_bill', y= 'tip', data = tips)

plt.show()



**This is all for seaborn for now**

* You can explore the library more by checking out its docs

Last changed by

**Matplotlib backend, Different plots, subplots and annotation**

**Topics covered:**

* Matplotlib backend:
  + inline v/s notebook
* Different plots in matplotlib:
  + Error bars
  + Contour plots
* Using plt.plot() methods with subplots
  + plt.xlabel() → ax.set\_xlabel()
  + plt.ylabel() → ax.set\_ylabel()
  + plt.xlim() → ax.set\_xlim()
  + plt.ylim() → ax.set\_ylim()
  + plt.title() → ax.set\_title()
* Styling and customisation:
  + Adding arrows to plots

**Matplotlib backend**

One of Matplotlib’s most important features is its ability to play well with many operating systems and graphics backends.

Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish.

Now there are 2 ways of viewing matplotlib plots:

1. Inline

* Displays plots in the script as static images
* We have been using this in the matplotlib lecture

%matplotlib inline

1. Notebook

* Uses the IPython backend
* Displays interactive plots in the script

%matplotib

OR

%matplotlib notebook

We have already seen the inline backend in the lecture

Lets look at the IPython backend now

Now that we are familiar with the matplotib backends lets talk about some of the other plots that can be drawn using matplotlib

**1. Error bars**

* For any scientific measurement, accurate accounting for errors is very important
* In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.
* This is where Error bars come into picture
* Lets see how we can draw them using matplotlib

import matplotlib.pyplot as plt

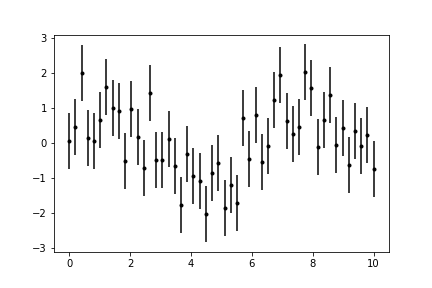
import numpy as np

x = np.linspace(0, 10, 50)

dy = 0.8

y = np.sin(x) + dy \* np.random.randn(50)

plt.errorbar(x, y, yerr=dy, fmt='.k')



This plot takes dy as the error in values of y

It is fixed as dy = 0.8

This gives a useful insight of how information might vary when some real-world data might be used

**2. Density/Contour plots**

* We have already studied how to plot 3D graphs
* But it turns out we can visualise them in 2D as well using Density/Contour plots
* Lets understand this through a code

We will plot a function z = f(x, y)

f(x,y)=sin(x)10+cos(10+xy)∗cos(x)f(x,y)=sin(x)10+cos(10+xy)∗cos(x)

def f(x, y):

return np.sin(x) \*\* 10 + np.cos(10 + y \* x) \* np.cos(x)

We can create a contour plot using plt.contour()

It takes three arguments: a grid of x values, a grid of y values, and a grid of z values.

The x and y values represent positions on the plot, and the z values will be represented by the contour levels.

We will create 2D grids from 1D array using np.meshgrid()

x = np.linspace(0, 5, 50)

y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)

Z = f(X, Y)

plt.contour(X, Y, Z, colors='black')

plt.show()

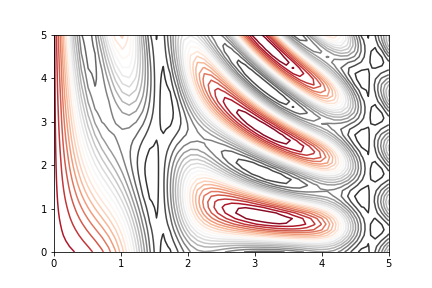


By default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines.

Alternatively, you can color-code the lines by specifying a colormap with the cmap argument.

Here, we’ll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range

plt.contour(X, Y, Z, 20, cmap = "RdGy")



There are other different plots that matplotlib can render

Refer to its docs for more details

**We will now talk about matplotlib subplots**

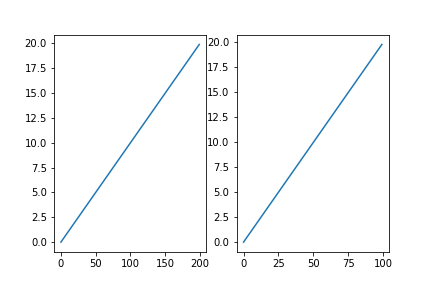
* We already know what they are and how to make them
* In this reading we will discover how different methods of plt.plot() can work with ax.plot()

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

ax[0].plot(np.arange(0, 20, step = 0.1))

ax[1].plot(np.arange(0, 20, step = 0.2))

plt.show()



Now to set labels of axes xlabel() and ylabel() methods are used with plt.plot()

For subplots axes these methods become set\_xlabel(), set\_ylabel()

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

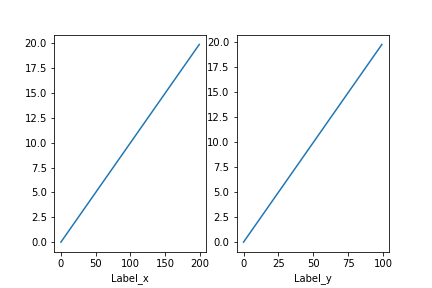
ax[0].set\_xlabel("Label\_x")

ax[0].plot(np.arange(0, 20, step = 0.1))

ax[1].set\_xlabel("Label\_y")

ax[1].plot(np.arange(0, 20, step = 0.2))

plt.show()



We can also set the y and x axis to be common for all the subplots by setting sharey and sharex argument true

fig, ax = plt.subplots(2, 2, sharey = True, sharex = True)

ax[0, 0].set\_xlabel("Label\_0")

ax[0, 0].plot(np.arange(0, 20, step = 0.1))

ax[0, 1].set\_xlabel("Label\_1")

ax[0, 1].plot(np.arange(0, 20, step = 0.2))

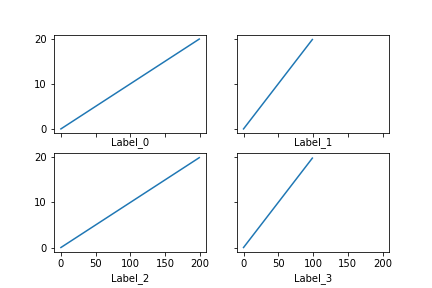
ax[1, 0].set\_xlabel("Label\_2")

ax[1, 0].plot(np.arange(0, 20, step = 0.1))

ax[1, 1].set\_xlabel("Label\_3")

ax[1, 1].plot(np.arange(0, 20, step = 0.2))

plt.show()



Similarly some other methods change as:

* plt.xlim() → ax.set\_xlim()
* plt.ylim() → ax.set\_ylim()
* plt.title() → ax.set\_title()

**Customisation**

* We have already learned how to style matplotlib plots
* We also know how to add text to plots
* Matplotlib also provides the functionality to annotate certain points using arrows
* To do this we can use plt.annotate() function.
* This function creates some text and an arrow, and the arrows can be very flexibly specified.

fig, ax = plt.subplots()

x = np.linspace(0, 20, 1000)

ax.plot(x, np.cos(x))

ax.axis('equal')

ax.annotate('local maximum', xy=(6.28, 1), xytext=(10, 4),

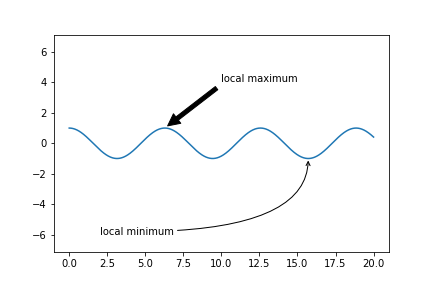
arrowprops=dict(facecolor='black', shrink=0.05))

ax.annotate('local minimum', xy=(5 \* np.pi, -1), xytext=(2, -6),

arrowprops=dict(arrowstyle="->",

connectionstyle="angle3,angleA=0,angleB=-90"))

plt.show()



The arrowprops argument allows you to decide the type of arrow to use

You can learn more about this by reading the docs

Matplotlib is a very vast library with a lot of functionalities. We will keep discussing new things in the future. Feel free to explore its documentation and try some things on your own

Last changed by

**Scaler Topics**Scaler Topics

 160

Published on [**HackMD**](https://hackmd.io/)