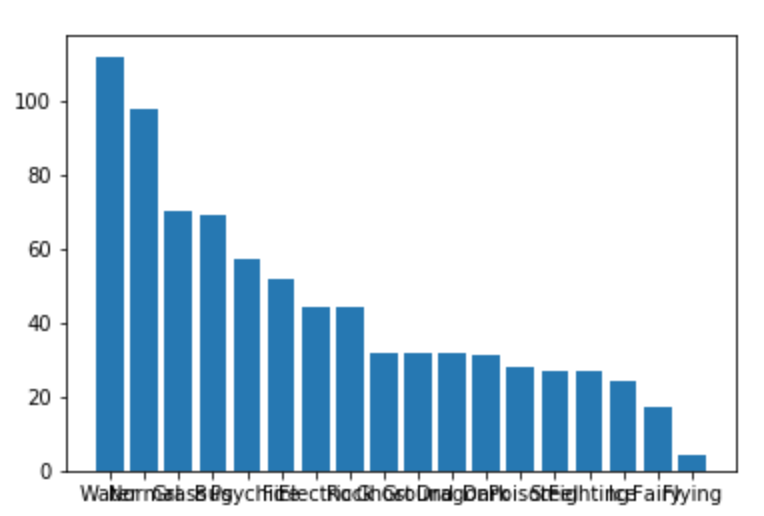
**Bar Chart**

We have now come to the right form of data that contains the answer. But is there an interesting way to onboard the answer to our friend? Well, a picture is worth a q thousand words. And thankfully with Python’s matplotlib library it fortunately, takes far less than a thousand words of code to create a production-quality graphic.

Let's look at a simple plot of the data constructed with default values. We will be using the bar plot to show the result. A **bar chart** or bar graph is a chart or graph that presents grouped data with rectangular bars with lengths proportional to the values that they represent. Bar plots can be both vertical and horizontal. To draw bar charts, we require two arrays; one representing the categories and the other denoting the heights of individual categories.



Look at the plots and can you see what is wrong? The plot has no information about what it is trying to show and no proper labels to clearly tell what they actually mean. The labels on the x-axis also are completely overlapping. We need to fix this. To do that first, we need to understand the basic anatomy of a plot made up of two important objects - Figure and Axes

* **Figure**: A Figure object is an outermost container for a matplotlib graphic. Within the **Figure**, everything else is contained. You can choose to create multiple independent figures.
* **Axes**: This actually refers to an individual plot and is added to a Figure; so, a Figure can contain multiple Axes. Usually, we'll set up an Axes with a call to subplot and so, both of them can be used interchangeably.

# importing libraries

import matplotlib.pyplot as plt

import numpy as np

import math

# Get the angles from 0 to 2 pie (360 degree) in narray object

X = np.arange(0, math.pi\*2, 0.05)

# Using built-in trigonometric function we can directly plot

# the given cosine wave for the given angles

Y1 = np.sin(X)

Y2 = np.cos(X)

Y3 = np.tan(X)

Y4 = np.tanh(X)

# Initialise the subplot function using number of rows and columns

figure, axis = plt.subplots(2, 2)

# For Sine Function

axis[0, 0].plot(X, Y1)

axis[0, 0].set\_title("Sine Function")

# For Cosine Function

axis[0, 1].plot(X, Y2)

axis[0, 1].set\_title("Cosine Function")

# For Tangent Function

axis[1, 0].plot(X, Y3)

axis[1, 0].set\_title("Tangent Function")

# For Tanh Function

axis[1, 1].plot(X, Y4)

axis[1, 1].set\_title("Tanh Function")

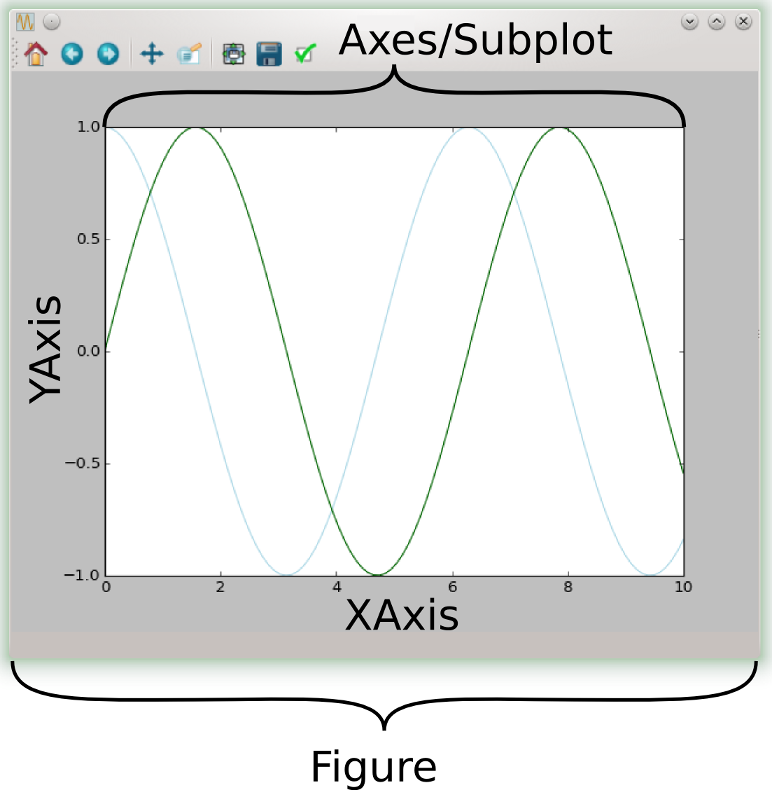
# Combine all the operations and display

plt.show()

Further, each Axes has an XAxis and a YAxis. These contain the **ticks, tick locations, labels** etc.

xticks(np.arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue'))

Almost every 'element' of a chart is its own manipulable Python object. But for our purpose, a basic understanding is sufficient. You will get a better understanding of the difference between a Figure and Axes in the snapshot below:



Now we will try to fix the bar chart that we have created using matplotlib. The steps we would need to do are:-

* adjust the size of the image so that the xticks become visible.
* label the x-axis and y-axis with meaningful labels.
* give a title for the plot.

We will apply these fixes to the plot and revisualize the plot and see if we can show it better. The labels for axes and the title must be concise and as self-explanatory as possible.

The steps generally associated with generating a plot through matplotlib are:

* First, importing matplotlib.pyplot as plt (customary)
* Initialize the figure with plt.figure(). It is the entire drawing canvas and you can adjust its size with the argument figsize()
* Then we label each of the axes through plt.xlabel() and plt.ylabel.
* Next, we title the plot using plt.title()
* Finally, build and show the plot with plt.bar()

Some of the steps are interchangeable like we can build the plot and label later, but it is a good practice to follow a neat sequence of steps.

**import** matplotlib.pyplot **as** plt

%matplotlib inline

*# initialize the figure*

plt.figure(figsize=[14,8])

*# label the axes*

plt.xlabel("Type 1 Pokemon Variants")

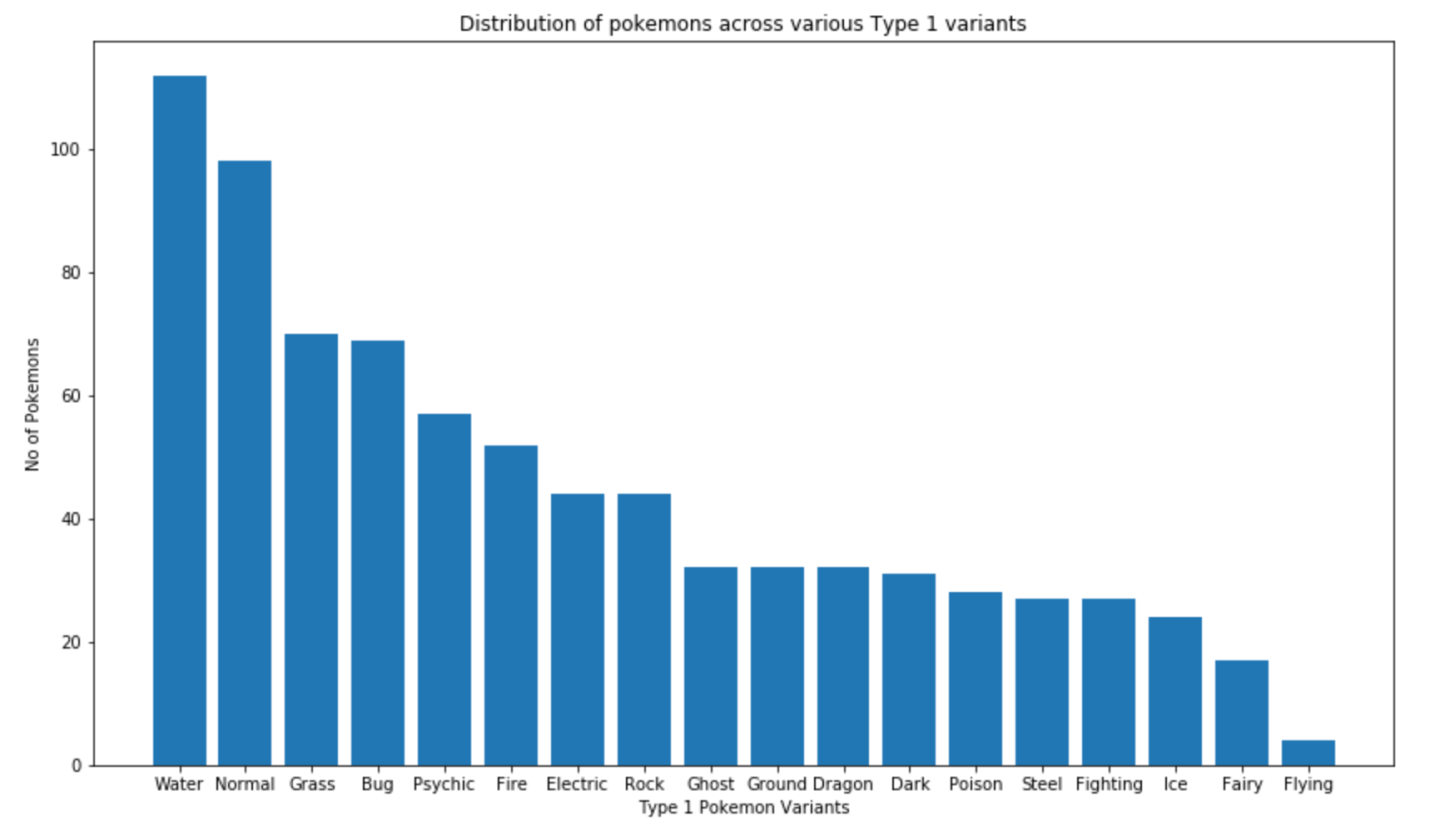
plt.ylabel("No of Pokemons")

*# title the plot*

plt.title("Distribution of pokemon across various Type 1 variants")

*# build and show the plot*

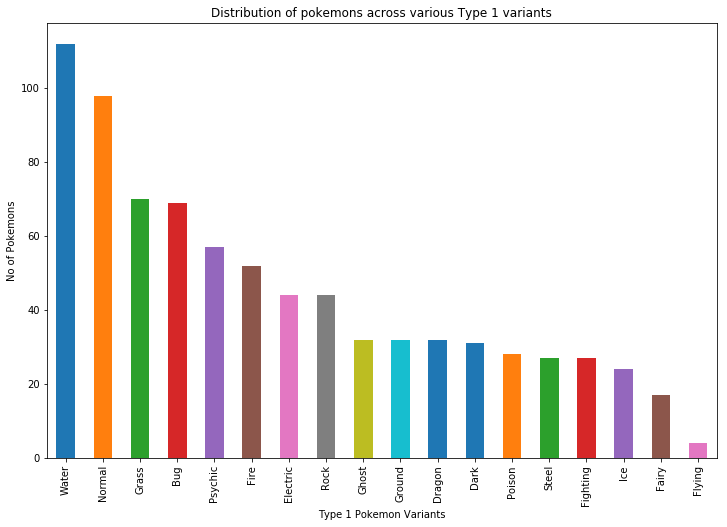
plt.bar(type\_1.index,type\_1\_data)

.

Additionally, from the pandas data frame itself, you can call the plot function and plot the same data. The internal defaults are a little different, so the graph might look a little different. But, essentially it is the same visualization.

*# keeping the same axis labels as earlier*

df['Type 1'].value\_counts().plot(kind="bar")



**Why use bar charts?**

If you have comparative data that you would like to represent through a chart then a bar chart is the best option. A bar chart uses bars to show comparisons between categories of data. These bars can be displayed horizontally or vertically. A bar graph will always have two axes. One axis will generally have numerical values, and the other will describe the types of categories being compared.

**Advantages and disadvantages of using bar charts**

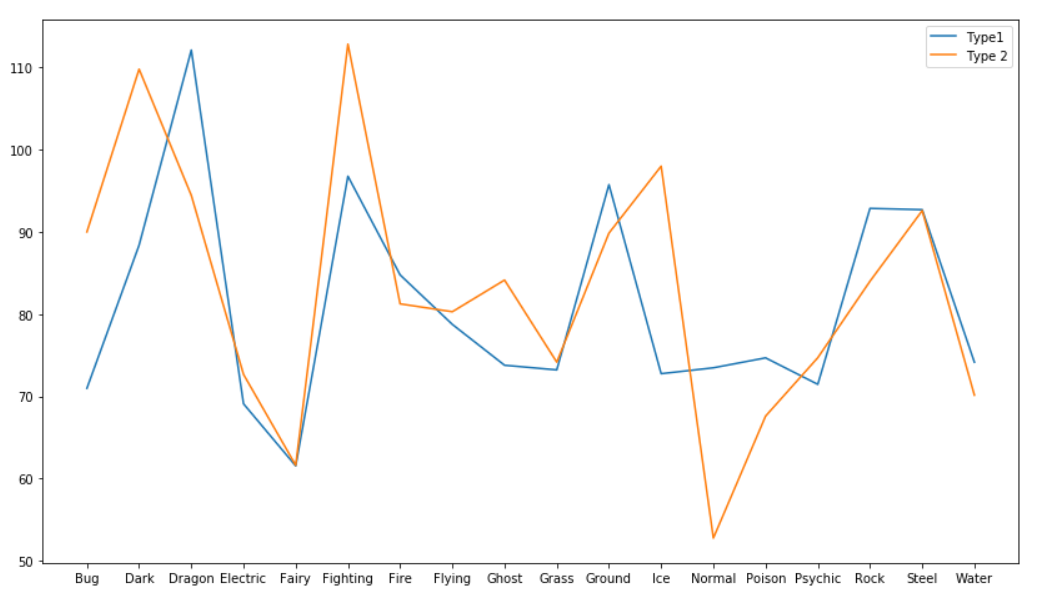
* The advantage is the bar chart is easy to read and understand. You get a good overview of values when using bar charts.
* The bar chart does not work so well with many dimension values due to the limitation of the axis length.

Line Plots and Plot Customizations

**What is a line plot?**

Line plots are simple charts used to display a series of data points connected by straight line segments. The X-axis lists the categories equally and the y-axis represents the corresponding category values. This kind of graph serves to visualize a trend summarized from data periodically and hence has wide applications in time series data. Some popular use cases of line graphs include the price trend of stocks, weather changes during a year etc.

A graph that looks like this:



The plot above helps in giving a clear comparison of the mean attack points for different variants of Type 1 (**depicted by blue line**) and Type 2 (**depicted by orange line**)

**Code for implementing line plot**

The code for implementing the line plot is quite simple:

plt.plot(x, y)

where x and y are the two arrays we want to plot on X-axis and Y-axis respectively.

**Advantages and disadvantages**

* The line chart is easy to understand and gives an instant perception of trends.
* Sometimes it might mess the entire chart if many categories are compared in one line chart.

**Why is it a good practice to customize your plot?**

We will answer this question by asking a set of questions. Suppose you are looking at the above image for the very first time. Can you answer these questions?

* What do the X-axes and Y-axes represent?
* Overall what does the line chart represent?

Obviously not. That's where plot customizations come to the rescue. You can customize your plot to make it unique and pleasing to the eye and depicting the important details; all at the same time. Across different plots, these elements (customizations) more or less remain the same. *Many types of customizations can be done to customize a plot; adjusting the colors, changing markers, lifestyles and linewidths, adding text, legend and annotations, and changing the limits and layout of your plots.*

**Widely used plot customizations**

Below are some of the widely used customization operations that you would be performing while using matplotlib:

1) **Adjusting size of the figure**: You can change of the figure using plt.figure(figsize=(x,y)) where you can set x and y values to satisfy your requirements

2) **Axes labels and title**: Use plt.title('Title') on the axes to set the title of the plot and plt.xlabel('xlabel'), plt.ylabel('ylabel') to set the labels

3) **Axes limits**: Use plt.xlim((a,b)) and plt.ylim((a,b)) to fix the boundaries in the range (a,b) within which you want to display the plot

4) **Changing color**: Use the color argument inside the different types of plot functions to change its color.

5) **Legends**: In case we have multiple types of charts in a single plot, you can differentiate them with legends. Use plt.legend() to display legends with the help of the **labels** keyword argument inside it. To

6) **Save figure**: You can easily save a figure to, for example, a **png** file by making use of plt.savefig(). The only argument you need to pass to this function is the file name, just like in this example:

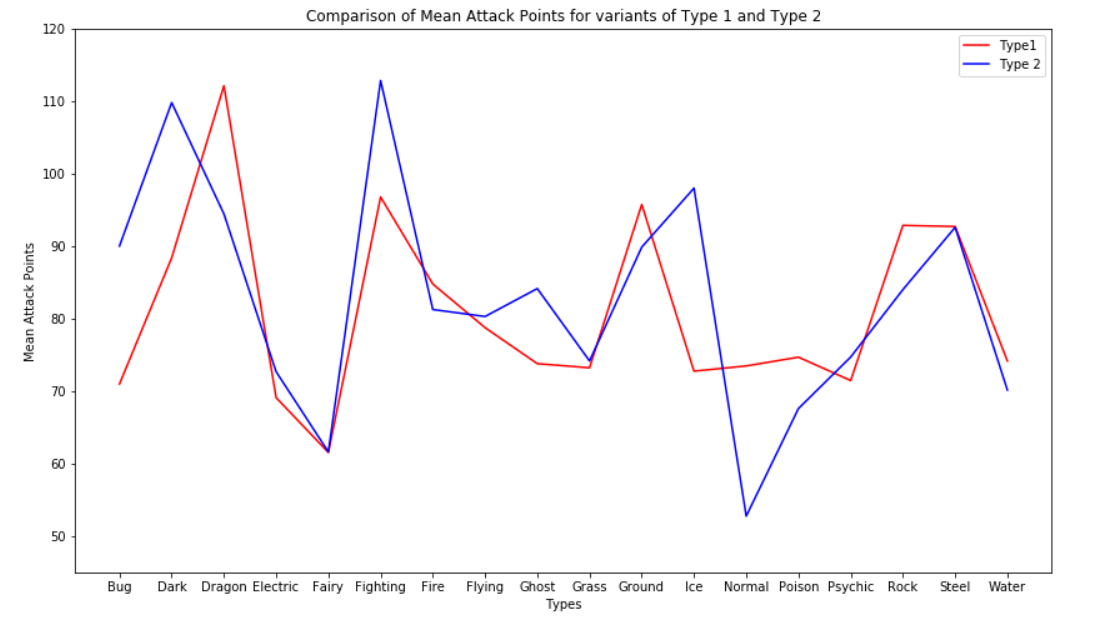
*# Save Figure*

plt.savefig("foo.png")

*# Save Transparent Figure*

plt.savefig("foo.png", transparent=**True**)

**Output**



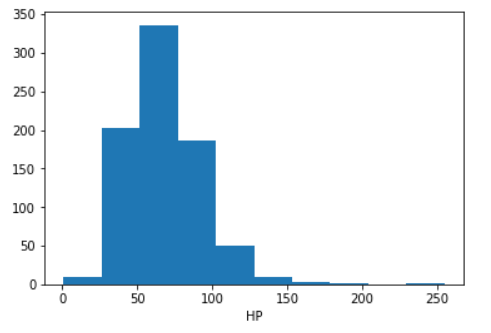
**Histogram**

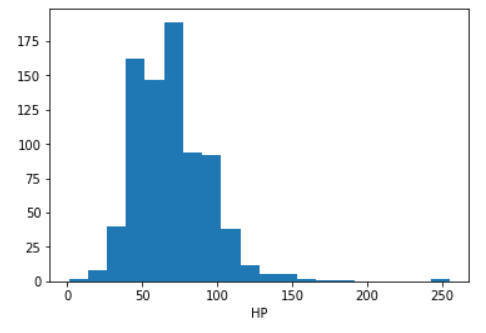
# What is a histogram and why do you need it?

A histogram is a plot that lets you discover, and show, the underlying frequency distribution (shape) of a set of continuous data. This allows the inspection of the data for its underlying distribution (e.g., normal distribution), outliers, skewness, etc. Note that it requires only one array or series since it displays the frequencies.

# How to construct a histogram using matplotlib?

Matplotlib's .hist() the method provides an easy way of generating histograms. To construct a histogram from a continuous variable you first need to split the data into intervals, called **bins**. Each bin contains the number of occurrences of scores in the data set that are contained within that bin. Below are two plots with bins 10 and 20 carried out with the feature HP of the Pokemon dataset.





**Key takeaways from these plots**

* First and the most important thing; **both these plots are the same**.
* An important concept with histograms is **binning**.
  + Binning is a way to group continuous values into a smaller number of **bins**.
  + More the number of bins more is the number of intervals that leads to less frequency for every interval and vice versa.
  + In the left image, the number of bins is small and so values are tightly packed. The right image has a number of bins, values are loosely packed and hence looks that way.
  + **But always remember that both these plots are the same; the difference in their appearance is due to the different number of bins.**
* Both these plots have a right tail i.e. have a long tail on the right side
* Some observations have extreme values in the interval 230-260
* Most of the observations lie in the interval 40-75

**Difference with bar charts**

Unlike a bar chart, there are no gaps between the bars (*although some bars might be absently reflecting no frequencies*). This is because a histogram represents a continuous data set, and as such, there are no gaps in the data.

**Histograms are based on the area of the bars, not the height of bars**

In a histogram, it is the area of the bar that indicates the frequency of occurrences for each bin. This means that the height of the bar does not necessarily indicate how many occurrences of scores there were within each individual bin. It is the product of height multiplied by the width of the bin that indicates the frequency of occurrences within that bin. One of the reasons that the height of the bars is often incorrectly assessed as indicating the frequency and not the area of the bar is due to the fact that a lot of histograms often have equally spaced bars (bins), and under these circumstances, the height of the bin does reflect the frequency.

Example:

Visualize the distribution of `Attack` points for `Dragon` type (`Type 1`) Pokemons

In this task, you will plot the distribution of Attack points for Pokemons which have their first type (Type 1) as Dragon. You will also compare the mean values for Attack for all the Pokemons against the mean value of Attack for dragon type (by drawing a vertical line).

* Calculate the mean attack points for all the Pokemons and store it in a variable mean\_attack
* Create a dataframe named dragon consisting of only Dragon type pokemon using conditional filtering (based on Type 1)
* Calculate the mean attack points for dragon and store it in a variable mean\_dragon
* Use matplotlib's .hist() on dragon and pass arguments column='Attack', bins=8
* To compare mean attack points you need to draw two lines; one with mean\_attack and mean\_dragon
* Use .axvline() and pass arguments x=mean\_attack, color='green' to plot a vertical line representing mean attack points for all the Pokemons
* Use .axvline() and pass arguments x=mean\_dragon, color='black' to plot a vertical line representing mean attack points for Dragon Pokemons

Skills Covered:

VisualizationPython

Hint

* Calculate mean attack points for all Pokemons using mean\_attack = np.mean(df['Attack'])
* Make dataframe consisting of only 'Dragon' type Pokemons with dragon = df[df['Type 1'] == 'Dragon']
* Calculate mean attack points for dragon type pokemons using mean\_dragon = np.mean(dragon['Attack'])
* To visualize distribution of attack points for dragon pokemons use dragon.hist(column='Attack', bins=8, figsize=(10,10))
* Use plt.axvline(x=mean\_attack, color='green') to draw a vertical green colored line indicating the mean attack points for all pokemons
* Use plt.axvline(x=mean\_dragon, color='black') to draw a vertical black colored line indicating the mean attack points for only dragon pokemons
* Finally use plt.show() to display the plot

Scatter Plot

**Why use scatter plots?**

A question for many data sets is whether two items are related to each other in some way, that is, are they correlated? In our case, you can ask the question of whether Attack and Defense points are related to each other. Scatter plot helps us answer this kind of questions.

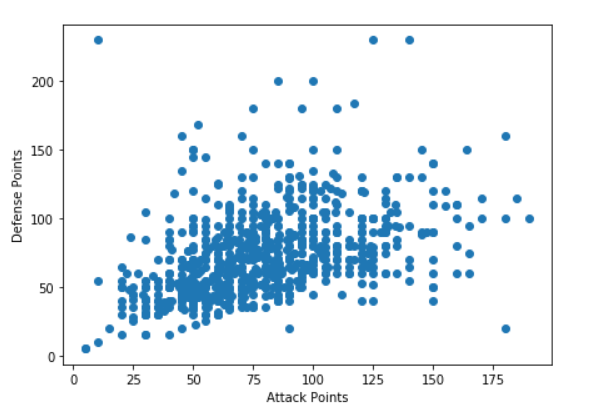
**What is a scatter plot?**

A scatter plot is a two-dimensional data visualization that is used to represent the values obtained for two different variables - one plotted along the x-axis and the other plotted along the y-axis. For generating a scatter plot you need **two numerical arrays** of data.

**When to use a scatter plot?**

A scatter plot helps us determine if two quantities are weakly or strongly correlated. Correlation implies that as one variable changes, the other also changes. While calculating the correlation coefficient will give us a precise number, a scatter plot helps us find outliers, gain a more intuitive sense of how to spread out the data is, and compare more easily.

For example:- The following scatter plot gives a clear indication between Attack and Defense points. As you can see there is a positive linear relationship between the Attack and Defense points.



**Drawing scatter plots**

You can use .scatter(x, y) to generate a scatter plot for two numeric arrays x and y. Using pandas you can achieve the same with a dataframe df using df.plot.scatter(x=column1, y=column2) where column1 and column2 are the column names present in the dataframe df. Keep in mind that both column1 and column2 must be numeric columns.

The code snippet is given below:

*# Scatter plot with matplotlib*

plt.scatter(df['Attack'], df['Defense'])

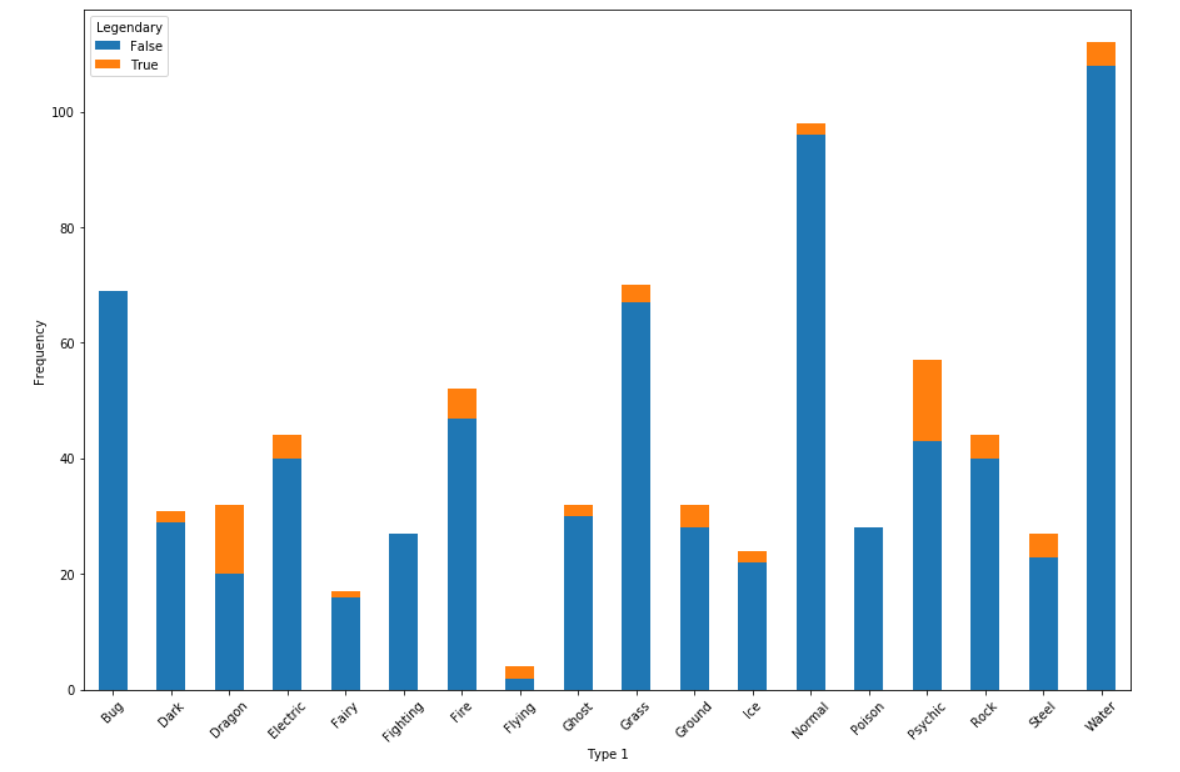
or you can do it with pandas also by:

*# Scatter plot with pandas*

df.plot.scatter(x='Attack', y='Defense')

Stacked Bar Chart

. A bar chart is a powerful visualization capable of representing category counts for a particular feature and can be either horizontal or vertical. But there is another kind of bar chart called **stacked bar chart** which can factor in another feature besides the feature you already have in the X-axis in a bar chart. Lets say you want to visually inspect the question: **Which type (Type 1) of Pokemons have the highest chances of being Legendary?**



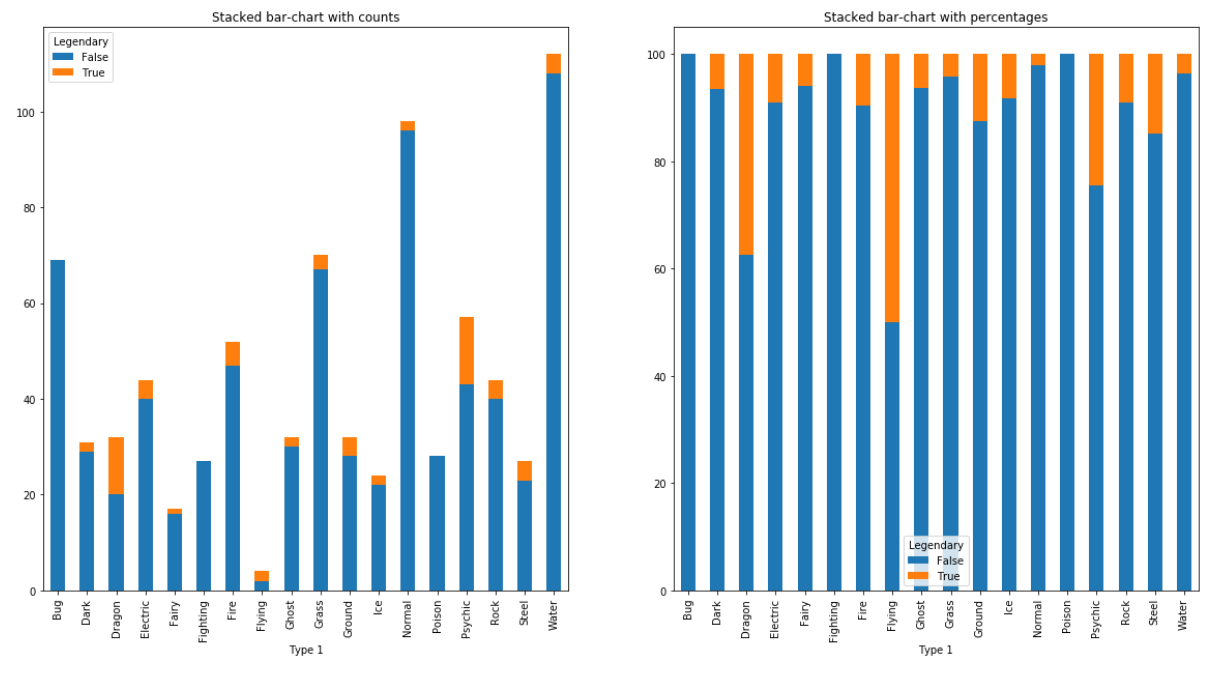
The above figure shows a stacked bar chart with number of legendary pokemons denoted by **orange** and non-legendary one by **blue**. It is evident from this plot that type (Type 1)**Flying** has the highest chances of being Legendary.

Drawing Multiple Plots

**Subplots**

Till now you have drawn only a single plot within the figure. Many times you will be required to draw multiple plots on the same figure for better comparison. Remember the stacked bar chart that had every variant of Type 1 and displayed their Legendary status? It was used to answer the question **Which type (Type 1) of Pokemons has the highest chances of being Legendary?**

A better way would answer this question is rather than getting the counts why don't we get the percentages and compare both the side of the plot by side? To achieve it i.e. place two plots side by side in the same figure we will use .subplots() method of matplotlib.



**It is very clear that the second plot (the right one) gives a much better visualization and effectively answers our question as all the categories are on the same scale**

The plot() method can have an optional format string argument to specify **color, style and size of line and marker.**

Legend

The **legend()** method of axes class adds a legend to the plot figure. It takes three parameters −

ax.legend(handles, labels, loc)

Where labels is a sequence of strings and handles a sequence of Line2D or Patch instances. loc can be a string or an integer specifying the legend location.

|  |  |
| --- | --- |
| **Location string** | **Location code** |
| Best | 0 |
| upper right | 1 |
| upper left | 2 |
| lower left | 3 |
| lower right | 4 |
| Right | 5 |
| Center left | 6 |
| Center right | 7 |
| lower center | 8 |
| upper center | 9 |
| Center | 10 |

Color codes

|  |  |
| --- | --- |
| **Character** | **Color** |
| ‘b’ | Blue |
| ‘g’ | Green |
| ‘r’ | Red |
| ‘b’ | Blue |
| ‘c’ | Cyan |
| ‘m’ | Magenta |
| ‘y’ | Yellow |
| ‘k’ | Black |
| ‘b’ | Blue |
| ‘w’ | White |

Marker codes

|  |  |
| --- | --- |
| **Character** | **Description** |
| ‘.’ | Point marker |
| ‘o’ | Circle marker |
| ‘x’ | X marker |
| ‘D’ | Diamond marker |
| ‘H’ | Hexagon marker |
| ‘s’ | Square marker |
| ‘+’ | Plus marker |

Line styles

|  |  |
| --- | --- |
| **Character** | **Description** |
| ‘-‘ | Solid line |
| ‘—‘ | Dashed line |
| ‘-.’ | Dash-dot line |
| ‘:’ | Dotted line |
| ‘H’ | Hexagon marker |