Python Libraries

For Data Science

Last lecture reminder

We learned about:

- Pandas pivot table
- Different ways to extract Pandas Dataframe
- Introduction to Matplotlib library
- Matplotlib - Basic plot functions
- Figure object methods and functions
- Generate complex plots using figure object

Matplotlib - Subplot

plt.subplot() → Subplot method allow us to create multiple charts and handle each of the individually.

The subplot method return 2 objects:

- **Figure object** → The “canvas” object that everything is drawn on.
- **Array of axes** → These are the actual plots. An Axes object is essentially what we think of as 'a plot'.
  It is the region of the image with the data space. Axes can be modified with labels, title, xlim, ylim..

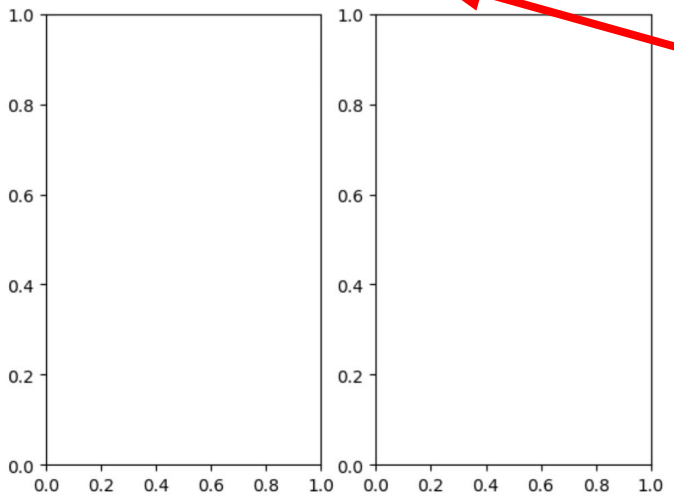
It is recommended to get back both objects once using subplot.

We can also provide the subplot method “**nrows”** and “**ncols”** parameters, those parameters determine how the plot grid will be organized.

In case we pass nrows > 1 & ncols > 1 we will get a matrix of charts in our subplot object.

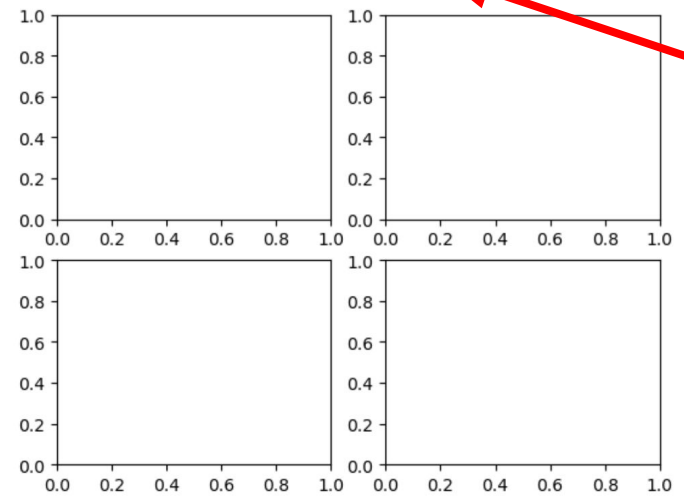
Matplotlib - Subplot

```
In [15]: fig, axes = plt.subplots(nrows=1, ncols=2)
```



When passing nrows = 1 or ncols = 1 we getting array of axes

```
In [16]: fig, axes = plt.subplots(nrows=2, ncols=2)
```



When passing nrows > 1 and ncols > 1 we getting matrix of axes

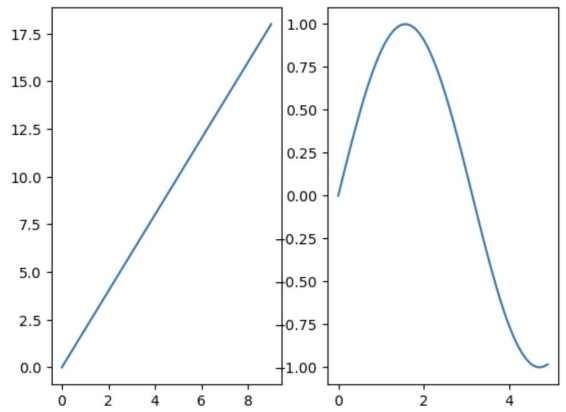
Matplotlib - Subplot

Once we created a new axes object we can now add the data we want to plot individually to each chart in the array by using [{index}], In case we create an axes matrix we will need to provide both the row index and the column index. **For example →**

Array type subplot

```
In [35]: fig, axes = plt.subplots(nrows=1, ncols=2)
         x1 = np.arange(10)
         y1 = x1*2
         x2 = np.arange(0,5,0.1)
         y2 = np.sin(x2)
         axes[0].plot(x1,y1)
         axes[1].plot(x2,y2)
```

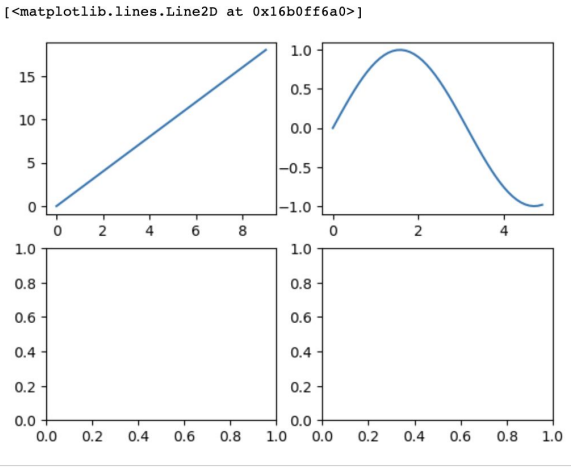
```
Out[35]: [<matplotlib.lines.Line2D at 0x16aec8820>]
```



Matrix type subplot

```
In [36]: fig, axes = plt.subplots(nrows=2, ncols=2)
         x1 = np.arange(10)
         y1 = x1*2
         x2 = np.arange(0,5,0.1)
         y2 = np.sin(x2)
         axes[0][0].plot(x1,y1)
         axes[0][1].plot(x2,y2)
```

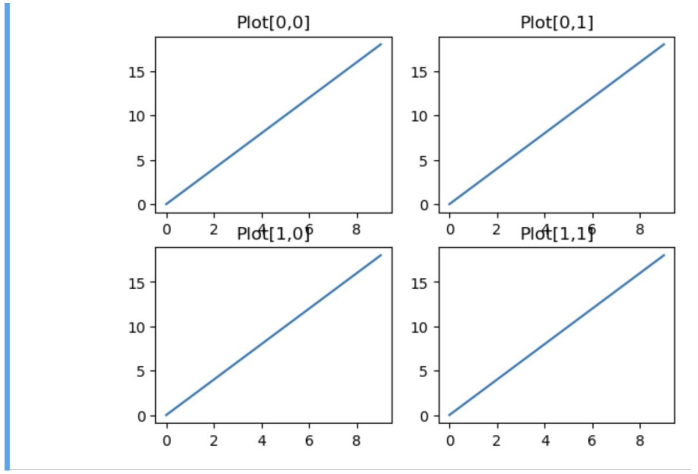
```
Out[36]: [<matplotlib.lines.Line2D at 0x16b0ff6a0>]
```



Matplotlib - Subplot

Because the axes object is an array / matrix, we can use a simple Python for loop to iterate on each chart and populate it with the relevant content. **For example →**

```
In [49]: fig, axes = plt.subplots(nrows=2, ncols=2)
         x = np.arange(10)
         y = x1*2
         shape = axes.shape
         for i in range(shape[0]):
             for j in range(shape[1]):
                 axes[i, j].plot(x, y)
                 axes[i, j].set_title(f'Plot[{i},{j}]')
```



We want to use the shape attribute because it returned a tuple represent the dimensions of the matrix

fig.tight_layout() → Will automatically adjust the space between the charts in the subplot.

fig.subplot_adjust() → Allow us to determine manually the space between the charts in the subplot. When using, we need to pass the **'left'**, **'right', 'top', 'bottom'** parameters for the direction of the space, and the **'wspace'**, **'hspace'** for height spacing or width spacing across all our chart grid.

Matplotlib - Subplot

fig.suptitle() → Allow us to add title above all of our charts in the subplot. We can also provide **'fontsize'** parameter to determine the size of the title.

```
In [57]: fig, axes = plt.subplots(nrows=2, ncols=2)
         x = np.arange(10)
         y = x1*2
         fig.suptitle('Figure Level', fontsize=15)
         fig.tight_layout()
         shape = axes.shape
         for i in range(shape[0]):
             for j in range(shape[1]):
                 axes[i, j].plot(x, y)
                 axes[i, j].set_title(f'Plot[{i},{j}]', fontsize=10)
```

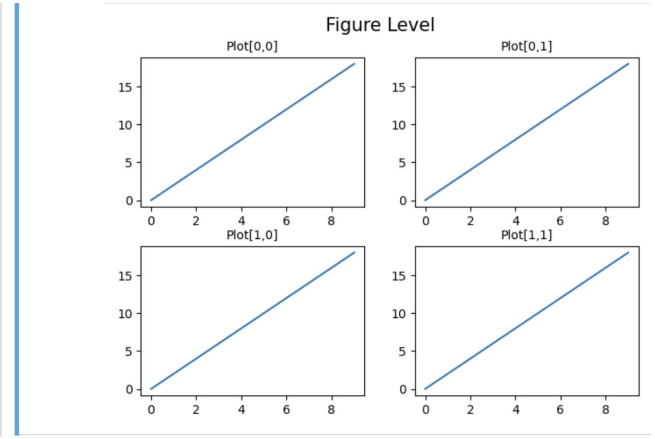


fig.savefig() → Allow us to save our subplot fig into a external file, by providing **'bbox_inches' = 'tight'** the file will also included all the titles and labels we added to our fig.

Matplotlib - Legends

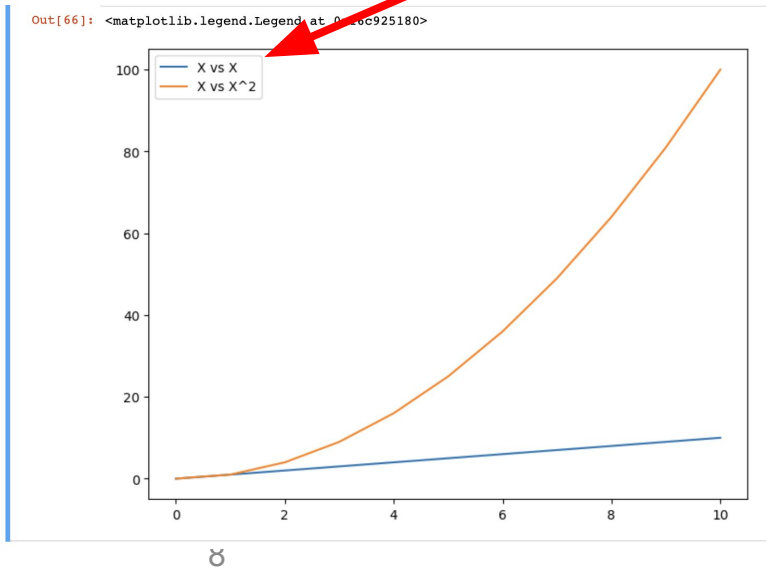
ax.legend() → In case we have multiple graphs on the same chart we most likely will want to add additional explanation about each chart content. In Matplotlib, **legends** provide us a way to add this additional information so our plots will become more understandable.

When providing a legend we also want to specify the labels the legend will use for each graph so when create each plot we need to provide the **'label'** attribute as well.

For example → Simple legend configuration:

```
In [62]: x = np.arange(0,11)
         fig = plt.figure()
         ax = fig.add_axes([0,0,1,1])
         ax.plot(x, x, label='X vs X')
         ax.plot(x, x**2, label= 'X vs X^2')
         ax.legend()
```

Chart simple legend



We provided label for each graph and called the legend() method

Matplotlib - Legends

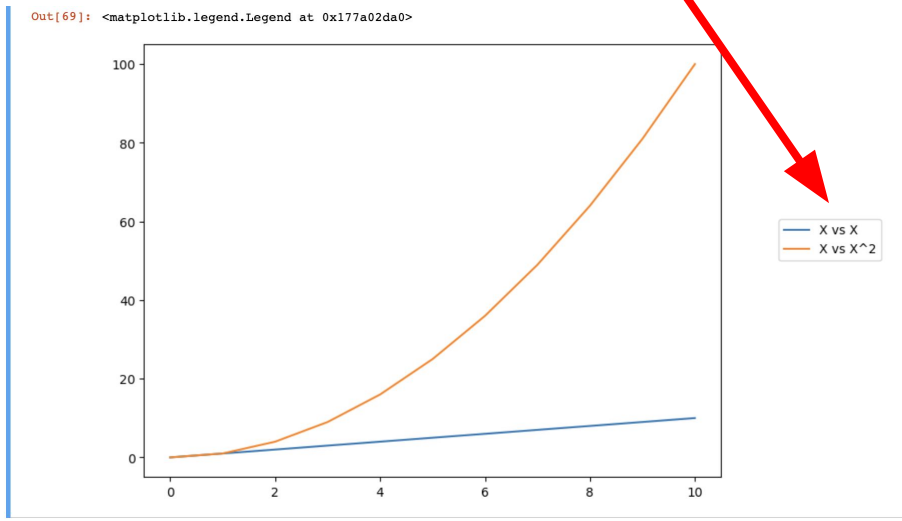
In addition, the legend() method provide us some parameters to control its default configuration and make it more customized:

'loc' → Parameter that determine where to locate the legend (loc - location), we should provide a list so the first number is the x axis location of the legend and the second number is the y axis location of the legend.

- **Numbers between 0 - 1** → Located inside the chart
- **Numbers above 1 or below 0** → Locate outside the chart

```
In [69]: x = np.arange(0,11)
         fig = plt.figure()
         ax = fig.add_axes([0,0,1,1])
         ax.plot(x, x, label='X vs X')
         ax.plot(x, x**2, label= 'X vs X^2')
         ax.legend(loc=(1.1,0.5))
```

The legend is outside the chart



Matplotlib - Plot Styling

Until now we only used the default plot styling but Matplotlib provide us some parameters that will enable us to customize our plot styling.

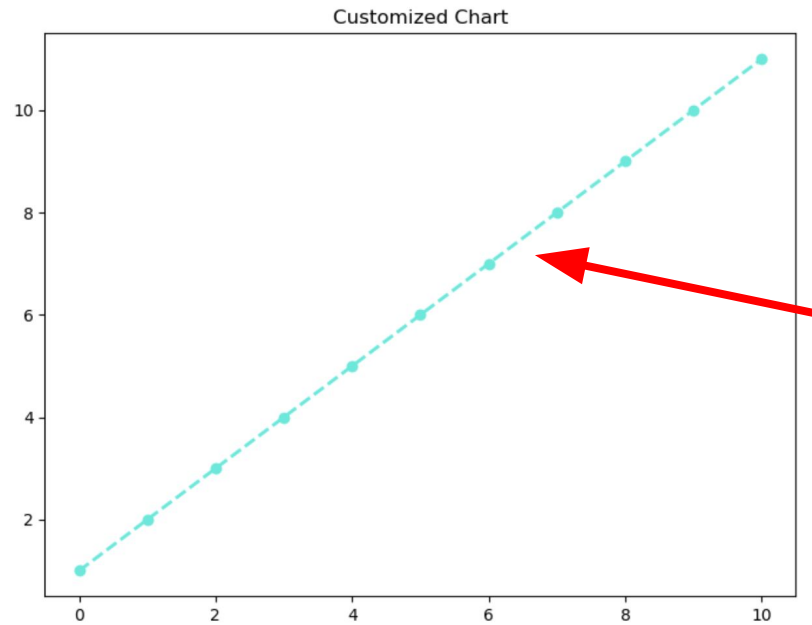
- **‘color’** → The color parameter determine the chart line color, we can pass colors in RGB or Hex.
- **‘lw’** → The lw parameter determine the width of the chart line (for example, 10 will be 10X of the default line width, 0.5 will be half of the default line width).
- **‘linewidth’** → Works and perform the same action as the ‘lw’ parameter.
- **‘ls’** → The ls parameter determine the line style of the chart, we can provide a customized style to the ‘ls’ parameter and Matplotlib will plot a chart with our customized line style.
- **‘linestyle’** → Works and perform the same action as the ‘ls’ parameter.
- **‘marker’** → The marker parameter allow us to add a dot mark in every real dot place.
  We need to understand that when plotting a graph the graph line itself looks continuous but it’s only appears like that because Matplotlib connecting the line between the real chart dots.
  We can choose from different types of marker styles such as: ‘o’, ‘+’, ‘>’, ‘<’, ‘-’
- **‘markersize’** → The markersize parameter determine the size of the marker point.

Matplotlib - Plot Styling

Let’s create a simple example of a customized plot chart using Matplotlib styling parameters:

```
In [83]: fig = plt.figure()
         ax = fig.add_axes([0,0,1,1])
         ax.set_title('Customized Chart')
         ax.plot(x, x+1, marker='o', lw=2, ls='--', color='#1FEBDD')
```

```
Out[83]: [<matplotlib.lines.Line2D at 0x28b6d4760>]
```



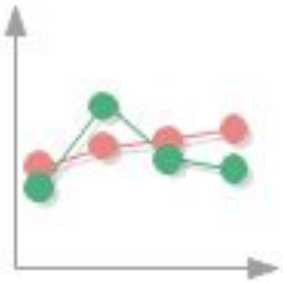
We created customized chart line with custom style, X2 width line size, with customized color and with marks on every real dot

Different Chart Types

Pie

Bar

Column



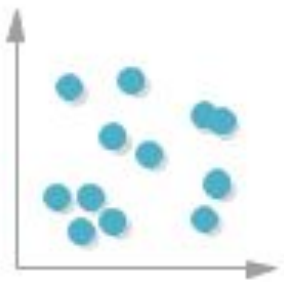
Line

Area

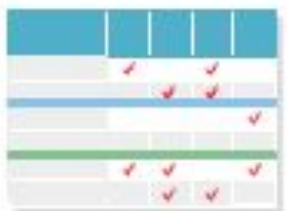
Doughnut

Bubble Chart

Spider and Radar



Scatter



Comparison Chart

Stacked bar chart

Gauges

Different Chart Types

Until now we only used the plot chart type (line chart) to learn about Matplotlib capabilities, but the library provide us a variety of chart types to choose from.

Before we start exploring the different chart types in Matplotlib we first need to understand what are the main characters of each chart type and for what it used for.

Line Chart → A line chart graphically displays data that changes continuously over time. Each line graph consists of points that connect data to show a trend (continuous change). Line graphs have an x-axis and a y-axis. In the most cases, time is distributed on the horizontal axis.

When should we use:

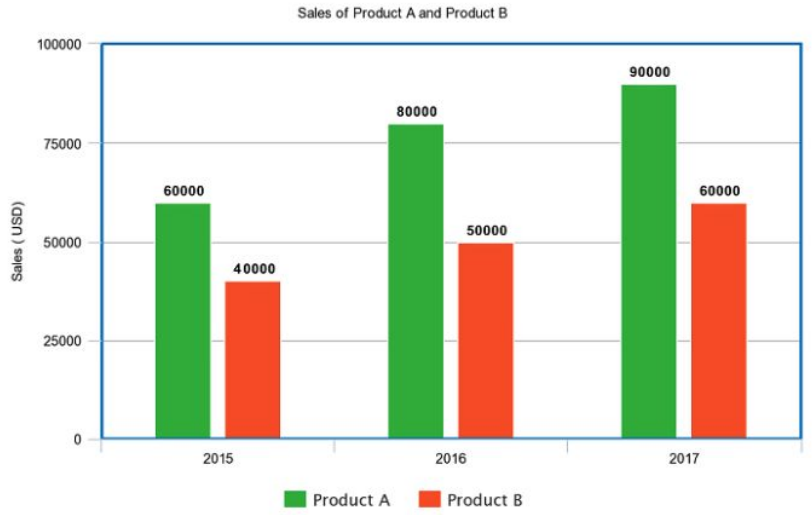
- When you want **to show trends**. For example, how house prices have increased over time.
- When you want **to make predictions** based on a data history over time.
- When **comparing** two or more different variables, situations, and information over a given period of time.

Different Chart Types

Bar Chart → Bar charts represent categorical data with rectangular bars. Bar graphs are among the most popular types of graphs and charts in economics, statistics, marketing, and visualization in digital customer experience. They are commonly used to compare several categories of data.

When should we use:

- **To compare data** among different categories.
- Bar charts can also show **large data changes** over time.
- Bar charts are ideal for visualizing the distribution of data when we have more than three categories.



Different Chart Types

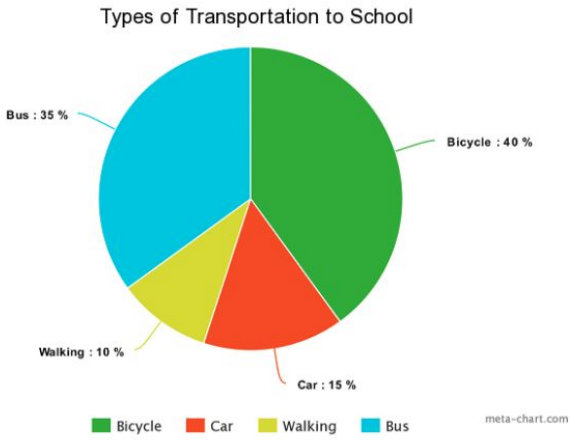
Pie Chart → Pie charts displays data and statistics in an easy-to-understand ‘pie-slice’ format and illustrates numerical proportion. Each pie slice is relative to the size of a particular category in a given group as a whole.

The pie chart brakes down a group into smaller pieces. It shows part-whole relationships.

To make a pie chart, you need a list of categorical variables and numerical variables.

When should we use:

- When you want to create and **represent the composition of something**.
- To **show percentage** or proportional data.
- When **comparing areas of growth** within a business such as profit.
- Pie charts work best for displaying data for **3 to 7 categories**.



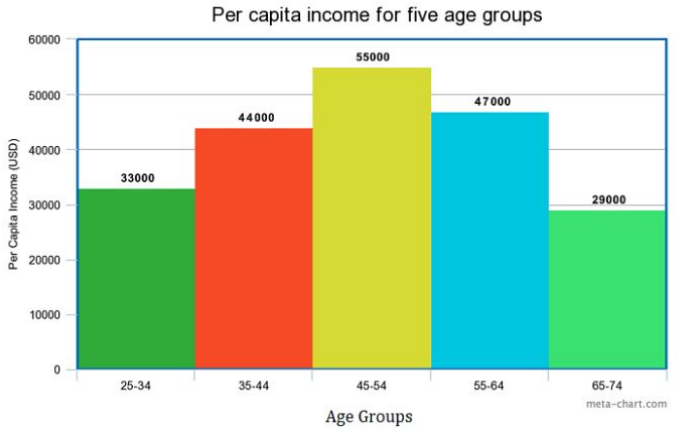
Different Chart Types

Histogram → Histogram shows continuous data in ordered rectangular columns. Usually, there are **no gaps** between the columns.

The histogram displays a frequency distribution (shape) of a data set. At first glance, histograms look alike to bar graphs. However, there is a key difference between them. Bar Chart represents categorical data and histogram represent continuous data.

When should we use:

- When the **data is continuous**.
- When you want to represent the shape of the **data's distribution**.
- When you want to see whether the outputs of two or more processes are different.
- To summarize **large data sets** graphically.
- To communicate the data distribution quickly to others.



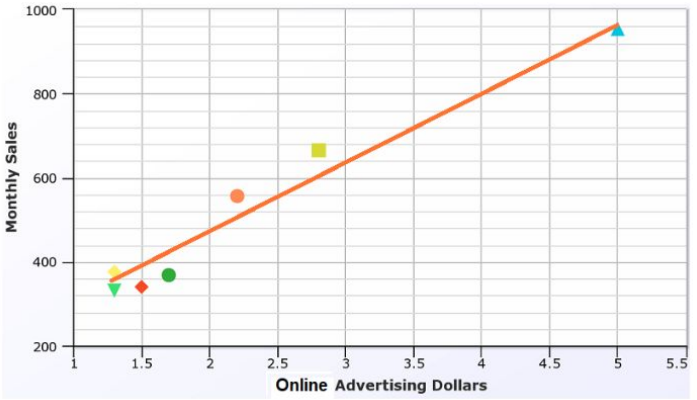
Different Chart Types

Scatter plot → The scatter plot is an X-Y diagram that shows a relationship between two variables. It is used to plot data points on a vertical and a horizontal axis. The purpose is to show how much one variable affects another. Usually, when there is a relationship between 2 variables, the first one is called independent. The second variable is called dependent because its values depend on the first variable.

Scatter plots also help you predict the behavior of one variable (dependent) based on the measure of the other variable (independent).

When should we use:

- When trying to find out whether there is a **relationship between 2 variables**.
- **To predict** the behavior of dependent variable based on the measure of the independent variable.
- When trying to identify the potential for problems.



Matplotlib - Different Chart Types

Now that we learned about the most commonly used charts, we need to know how we can create them using Matplotlib:

- **ax.plot()** → Generate a line chart, takes 2 data parameters, the first is the x axis and the second is the y axis.
- **ax.scatter()** → Generate a scatter chart, takes 2 data parameters the first is the independent data set and the second is the dependent data set.
- **ax.bar()** → Generate a bar chart, takes 2 data parameters, the first is the bar categories and the second is the values of each bar category.
- **ax.hist()** → Generate a histogram, take 1 data parameters representing the data set we want to put in the histogram chart, the other parameter is **‘bins’** and it determine the number of bins we want to generate in the histogram.

  More bins meaning the histogram is more detailed and accurate but harder to read and understand (more noise)

Matplotlib - Different Chart Types

For example → Let's create 2X2 subplot with line chart, scatter chart, bar chart and histogram:

First let's set the random seed to 0 so everyone will get the same chart results →

By running this command: **np.random.seed(0)**

Now, for the **line chart data** we will generate the following data set:

```
In [23]: x = np.arange(10)
         y_linear = x*2
         print(x)
         print(y_linear)

         [0 1 2 3 4 5 6 7 8 9]
         [ 0  2  4  6  8 10 12 14 16 18]
```

For the **scatter chart data** we will generate the following data set:

```
In [24]: x = np.arange(10)
         y_scatter = np.random.randn(10)
         print(x)
         print(y_scatter)

         [0 1 2 3 4 5 6 7 8 9]
         [ 1.76405235  0.40015721  0.97873798  2.2408932   1.86755799 -0.97727788
           0.95008842 -0.15135721 -0.10321885  0.4105985 ]
```

Matplotlib - Different Chart Types

For the **bar chart data** we will generate the following data set:

```
In [25]: bar_categories = ['A', 'B', 'C', 'D', 'E']
         bar_values = np.random.rand(5)
         print(bar_categories)
         print(bar_values)

         ['A', 'B', 'C', 'D', 'E']
         [0.79172504 0.52889492 0.56804456 0.92559664 0.07103606]
```

For the **histogram chart data** we will generate the following data set:

```
In [21]: y_hist = np.random.randn(1000)
         print(y_hist)

         [-1.00158491 -1.36714098  0.32427164  1.29684299  1.30390745 -0.62954069
           1.12508588  1.02068783 -0.769062    0.96548615  1.91008636  2.12613731
           0.88351176 -0.66880584  0.90889269  1.81206281 -0.25648042  0.15793198
          -1.51757763  0.08734574  0.94419316 -1.05582611  0.1276651  -2.96165033
           2.11029984  0.58067004 -0.73499208 -0.28586443 -0.92654173 -0.50708301
          -1.87769771  0.57921828  1.46035095  1.49302147 -0.75766303  1.07737976
          -1.18589639 -0.5337765   0.27636542 -0.00724315 -1.40883614  0.17466596
           1.13951624  1.37204521 -0.35610082 -0.55979931 -0.26632488 -0.31922338
          -0.2980101   0.12268824 -1.15213442 -1.05028381  0.85281083 -0.83374729
           0.00551354  0.08081324 -0.13748082  0.59067046 -0.20894054 -0.90083778
```

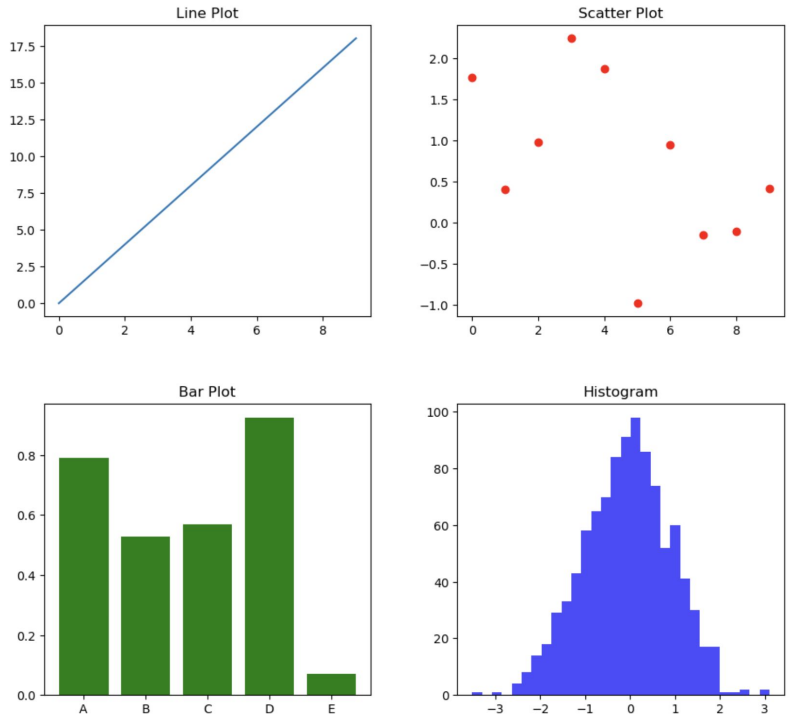
ECOM SCHOOL

המכללה למקצועות הדיגיטל וההייטק

Matplotlib - Different Chart Types

Once we have all charts data ready, we can configure the subplot and populate each chart with the relevant chart type:

```
In [31]: fig, ax = plt.subplots(2, 2, figsize=(10, 10))

         # Line Plot
         ax[0, 0].plot(x, y_linear)
         ax[0, 0].set_title('Line Plot')

         # Scatter Plot
         ax[0, 1].scatter(x, y_scatter, color='r')
         ax[0, 1].set_title('Scatter Plot')

         # Bar Plot
         ax[1, 0].bar(bar_categories, bar_values, color='g')
         ax[1, 0].set_title('Bar Plot')

         # Histogram
         ax[1, 1].hist(y_hist, bins=30, color='b', alpha=0.7)
         ax[1, 1].set_title('Histogram')

         # Set the title for the figure
         fig.suptitle('4 Types of Matplotlib Plots', fontsize=16)

         plt.tight_layout()
         plt.show()
```



Class Exercise - Matplotlib Charts

Instructions:

For this exercise use the **matplotlib_charts_class_exc.csv** file, load the file into a proper Dataframe.

Create Matplotlib charts that represent the following:

- Create a line chart representing the trend of 'GDP per Capita' over the years for a country of your choice. Ensure the line chart has a title and labelled axes.
- Using a histogram, visualize the distribution of the 'GDP per Capita' values for a particular year of your choice.
- Plot a bar chart to compare the 'GDP per Capita' for five different countries in the year 2010.
- Using a scatter plot, explore the relationship (if any) between the year and 'GDP per Capita' for a chosen country.

Class Exercise Solution - Matplotlib Charts