# **Module 16 -**Data Visualization

Contents

[**Module 16 -**Data Visualization 1](#_Toc216014136)

[Data Visualization 1](#_Toc216014137)

[Matplotlib 1](#_Toc216014138)

[Plotly 2](#_Toc216014139)

[Seaborn 2](#_Toc216014140)

[Pandas 2](#_Toc216014141)

[**Panda Series** 4](#_Toc216014142)

[NumPy 5](#_Toc216014143)

[Working with Datasets 7](#_Toc216014144)

[Reference and Additional Materials: 14](#_Toc216014145)

## Data Visualization

The process of representing trends and correlations in our data visually is called [Data Visualization](https://www.simplilearn.com/tutorials/python-tutorial/data-visualization-in-python). Data visualization is the heart of data analysis. Making plots and static or interactive visualizations is one of the most important tasks for data analysis. Data visualization is more effective than plain numbers at providing an overview or a summary of data. It helps us understand the underlying structure of a dataset. To perform data visualization in python, we can use various data visualization [libraries](https://www.geeksforgeeks.org/libraries-in-python/) such as Matplotlib, Seaborn, etc. Instead of writing a lot of complex code to draw a graph, you can use a few commands from one of these libraries. Each library has its own features and advantages.

## Matplotlib

[Matplotlib](https://www.geeksforgeeks.org/python-introduction-matplotlib/) is a 2D plotting library that was initially released in 2003 and it's the most popular and widely used library in the Python community. It comes with an interactive environment across multiple platforms. You can use [Matplotlib](https://matplotlib.org/cheatsheets/_images/handout-beginner.png) to create [plots](https://www.w3schools.com/python/matplotlib_plotting.asp), [bar charts](https://www.w3schools.com/python/matplotlib_bars.asp), [pie charts](https://www.w3schools.com/python/matplotlib_pie_charts.asp), [histograms](https://www.w3schools.com/python/matplotlib_histograms.asp), [scatterplots](https://www.w3schools.com/python/matplotlib_scatter.asp), [error charts](https://www.geeksforgeeks.org/errorbar-graph-in-python-using-matplotlib/), [power spectral density (PSD)](https://matplotlib.org/stable/gallery/lines_bars_and_markers/psd_demo.html), [subplots](https://www.w3schools.com/python/matplotlib_subplot.asp), and many others. To get started with Matplotlib, you can easily install it by following the instructions [here](https://www.w3schools.com/python/matplotlib_getting_started.asp).

**Using matplotlib for plotting the series**

The [plot()](https://www.w3schools.com/python/matplotlib_plotting.asp) function in matplotlib is used to draw points (markers) in a diagram. Parameter 1 is an array containing the points on the x-axis. Parameter 2 is an array containing the points on the y-axis. If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

xpoints = np.array([1, 8])

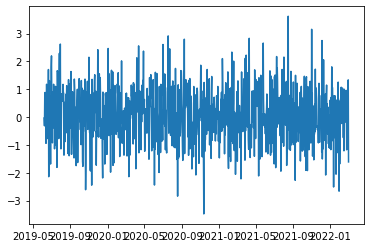
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints)

plt.show()

**Calling the plot operation using matplotlib**

plt.plot(df)



## Plotly

[Plotly](https://www.geeksforgeeks.org/python-plotly-tutorial/) can be used to create web-based data visualizations that can be displayed in Jupyter notebooks or web applications using Dash or saved as individual HTML files.

Plotly provides more than 40 unique [chart types](https://plotly.com/python/) like [scatter plots](https://www.geeksforgeeks.org/python-plotly-tutorial/#3d-scatter-plot-plotly:~:text=Plotly%20in%20Python-,3D%20Scatter%20Plot%20Plotly,-3D%20Scatter%20Plot), histograms, [line charts](https://www.geeksforgeeks.org/python-plotly-tutorial/#:~:text=Output%3A-,3D%20Line%20Plots,-Line%20plot%20in), bar

charts, pie charts, error bars, box plots, multiple axes, sparklines, dendrograms, 3-D charts, and more. Plotly also provides [contour plots](https://plotly.com/python/contour-plots/), which are not that common in other data visualization libraries. You can start with Plotly by installing it with the instructions provided [here](https://www.geeksforgeeks.org/download-and-install-pip-latest-version/).

## Seaborn

[Seaborn](https://www.w3schools.com/python/numpy/numpy_random_seaborn.asp) is a Python data visualization library that is based on Matplotlib and closely integrated with the NumPy and pandas data structures. [Seaborn](https://www.geeksforgeeks.org/introduction-to-seaborn-python/) has various dataset-oriented plotting functions that operate on data frames and arrays that have whole datasets within them. It is a high-level interface for creating beautiful and informative statistical graphics that are integral

to explore and understand data. The Seaborn data graphics can include [bar charts](https://seaborn.pydata.org/generated/seaborn.barplot.html), pie charts, [histograms](https://seaborn.pydata.org/generated/seaborn.histplot.html), scatterplots, error charts, etc. The Seaborn package is imported as sns and by using [load\_dataset](https://www.geeksforgeeks.org/python-seaborn-load_dataset-method/) function we can load any [dataset](https://www.geeksforgeeks.org/seaborn-datasets-for-data-science/) from the seaborn library. There are numerous different datasets available in seaborn that can be accessed directly by load\_dataset. You can get started by installing Seaborn from their website [here](https://seaborn.pydata.org/installing.html).

## Pandas

[Pandas](https://www.w3schools.com/python/pandas/default.asp) is not a data visualization library like the others we have discussed but a data analysis and manipulation library. Even though it's not a data visualization library, it's capable of creating basic plots. It has functions for analyzing, cleaning, exploring, and manipulating data. Pandas can clean messy data sets and make them readable and relevant. If you are just creating plots for exploratory data analysis, Pandas might be highly useful and practical. It makes it easy to manipulate data in tabular form. The [various functions](https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf) of Pandas constitute a powerful and versatile data analysis tool. You can start working with Pandas by using the instruction to install it [here](https://www.geeksforgeeks.org/how-to-install-python-pandas-on-windows-and-linux/).

**Pandas DataFrame**

A [DataFrame](https://www.geeksforgeeks.org/python-pandas-dataframe/) is the core data structure in Pandas. It a 2D tabular data structure with labeled rows and columns, similar to an Excel spreadsheet or a SQL table.The following line of code would be the syntax for a DataFrame:

pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)

Index and columns are optional and would hold row labels and column labels respectively. Data holds the actual data. It could be a [list](https://youtu.be/jk7iKthzMHA?si=A3Hu_jhACjda-Eyo), [dictionary](https://youtu.be/8Pcnn-HIWGk?si=Spt5kVL8ezKsvs63), [NumPy array](https://youtu.be/-5dJk3WnaOo?si=ekZo6eP2iahdoj8g), etc.

**Creating a DataFrame from a Dictionary**

import pandas as pd

week = {

'Days': ["Monday", "Tuesday", "Wednesday"],

'index': [1, 2, 3]

}

tab = pd.DataFrame(week)

print(tab)

Output:

Days index

0 Monday 1

1 Tuesday 2

2 Wednesday 3

**Creating a DataFrame from a List**

courses = ['DP I', 'DP II', 'Web page Development', 'Data Base']

df = pd.DataFrame(courses)

print(df)

Output:

0

0 DP I

1 DP II

2 Web page Development

3 Data Base

### **Panda Series**

A [Pandas Series](https://www.w3schools.com/python/pandas/pandas_series.asp) is a one dimensional labeled array that can hold any kind of data. Each item in a [Series](https://youtu.be/sZfm63OjWGE?si=WfiN6CDelR5Ks5li) has a label called an index. An index helps access and organize data easier.

x = [10, 20, 30, 40]

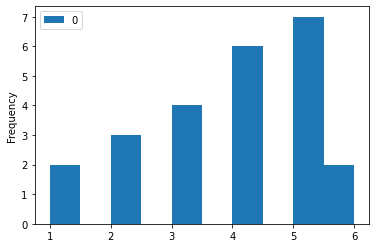
ser = pd.Series(x)

print(ser)

**Pandas Histogram**

The plot function of pandas provides different kinds of plots which can be set by "kind" parameter. In this case we assign ‘hist’ to kind to create a histogram.

df1.plot(kind='hist')



**The date\_range() function**

The date\_range() function in Pandas is used to create a sequence of dates. This is useful when you're working with time series data or want to generate a timeline for your DataFrame.

Syntax:

pandas.date\_range(start=None, end=None, periods=None, freq=None, tz=None, normalize=False, name=None, closed=None)

Here’s what those parameters mean:

* start: The starting date of the range.
* end: The ending date of the range.
* periods: The number of dates to generate.
* freq: The frequency between each date (e.g., 'D' for daily, 'H' for hourly, '5H' for every 5 hours, etc.).
* tz: Time zone (optional)
* normalize: If True, the start and end dates will be set to midnight.
* name: Name of the resulting date index (optional)
* closed: Choose if the interval should include the start ('left'), the end ('right'), or both (None means include both by default).

In this example below, we’re creating a DataFrame in Pandas with 1000 rows of random data. First, we use [randn](https://www.w3schools.com/python/numpy/numpy_random.asp)(1000) from NumPy to generate 1000 random numbers. Then, we use pd.date\_range() to create a sequence of 1000 dates starting from June 7, 2019. These dates are used as the index of the DataFrame, meaning each row is labeled with a date. The columns=['value'] part just gives a name to the column of random numbers. Finally, df.head(10) shows us the first 10 rows of the DataFrame so we can see what it looks like

from numpy.random import randn, randint, uniform, sample

#date\_range we have to give initial data and in periods we have to number of dates

df = pd.DataFrame(randn(1000), index = pd.date\_range('2019-06-07', periods = 1000), columns=['value'])

df.head(10)

## NumPy

[Numpy](https://www.geeksforgeeks.org/python-numpy/), short for Numerical Python, is a powerful Python library used for working with arrays and performing complex mathematical operations efficiently. While Python has built-in lists, they are slower and less efficient for numerical tasks. Numpy introduces a special data structure called [ndarray](https://www.w3schools.com/python/numpy/numpy_creating_arrays.asp) (n-dimensional array), which stores data in a single continuous block of memory. This allows for much faster processing and manipulation, especially when working with large data sets. With NumPy, you can easily perform mathematical and logical operations on arrays, work with linear algebra, reshape data, generate random numbers, and so much more. These [built-in functions](https://www.codecademy.com/resources/docs/numpy/built-in-functions) make it a key tool for scientific and numerical computing. NumPy is often used alongside other popular libraries like [SciPy](https://www.w3schools.com/python/scipy/scipy_intro.php) and Matplotlib. Together, they form a powerful Python-based alternative to [MATLAB](https://www.mathworks.com/discovery/what-is-matlab.html), which is widely used in engineering and data science but it requires a paid license unlike NumPy. NumPy is completely open source and free to use. To get started with NumPy, you can easily install it by following the instructions on the [official website](https://numpy.org/install).

**The cumsum() function**

The [cumsum() function](https://www.geeksforgeeks.org/numpy-cumsum-in-python/) returns the cumulative sum of the elements along a given axis.

a = np.array([[1,2,3], [4,5,6]])

np.cumsum(a)

## Working with Datasets

Try these examples on your own and feel free to explore everything python libraries can achieve.

Don’t forget to install and import the libraries you will need.

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

**IRIS Dataset**

The [IRIS dataset](https://www.geeksforgeeks.org/iris-dataset/) is a well known and widely used dataset in Python and data science for practicing machine learning and data analysis. It contains information about 150 different flowers from three species of iris. Each flower is described by four features: sepal length, sepal width, petal length, and petal width. There are many other types of datasets that are available in seaborn that can be accessed directly by load\_dataset. Load IRIS dataset from seaborn library load\_dataset and this will get the dataset from the library.

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

iris.head(10)

The first line of code loads the dataset while the second shows the first 10 rows of the dataset. You could put any valid number in to replace 10, but we stick with 10 for now just to get a quick look at the data. Now we can use this dataset to practice numerous techniques in data visualization.

**Labeling a Line Graph**

We can label the graph using the title parameter, set\_xlabel will set the label for x axis and set\_ylabel will set the label for Y axis.

ax = iris.plot(figsize=(10,10), title='Iris Dataset plotting')

ax.set\_xlabel('X\_cooridnate')

ax.set\_ylabel('Y\_coordinate')

This should output a graph that has the title ‘Iris Dataset plotting’ and coordinates named ‘X\_cooridnate’ and ‘Y\_coordinate’.

**Drop a Column**

In case if we want to drop any column from plotting, we can call the drop function as follows:

df = iris.drop(['species'], axis = 1)

df

This code drops the column named ‘species’ and outputs the result in a table.

This code drops the column named ‘petal\_width’ and outputs the result in a table. Notice that a different value is assigned to this block of code (df1) so the ‘species’ column you dropped above will appear in this data.

df1 = iris.drop(['petal\_width'], axis = 1)

df1

Now you can plot this data:

ax = df1.plot(figsize=(10,10), title='Iris Dataset')

This will output a graph without the column ‘petal\_width’ column we dropped from df1.

**Select and Plot Particular Row**

Selecting and plotting particular rows from a dataset can be helpful when you want to focus on specific data points like comparing one flower's measurements in the IRIS dataset to another. It allows you to analyze individual examples in more detail which could be great for understanding outliers or testing out visualization on a smaller scale before applying them to the whole dataset. The two lines of code below will select the row specified and output the data.

#select particular row based on the index values

df.iloc[0]

df.iloc[149]

This line below will create a bar chart of the information on the row specified.

# to plot the particular row

df.iloc[0].plot(kind='bar')

This does the exact same thing as the line above, .bar() is just a simpler way to write it instead of using kind='bar'. Both are valid and give the same result.

#bar is function that is subset of plot which can be called instead of kind parameter

df.iloc[0].plot.bar()

**Histogram Plot**

Histogram plots are useful when you want to see the distribution of values in your data. This will help you understand the shape of the data. One way of plotting a histogram is the line of code below:

iris.plot.hist()

This creates histograms for all the numeric columns in the Iris dataset. Each feature (like sepal length, petal width, etc.) will be shown in the same plot to give you an overall idea of how the values are spread. We can also use the line below to accomplish the same task.

iris.plot(kind = 'hist')

You can also look at just one set of data within the data if you wanted. The code below creates a histogram just for the sepal length column. It shows how many flowers have certain sepal lengths, making it easier to focus on that one feature.

iris['sepal\_length'].plot.hist()

Df1, like we discussed above, is a filtered version of the IRIS dataset. The code below will create a histogram for all columns in df1 just like it did for IRIS.

df1.plot.hist()

**Stacked Bar**

A stacked bar plot is a kind of bar graph in which each bar is visually divided into sub bars to represent multiple column data at once. To plot the Stacked Bar plot we need to specify stacked=True in the plot method. We can also pass the list of colors as we need to color each sub bar in a bar.

The code below creates a stacked histogram for all the numeric columns in the Iris dataset. Each column's values are stacked on top of each other for each bin (range of values), giving you a clearer idea of how the features add up together. In this case, bins is set to 150 since there are 150 rows of data in the IRIS dataset.

iris.plot(kind = 'hist', stacked = True, bins = 150)

Next is a regular, non-stacked histogram with the same number of bins. Each feature is overlaid in the same plot but not stacked, so it may be harder to tell them apart.

iris.plot(kind = 'hist', bins = 150)

Lastly, a horizontal stacked histogram, which flips the bars so they go left to right instead of up and down. It's useful when you want a different layout or when labels on the x-axis are too long.

#horizontal graph

iris.plot(kind = 'hist', stacked = True, bins = 50, orientation = 'horizontal')

You can also add a list of colors if you want to style each sub bar differently.

**Tracking Changes in Data Using diff()**

The [diff() function](https://www.w3schools.com/python/pandas/ref_df_diff.asp) calculates the difference of a DataFrame element compared with another element in the DataFrame. Basically, the difference between the previous data and current data. It’s helpful when you want to see how much a value has increased or decreased compared to the previous one.

The line of code below will show the difference between each sepal width value and the one before it. If a value goes up or down, this function tells you by how much.

iris['sepal\_width'].diff()

This next line will create a stacked histogram showing how much the sepal width values change from row to row. It's a great way to see if most changes are small, large, or stay the same.

iris['sepal\_width'].diff().plot(kind = 'hist', stacked = True, bins = 50)

If you will recall, we removed the ‘species’ column from df earlier. In this next line, diff() is applied to the rest of the numeric columns and .head() will show the first five rows of differences.

df = iris.drop(['species'], axis = 1)

df.diff().head()

This last line will create histograms for all the differences across the numeric columns. The color='b' makes the bars blue, and alpha=0.8 sets the transparency so overlapping bars can still be seen. figsize=(10,10) adjusts the size of the entire plot window.

#color specifies the color value and alpha denotes the transparency of the color

df.diff().hist(color = 'b', alpha = 0.8, figsize=(10,10))

**Scatter Plot**

A scatter plot is a type of graph that helps you see how two things are related. Each dot on the graph stands for one row of data, with one value on the x-axis and the other on the y-axis. It’s really helpful when you want to look for patterns, trends, or groups in your data.

In the scatter plot we have to give the sepal\_length in x axis and petal\_length in y axis.

iris.plot.scatter(x = 'sepal\_length', y = 'petal\_length')

This code below will accomplish the same as the previous plot, but also colors each point based on sepal width, which adds a third variable to the plot.

iris.plot.scatter(x = 'sepal\_length', y = 'petal\_length', c = 'sepal\_width')

This next line adds a label to the graph elements, which can help make the plot easier to understand, especially if you're using a legend.

#label parameter will take the labelling for the graph element

iris.plot.scatter(x = 'sepal\_length', y = 'petal\_length', label = 'Length');

This next line changes the variables being compared and also sets the color to green using color='g'.

iris.plot.scatter(x = 'sepal\_width', y = 'petal\_width', label = 'Width', color = 'g')

This version changes the size of the points using s=100, making the dots larger. It still uses color (c='sepal\_width') to show an extra dimension of data.

#s denotes the diameter of the plotting elements

iris.plot.scatter(x = 'sepal\_length', y = 'petal\_length', c = 'sepal\_width', s = 100)

**Hexbin**

A hexbin plot is similar to a scatter plot, but instead of plotting individual points, it groups them into hexagon-shaped bins. This is especially useful when you have a lot of overlapping data points, making it hard to see patterns in a regular scatter plot. Grid size denotes the size of the hexbin, smaller values have big hexagons and larger values will create smaller hexagons.

#you can change the shape of the plot

iris.plot.hexbin(x = 'sepal\_length', y = 'petal\_length', gridsize = 15, C = 'sepal\_width')

This makes a hexbin plot with sepal length on the x-axis and petal length on the y-axis. The gridsize=15 controls how big or small the hexagons are—a smaller number gives you bigger hexagons, while a larger number makes them smaller and more detailed. The C='sepal\_width' part changes the color of each hexagon based on the average sepal width of the points inside it. Darker or brighter colors mean a higher value.

**Pie Chart**

A pie chart is used to show how parts make up a whole. Each "slice" of the pie represents a different category or value, and the size of each slice shows how much it contributes to the total.

df = iris.drop(['species'], axis = 1)

d = df.iloc[0]

d.plot.pie(figsize = (10,10))

This code first removes the ‘species’ column from the Iris dataset since pie charts are used for numerical data generally and species is categorical. Then, the first row of the dataset is selected which contains the measurements for one flower. The last line creates the pie chart using the values defined in the lines above it.

**Creating and Customizing Pie Chart Subplots**

Subplots allow you to display multiple charts in one figure, making it easy to compare different data sets side by side. In this case, subplots are created for the different elements of the data to show how they each contribute to the whole.

The line below will create subplots for each value in row d and displays them as pie charts.

#subplots will be created for index 0,1 and 2

d.plot.pie(subplots = True, figsize = (20, 20))

This next line is another version of the original subplots above. This version customizes the pie charts by setting the font size (fontsize=10) and adding percentage values to each slice with autopct='%.2f', which shows the percentage of each slice to two decimal places.

d.plot.pie(subplots = True, figsize = (20, 20), fontsize = 10, autopct = '%.2f')

In this last block of code, a new series is created with 5 equally distributed values (all 0.2) and labeled with 'a', 'b', 'c', 'd', and 'e'. This is then plotted as a pie chart. The name='Pie Plot' adds a title to the chart.

#adding index and title to pie plot

series = pd.Series([0.2]\*5, index = ['a','b','c', 'd','e'], name = 'Pie Plot')

series.plot.pie()

You can create subplots with all different types of plots and charts.

**Matrix Plot**

A matrix plot is useful when you want to see the relationships between all pairs of features in a dataset. Instead of plotting one feature against another individually, a matrix plot shows all of them together. This can help you spot patterns and see how the different features interact with each other.

First, you will need to import the library to help create a matrix plot.

from pandas.plotting import scatter\_matrix

This next command will give us information on how to use scatter\_matrix. It will show all available options and what each parameter does.

help(scatter\_matrix)

Next, you can plot the matrix plot. Try both plots below to see the differences.

#plotting all column vs all column

scatter\_matrix(df, figsize= (8,8), color = 'r', alpha=0.5)

plt.show()

#plotting all column vs all column

scatter\_matrix(df, figsize= (8,8), diagonal='kde', color = 'r', alpha=0.5)

plt.show()

## Reference and Additional Materials:

<https://matplotlib.org>

<https://www.geeksforgeeks.org/libraries-in-python/>

<https://www.simplilearn.com/tutorials/python-tutorial/>

<https://www.geeksforgeeks.org/python-plotly-tutorial/>

<https://plotly.com/python/>

<https://seaborn.pydata.org/>