19AD501 - DATA SCIENCE & ANALYTICS

UNIT IV

Data Munging: Introduction to Data Munging

Data Munging also referred as Data wrangling can be defined as the process of cleaning, organizing, and transforming raw data into the desired format for analysts to use for prompt decision-making.

Often, data munging occurs as a precursor to data analytics or data integration. High-quality data is essential for sophisticated data operations. The munging process typically begins with a large volume of raw data. Data scientists will mung the data into shape by removing any errors or inconsistencies.

They will then organize the data according to the destination schema so that it's ready to use at the endpoint. Munging is generally a permanent data transformation process.

It further includes data aggregation, data visualization, and training statistical models for prediction. Data wrangling is one of the most important steps of the data science process.

The quality of data analysis is only as good as the quality of data itself, so it is very important to maintain data quality.

The modern data munging process now involves six main steps:
1. Discover: First, the data scientist performs a degree of data exploration. This is a first glance at the data to establish the most important patterns. It also allows the scientist to identify any major structural issues, such as invalid data formats.
2. Structure: Raw data might not have an appropriate structure for the intended usage. The data scientists will organize and normalize the data so that it's more manageable. This also makes it easier to perform the next steps in the munging process.
3. Clean: Raw data can contain corrupt, empty, or invalid cells. There may also be values that require conversions, such as dates and currencies. Part of the cleaning operation is to ensure there's consistency across all values. For instance, the state in a customer's address might appear as Texas, Tex, or TX. The cleaning process will standardize this value for every address.
4. Enrich: Data enrichment is the process of filling in missing details by referring to other data sources. For example, the raw data might contain partial customer addresses. Data enrichment lets you fill in all address fields by looking up the missing values elsewhere, such as in the CRM database or a postal records lookup.
5. Validate: Finally, it's time to ensure that all data values are logically consistent. This means checking things like whether all phone numbers have nine digits, that there are no

numbers in name fields, and that all dates are valid calendar dates. Data validation also involves some deeper checks, such as ensuring that all values are compatible with the specified data type.

6. Publish: When the data munging process is complete, the data science team will push it towards its final destination. Often this is a data repository, where it will integrate with data from other sources. This will make the munged data permanently available to all consumers.

Data munging deals with the following functionalities.

1. **Data exploration**: Visualization of data is made to analyze and understand the data.
2. **Dealing with missing values**: Having Missing values in the data set has been a common issue when dealing with large data set and care must be taken to replace them. It can be replaced either by mean, mode or just labelling them as NaN value.
3. **Reshaping data**: Here the data is either modified from the addressing of pre-existing data or the data is modified and manipulated according to the requirements.
4. **Filtering data**: The unwanted rows and columns are filtered and removed which makes the data into a compressed format.
5. **Others:** After making the raw data into an efficient dataset, it is bought into useful for data visualization, data analyzing, training the model, etc.

Some examples of basic data munging tools are:

- Spreadsheets / Excel Power Query - It is the most basic manual data wrangling tool
- OpenRefine - An automated data cleaning tool that requires programming skills
- Tabula – It is a tool suited for all data types
- Google DataPrep – It is a data service that explores, cleans, and prepares data
- Data wrangler – It is a data cleaning and transforming tool

Data Pipeline and Machine Learning in Python

A machine learning pipeline is used to help automate machine learning workflows. They operate by enabling a sequence of data to be transformed and correlated together in a model that can be tested and evaluated to achieve an outcome, whether positive or negative.

Machine learning (ML) pipelines consist of several steps to train a model. Machine learning pipelines are iterative as every step is repeated to continuously improve the accuracy of the model and achieve a successful algorithm.

The main objective of having a proper pipeline for any ML model is to exercise control over it. A well-organised pipeline makes the implementation more flexible.

A typical machine learning pipeline would consist of the following processes:

- Data collection

- Data cleaning
- Feature extraction (labelling and dimensionality reduction)
- Model validation
- Visualisation

The machine learning data pipeline helps identify patterns in given data, which leads businesses to better decision-making. The machine learning pipeline boosts the machine learning model's performance leading to more efficient model deployment and better management of the models.

ML Pipeline Architecture

There are various stages in a machine learning pipeline architecture, mainly- **Data preprocessing, Model training, Model evaluation, and Model deployment**. Each stage of the data pipeline passes processed data to the next step.

Data Preprocessing- This step entails collecting raw and inconsistent data selected by a team of experts. The pipeline processes the raw data into an understandable format. Data processing techniques include feature extraction, feature selection, dimensionality reduction, sampling, etc. The final sample used for training and testing the model is the output of data preprocessing.

Model Training- Selecting an appropriate machine learning algorithm for model training is crucial in a machine learning pipeline architecture. A mathematical algorithm specifies how a model will detect patterns in data.

Model Evaluation- The sample models are trained and tested on historical data to make predictions and choose the best-performing model for the next step.

Model Deployment- The final step is to deploy the machine learning model to the production line. Ultimately, the end-user can obtain predictions based on real-time data.

How do Machine Learning Pipeline Tools Benefit Businesses?

Accurate Machine Learning Models- It creates better machine learning models that will generate more accurate predictions.

Faster Deployment- Data pipeline automation accelerates the process of training, testing, and refining machine learning models, allowing you to deploy them sooner in the market.

Enhanced Business Forecasting- You may improve your business forecasting abilities by using data pipeline technologies that help you construct a better machine learning model. Improved business forecasting enables you to stay ahead of the competition, provide a better client experience, and reap business profits.

Popular tools used in building an end-to-end machine learning pipeline-

MLFlow

MLflow is a free and open-source tool for managing machine learning workflow, including experimentation, production, deployment, and a centralized model repository.

DVC

Data Version Management, or DVC, is an experimental tool that helps define your pipeline irrespective of the programming language used

Neptune

Neptune is a machine learning metadata repository designed for monitoring various experiments by research and production teams.

Polyaxon

Polyaxon is a Kubernetes machine learning platform for recreating and managing machine learning workflows.

Data Visualization Using Matplotlib

Matplotlib is a cross-platform, open source, data visualization and graphical plotting library for Python and its numerical extension NumPy. As such, it offers a viable open source alternative to MATLAB. Developers can also use matplotlib's APIs (Application Programming Interfaces) to embed plots in GUI applications.

A Python matplotlib script is structured so that a few lines of code are all that is required in most instances to generate a visual data plot. The matplotlib scripting layer overlays two APIs:

- The **pyplot** API is a hierarchy of Python code objects topped by **matplotlib.pyplot**
- An **OO** (Object-Oriented) API collection of objects that can be assembled with greater flexibility than pyplot. This API provides direct access to Matplotlib's backend layers.

Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. It was introduced by John Hunter in the year 2002.

Matplotlib consists of several plots like **line, bar, scatter, histogram** etc.

Matplotlib and Pyplot in Python

matplotlib.pyplot.figure: Figure is the top-level container. It includes everything visualized in a plot including one or more Axes.

matplotlib.pyplot.axes: Axes contain most of the elements in a plot: Axis, Tick, Line2D, Text, etc., and sets the coordinates. It is the area in which data is plotted. Axes include the X-Axis, Y-Axis, and possibly a Z-Axis, as well.

Importing matplotlib :

The following code is used to import

from matplotlib import pyplot as plt

or

import matplotlib.pyplot as plt

How to Create Matplotlib Plots

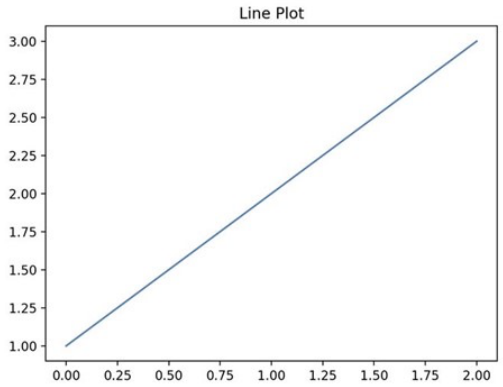
This section shows how to create examples of different kinds of plots with matplotlib.

Matplotlib Line Plot

In this example, pyplot is imported as plt, and then used to plot three numbers in a straight line:

```
import matplotlib.pyplot as plt

# Plot some numbers:

plt.plot([1, 2, 3])

plt.title(”Line Plot”)

# Display the plot:

plt.show()
```

Output

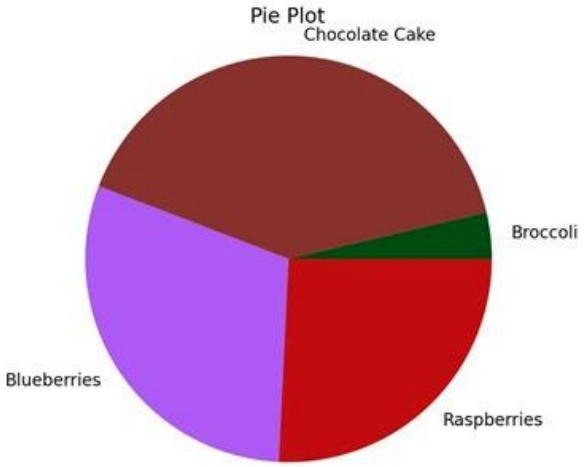


Matplotlib Pie Plot

In this example, pyplot is imported as plt, and then used to create a chart with four sections that have different labels, sizes and colors:

```
import matplotlib.pyplot as plt

# Data labels, sizes, and colors are defined:

labels = 'Broccoli', 'Chocolate Cake', 'Blueberries', 'Raspberries'

sizes = [30, 330, 245, 210]

colors = ['green', 'brown', 'blue', 'red']

# Data is plotted:

plt.pie(sizes, labels=labels, colors=colors)

plt.axis('equal')

plt.title("Pie Plot")

plt.show()
```

Output

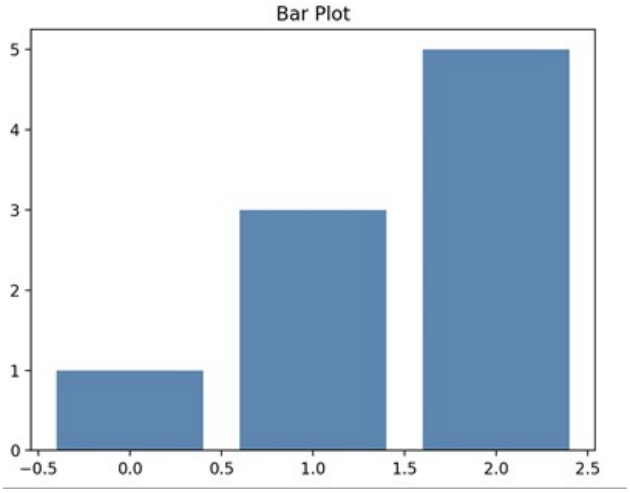


Matplotlib Bar Plot

In this example, pyplot is imported as plt, and then used to plot three vertical bar graphs:

```
import matplotlib.pyplot as plt

import numpy as np

# Create a Line2D instance with x and y data in sequences xdata, ydata:
```

```
# x data:
xdata=['A','B','C']
# y data:
ydata=[1,3,5]
plt.bar(range(len(xdata)),ydata)
plt.title(“Bar Plot”)
plt.show()
```



Plotting Graphs with Matplotlib

PPT : link uh

Pyplot

To start plotting graphs, we import the pyplot submodule from matplotlib. Pyplot is a collection of command style functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

In order to plot a 2-D graph (say), we need to have a pair of coordinates: those of the x and y axes. If we declare our coordinates as arrays, the size of the x-coordinate array and y-coordinate array must be same. Matplotlib is convenient in that in order to plot the coordinates with, say a scatter plot, we simply have to write pyplot.scatter() and we can have the points plotted on a scatter plot. Note that this will plot the graph but won't display it; we need to write pyplot.show() in order to display the graph.

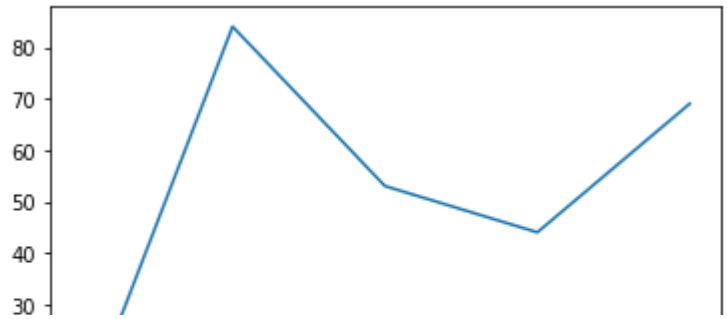
```
#Importing pyplot library
import matplotlib.pyplot as plt
#Importing Numpy library
import numpy as np
```

Matplotlib works with all sorts of data. If it were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally.

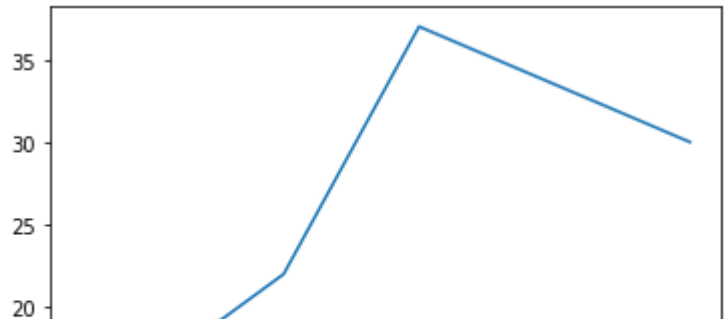
If we wish to have the points linked and not scattered, we can use pyplot.plot() for the same. The advantage of these functions is that there are several properties and parameters we can tinker with in order to enhance our graph further. We can give it colours and even the pattern of the lines in between the points, for instance dashed or bold.

You can make several plots like bar graphs, histograms etc. and also adjust scales – like setting a logarithmic scale etc. The plot() function accepts an arbitrary number of arguments.

```
#plotting values on the graph
A = [7,84,53,44,69]
plt.plot(A)
plt.show()
```




```
#plotting graph with 2 inter-related variables (through some function)
X = [2,5,7,11]
Y = [13,22,37,30]
plt.plot(X,Y)
plt.show()
```

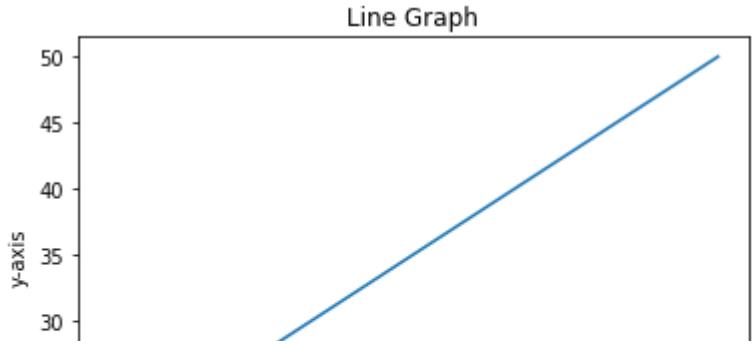


Line graph

We can plot linearly related mathematical functions (y=mx+b) using pyplot.

```
x = [10, 20, 30, 40]
y = [20, 30, 40, 50]

# plotting the data
plt.plot(x, y)

# Adding the title
plt.title("Line Graph")

# Adding the labels
plt.ylabel("y-axis")
```

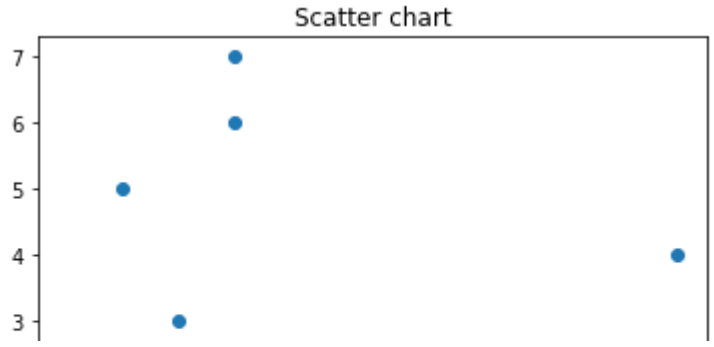
```
plt.xlabel("x-axis")
plt.show()
```



Scatter

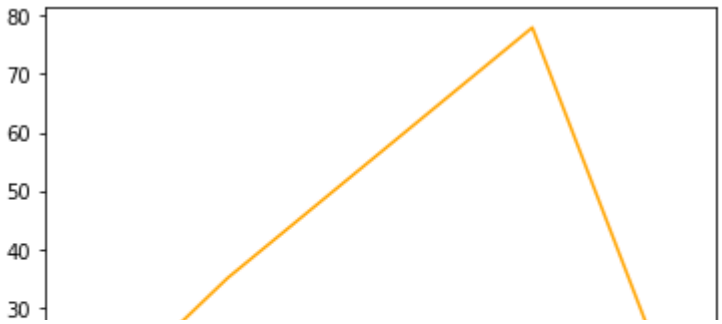
If we want to plot random data points without connecting them with lines, we can use scatter.

```
x = [3, 1, 3, 12, 2, 4, 4]
y = [3, 2, 1, 4, 5, 6, 7]

# This will plot a simple scatter chart
plt.scatter(x, y)

# Title to the plot
plt.title("Scatter chart")
plt.show()
```

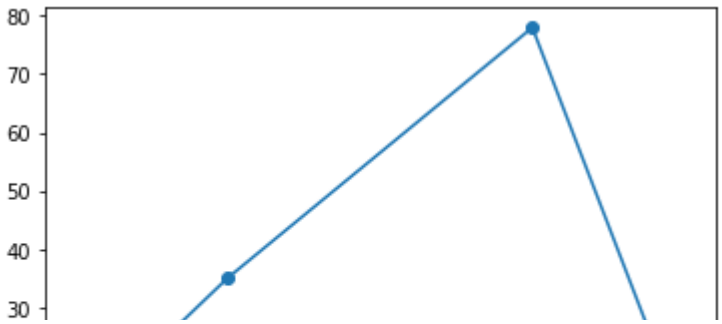


Adding Styles in graphs

```
N = 50
X = np.array([1,2,4,5])
Y = np.array([10,35,78,10])

#Color property can be used to change colours

plt.plot(X,Y,color = 'orange')
#We can use any color here in values

plt.show()
```




```
N = 50
X = np.array([1,2,4,5])
Y = np.array([10,35,78,10])

#Marker property can be used to change markers to any valid marker style like circle,diamo

plt.plot(X,Y,marker = 'o')
#We can use any color here in values

plt.show()
```

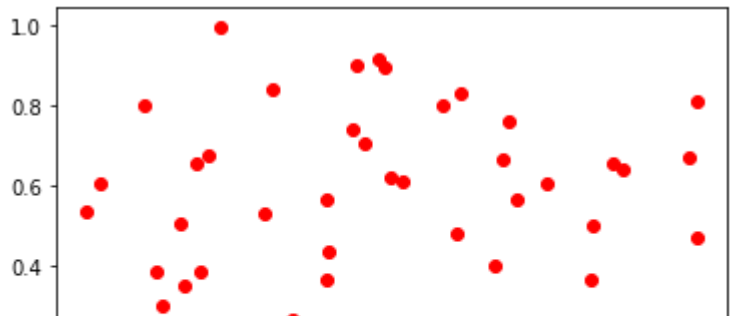


We can use format strings to change colour, markers and line styles.

fmt = '[color][marker][line]'

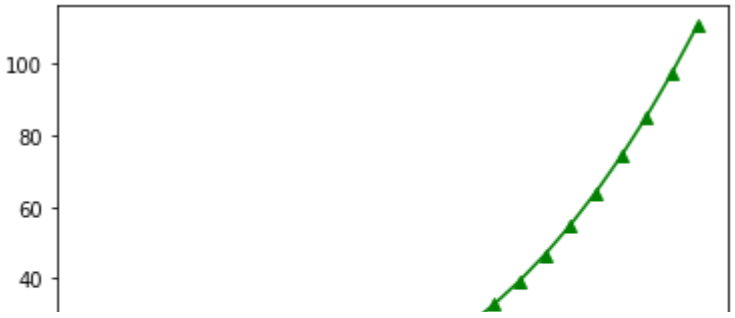
All of these are optional. By default, we have blue solid line.

```
N = 50
X = np.random.rand(N)
Y = np.random.rand(N)

plt.plot(X,Y,'ro')
# here, r is for red colour and o is for circle

# ro - is used for plotting red circles
#plt.plot(X,Y,'bo') #- for blue circle
#plt.plot(X,Y,'r+') #- red colour and + marker
#plt.plot(X,Y,'b--') #- for blue dotted lines
#plt.plot(X,Y,'o-')
#plt.plot(X,Y,'g--d') #- green dotted line with diamond shape marker
#plt.plot(X,Y,'k^:') #- black triangle_up markers connected by a dotted line

plt.show()
```




```
X = np.array([1,2,3,4])
Y = np.array([13,9,78,45])
plt.plot(X,Y,linewidth = 4.5)
#linewidth can take any float values
plt.show()
```

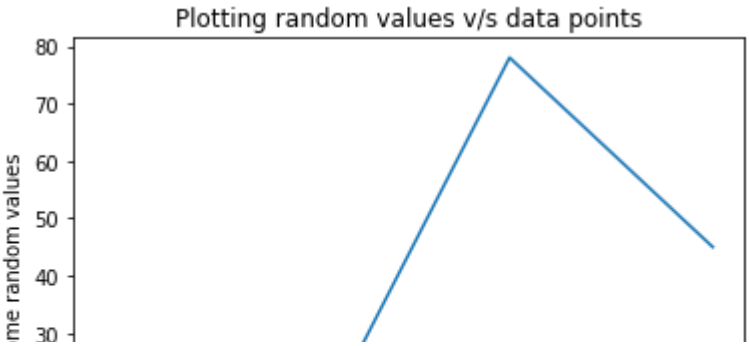

```
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
# we can plot multiple functions within one plot as well.
plt.plot(t, t**3, color = 'green', marker='^')
plt.show()
```




```
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2

# s - size of points, can be a single number or an array
# c - colour of the points
# alpha - transparency value of colours
plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```

Adding labels and title in graphs

```
X = np.array([1,2,3,4])
Y = np.array([13,9,78,45])
plt.plot(X,Y)
plt.ylabel("Some random values")
plt.xlabel("Points")
# Adding x and y axis labels
plt.title("Plotting random values v/s data points")
# Adding title on the graphs
plt.show()
```



Legends

While plotting lines on the graph, we can add legends for description about what information is shown or depicted with help of those plots.

```
plt.plot([0.1, 0.2, 0.3, 0.4], [1, 2, 3, 4], label='first plot')
plt.plot([0.1, 0.2, 0.3, 0.4], [1, 4, 9, 16], label='second plot')
plt.legend()
plt.show()
```

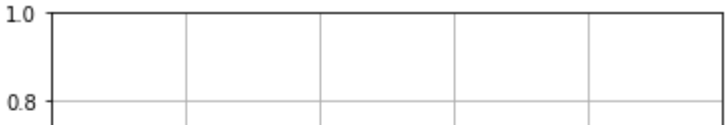
We can use axis to set axis properties easily.

```
X = np.array([2,4,6,8,10])
Y = np.array([15,29,2,44,9])
plt.plot(X,Y)
#We can set xlimits and ylimits for the plots in the following format [xmin,xmax,ymin,ymax
plt.axis([0,6,0,20])
plt.show()
```

Graph Styles

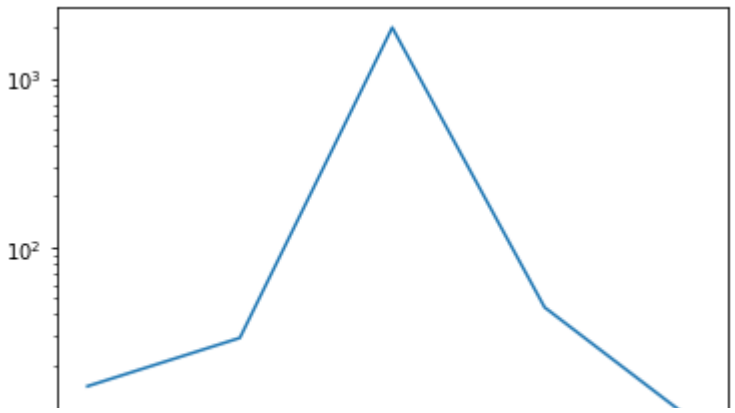
Grid style can be used to show grids on graphs.

```
plt.grid(True)
plt.show()
```



We can do axis scaling to scale the X and Y axis using the classes xscale and yscale respectively. Scaling can be linear, logarithmic, etc.

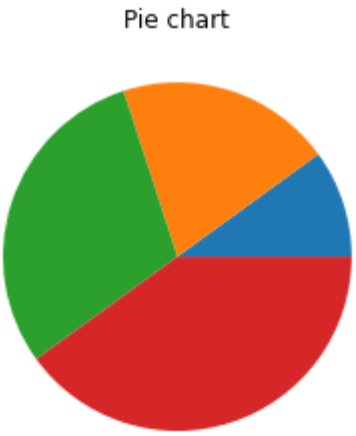
```
X = np.array([2,4,6,8,10])
Y = np.array([15,29,2000,44,9])
plt.xscale('linear')
#By default, we have linear scaling
plt.yscale('log')
plt.plot(X,Y)
plt.show()
```



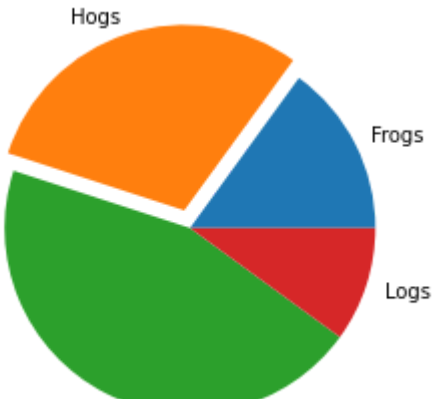
Different variations for graph plotting

Basic pie chart

```
import matplotlib.pyplot as plt
# data to display on plots
x = [1, 2, 3, 4]

# This will plot a simple pie chart
plt.pie(x)

# Title to the plot
plt.title("Pie chart")
plt.show()
```



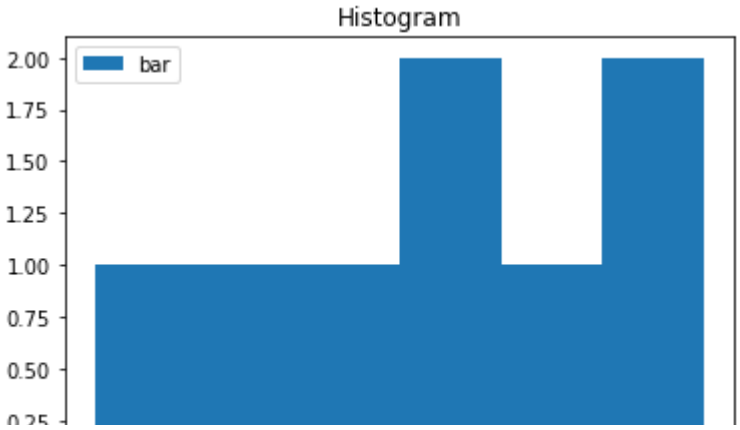

```
# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)
# only "explode" the 2nd slice (i.e. 'Hogs')

plt.pie(sizes, explode=explode, labels=labels)
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```



Histograms

```
#represent data in the form of some groups

# data to display on plots
x = [1, 2, 3, 4, 5, 6, 7, 4]
# This will plot a simple histogram
```

```
plt.hist(x, bins = [1, 2, 3, 4, 5, 6, 7])
# Title to the plot
plt.title("Histogram")
# Adding the legends
plt.legend(["bar"])
plt.show()
```



We can save the current plots or graphs using savefig.

```
X = np.array([1,2,3,4])
Y = np.array([13,9,78,45])
plt.plot(X,Y)
plt.ylabel("Some random values")
plt.xlabel("Points")
plt.title("Plotting random values v/s data points")
plt.savefig("figure.png")
#Saves the plot with filename figure.png in the same directory
```

Show hidden output

3D plot

for doing multivariate analysis and visualizing 3-D plots in 2-D space

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]
z = [1, 8, 27, 64, 125]
# Creating the figure object
fig = plt.figure()
# keeping the projection = 3d
# ctreates the 3d plot
ax = plt.axes(projection = '3d')
ax.plot3D(z, y, x)
```

```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f72d744e990>]
```

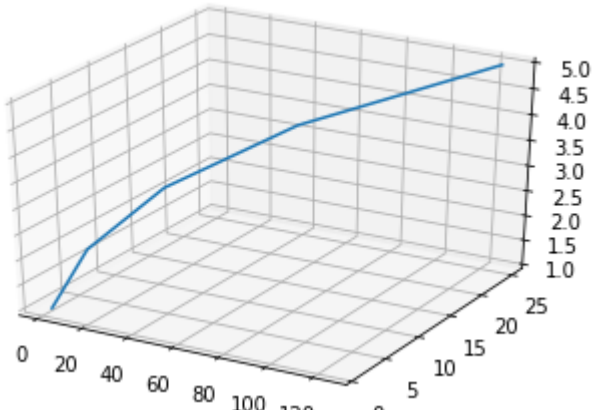
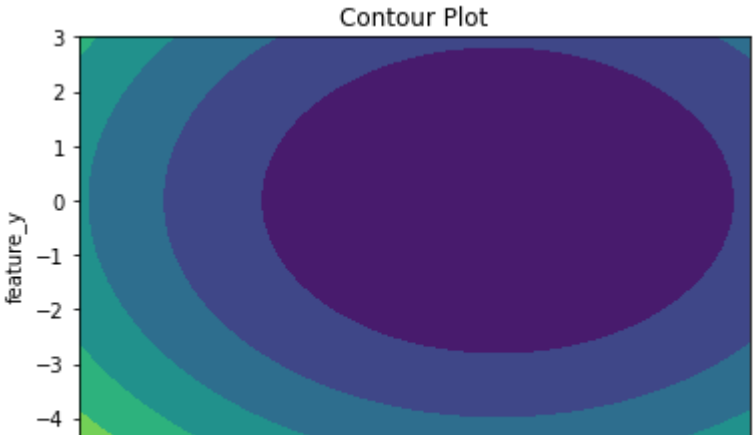


Image Plot

```
import requests
response = requests.get("https://snscourseware.org/images/SNS%20Institutionsapp.png")

file = open("sns.png", "wb")
file.write(response.content)
file.close()


# importing required libraries
import matplotlib.pyplot as plt
import matplotlib.image as img

# reading the image
testImage = img.imread('sns.png')

# displaying the image
plt.imshow(testImage)
```

Contour Plot

```
# Implementation of matplotlib function
import matplotlib.pyplot as plt
import numpy as np

feature_x = np.linspace(-5.0, 3.0, 70)
feature_y = np.linspace(-5.0, 3.0, 70)

# Creating 2-D grid of features
[X, Y] = np.meshgrid(feature_x, feature_y) #creates a rectangular grid out of given two on

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

Z = X ** 2 + Y ** 2

# plots filled contour plot
ax.contourf(X, Y, Z)

ax.set_title('Contour Plot')
ax.set_xlabel('feature_x')
ax.set_ylabel('feature_y')

plt.show()
```



Colab paid products - Cancel contracts here