UNIT-4 DATA VISUALIZATION

Section 4.2 Visualizing the Data

UNIT 4.2
VISUALIZING THE DATA

Choosing the Right Graph

- **Why to choose a Right Graph?**
- How people view the associated data, so choosing the right graph from the outset is important.
- For example, how various data elements contribute toward a whole, we need to use a pie chart.
- when you want people to form opinions on how data elements compare, we use a bar chart.
- It is used for concluding vision.

Showing Parts of a Hole with Pie Charts

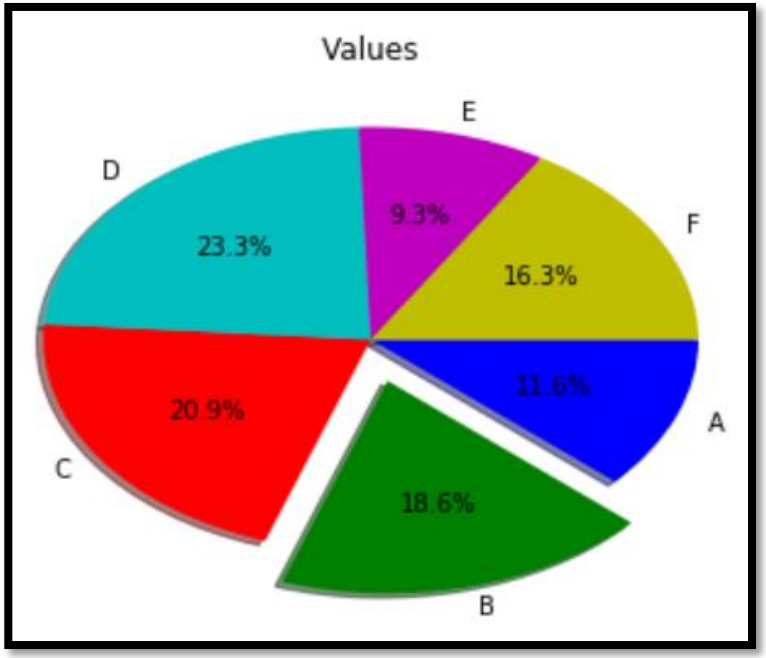
- Pie charts focus on showing parts of a whole.
- The entire pie would be 100 percent.
- The question is how much of that percentage each value occupies.

Showing Parts of a Hole with Pie Charts

- The colours parameter lets you choose custom colours for each pie wedge.
- You use the labels parameter to identify each wedge.
- In many cases you need to make one wedge stand out from the others, so you add the explode parameter with list of explode values.
- A value of 0 keeps the wedge in place — any other value moves the wedge out from the centre of the pie.
- Each pie wedge can show various kinds of information.

Showing Parts of a Hole with Pie Charts

```
import matplotlib.pyplot as plt
%matplotlib inline
values = [5, 8, 9, 10, 4, 7]
colors = ['b', 'g', 'r', 'c', 'm', 'y']
labels = ['A', 'B', 'C', 'D', 'E', 'F']
explode = (0, 0.2, 0, 0, 0, 0)
plt.pie(values, colors=colors, labels=labels,
explode=explode, autopct='%1.1f%%',
counterclock=False, shadow=True)
plt.title('Values')
plt.show()
```



Showing Parts of a Hole with Pie Charts

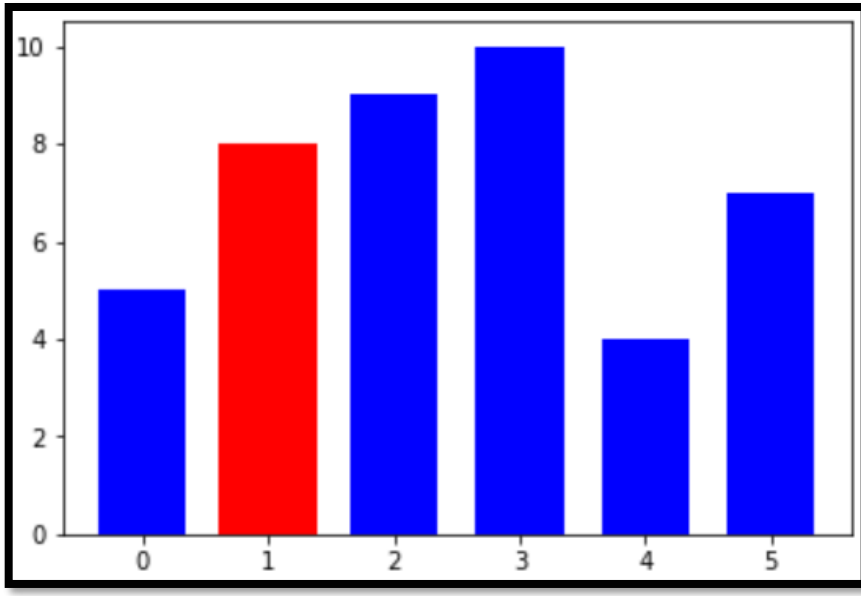
- This example shows the percentage occupied by each wedge with the autopct parameter.
- You must provide a format string to format the percentages.
- Use the counterclock parameter to determine the direction of the wedges.
- The shadow parameter determines whether the pie appears with a shadow beneath it (for a 3-D effect).
- You can find other parameters at https://matplotlib.org/api/pyplot_api.html .

Creating Comparison with Bar Charts

- Bar charts make comparing values easy.
- The wide bars and segregated measurements emphasize the differences between values.
- In the following example we can se Vertical Bar.

Creating Comparison with Bar Charts

```
import matplotlib.pyplot as plt
%matplotlib inline
values = [5, 8, 9, 10, 4, 7]
widths = [0.7, 0.8, 0.7, 0.7, 0.7, 0.7]
colors = ['b', 'r', 'b', 'b', 'b', 'b']
plt.bar(range(0, 6), values, width=widths,
color=colors, align='center')
plt.show()
```

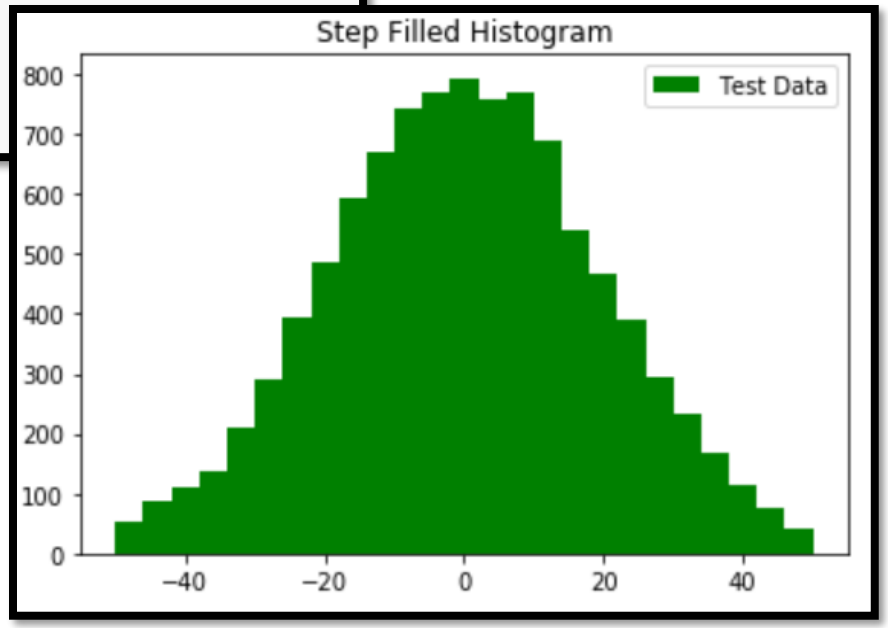


Showing distributions using histograms

- Histograms categorize data by breaking it into bins, where each bin contains a subset of the data range.
- A histogram then displays the number of items in each bin so that you can see the distribution of data and the progression of data from bin to bin.

Showing distributions using histograms

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
x = 20 * np.random.randn(10000)
plt.hist(x, 25, range=(-50, 50), histtype='stepfilled',
align='mid', color='g', label='Test Data')
plt.legend()
plt.title('Step Filled Histogram')
plt.show()
```



Depicting groups using boxplots

There are four main measures of variability: **Range, Quartiles, Variance, Standard deviation and Inter quartile range.**

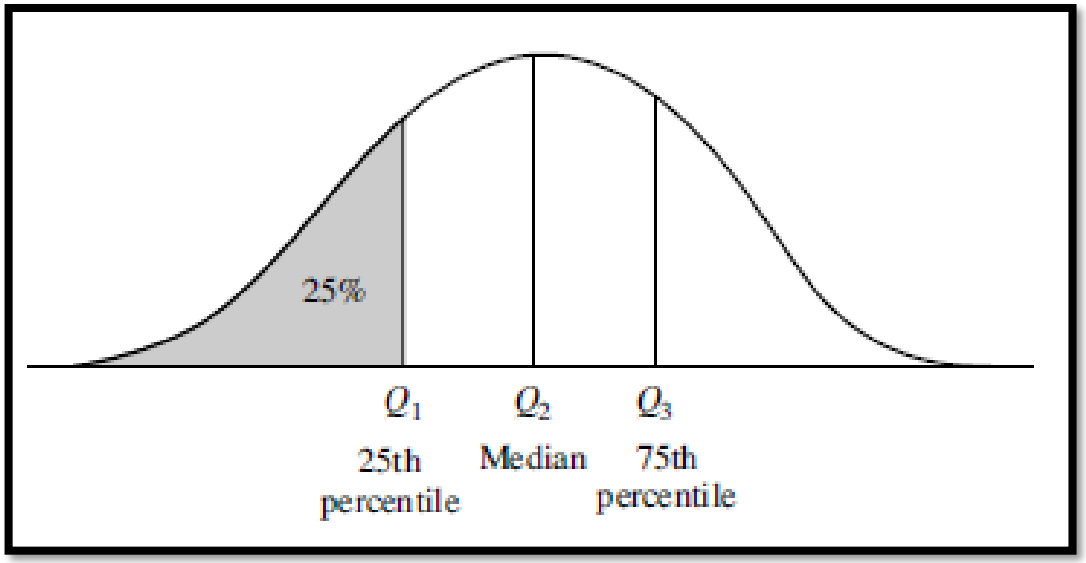
Definition: Quantiles

- Quantiles are points taken at regular intervals of a data distribution, dividing it into essentially equal size consecutive sets.

Depicting groups using boxplots

Description of Quantiles

- Let x1,x2,….,xn be a set of observations for some numeric attribute, X. The range of the set is the difference between the largest (max()) and smallest (min()) values. Suppose that the data for attribute X are sorted in increasing numeric order. Imagine that we can pick certain data points so as to split the data distribution into equal-size consecutive sets, as in Figure. These data points are called **quantiles.**



Depicting groups using boxplots

Definition: Quartiles

- As per above figure, each part represents one-fourth of the data distribution. They are more commonly referred to as quartiles.

Definition: Percentiles

- The 100-quantiles are more commonly referred to as percentiles, they divide the data distribution into 100 equal-sized consecutive sets.

Definition: Interquartile Range

- The distance between the first and third quartiles is a simple measure of spread that gives the range covered by the middle half of the data. This distance is called the interquartile range (IQR) and is defined as **IQR = Q3 - Q1**.

Depicting groups using boxplots

Definition: Five Number Summary

- The five-number summary of a distribution consists of the median (Q2), the quartiles Q1 and Q3, and the smallest and largest individual observations, written in the order of **Minimum, Q1, Median, Q3, Maximum.**

Definition: Boxplots

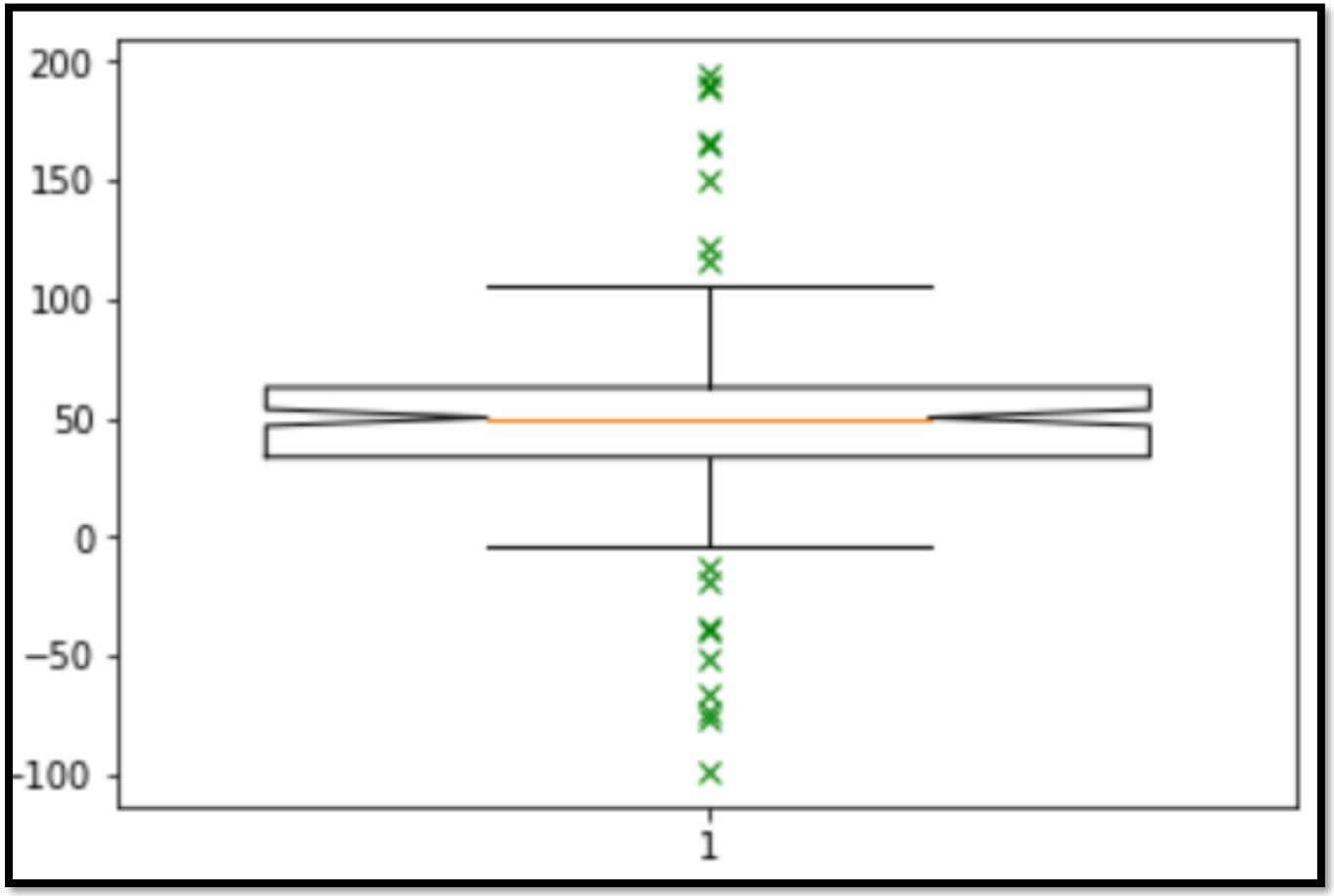
Boxplots are a popular way of visualizing a distribution. A boxplot incorporates the five-number summary as follows:

- Typically, the ends of the box are at the quartiles so that the box length is the interquartile range.
- The median is marked by a line within the box.
- Two lines (called whiskers)

Depicting groups using boxplots

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
spread = 100 * np.random.rand(100)
center = np.ones(50) * 50
flier_high = 100 * np.random.rand(10) + 100
flier_low = -100 * np.random.rand(10)
data = np.concatenate((spread, center,
flier_high, flier_low))
plt.boxplot(data, sym='gx', widths=.75, notch=True)
plt.show()
```

Depicting groups using boxplots



Depicting groups using boxplots

- spread: Contains a set of random numbers between 0 and 100
- center: Provides 50 values directly in the center of the range of 50
- flier_high: Simulates outliers between 100 and 200
- flier_low: Simulates outliers between 0 and –100
- The code combines all these values into a single dataset using concatenate().
- In this case, the code sets the presentation of outliers to green Xs by setting the sym parameter.
- You use widths to modify the size of the box (made extra large in this case to make the box easier to see).
- Finally, you can create a square box or a box with a notch using the notch parameter (which normally defaults to False).

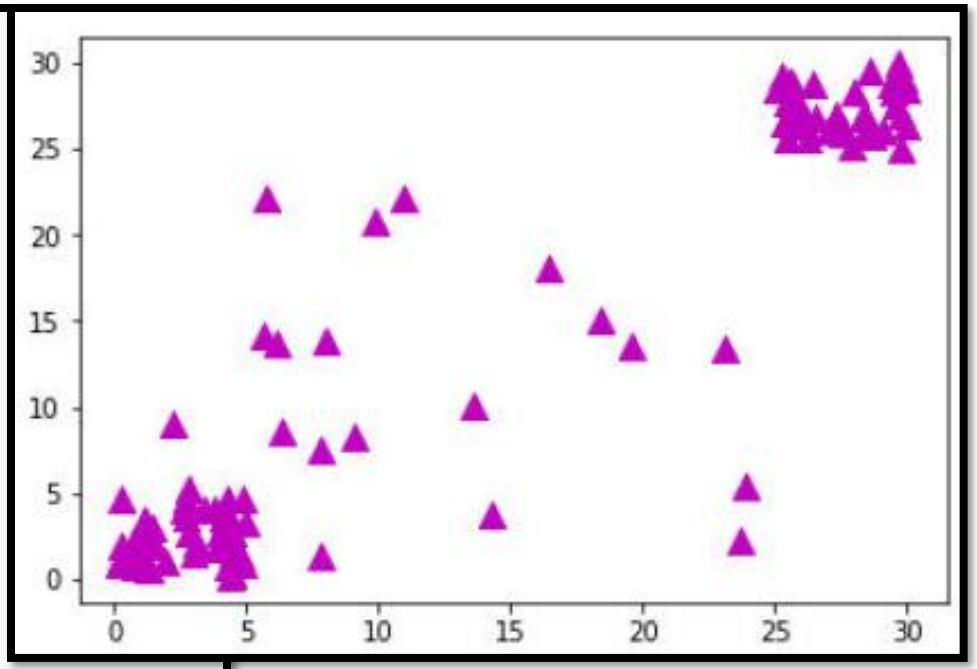
Depicting groups using boxplots

- The box shows the three data points as the box, with the red line in the middle being the median.
- The two black horizontal lines connected to the box by whiskers show the upper and lower limits (for four quartiles).
- The outliers appear above and below the upper and lower limit lines as green Xs.

Seeing Data Patterns using scatterplots

- Scatterplots show clusters of data rather than trends (as with line graphs) or discrete values (as with bar charts).
- The purpose of a scatterplot is to help you see data patterns.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

x1 = 5 * np.random.rand(40)
x2 = 5 * np.random.rand(40) + 25
x3 = 25 * np.random.rand(20)
x = np.concatenate((x1, x2, x3))

y1 = 5 * np.random.rand(40)
y2 = 5 * np.random.rand(40) + 25
y3 = 25 * np.random.rand(20)
y = np.concatenate((y1, y2, y3))

plt.scatter(x, y, s=[100], marker='^', c='m')
plt.show()
```

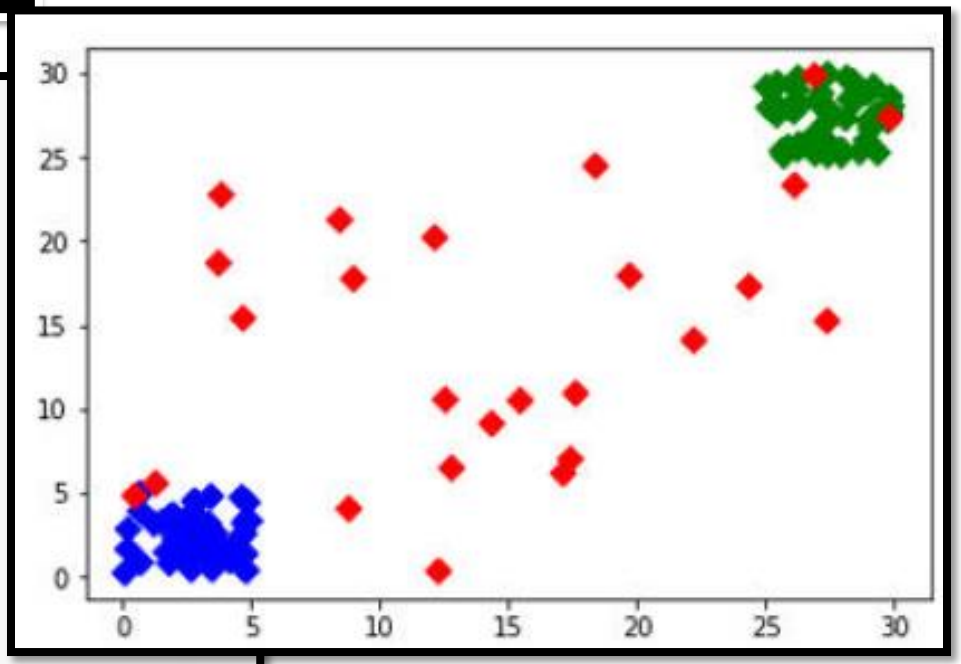


Seeing Data Patterns using scatterplots

- The example begins by generating random x and y coordinates.
- For each x coordinate, you must have a corresponding y coordinate.
- It's possible to create a scatterplot using just the x and y coordinates.
- In this case, the s parameter determines the size of each data point. The marker parameter determines the data point shape. You use the c parameter to define the colours for all the data points, or you can define a separate colour for individual data points.

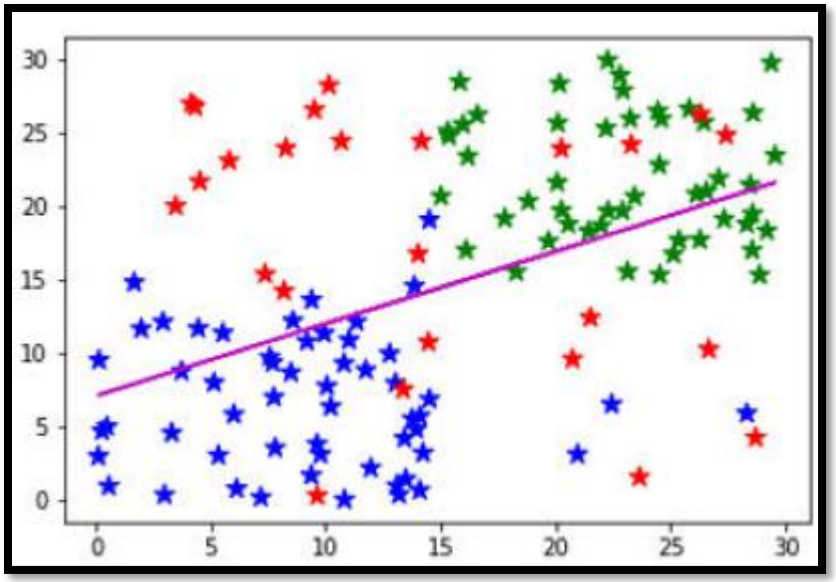
Creating Advanced Scatterplots Depicting groups

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

x1 = 5 * np.random.rand(50)
x2 = 5 * np.random.rand(50) + 25
x3 = 30 * np.random.rand(25)
x = np.concatenate((x1, x2, x3))

y1 = 5 * np.random.rand(50)
y2 = 5 * np.random.rand(50) + 25
y3 = 30 * np.random.rand(25)
y = np.concatenate((y1, y2, y3))

color_array = ['b'] * 50 + ['g'] * 50 + ['r'] * 25

plt.scatter(x, y, s=[50], marker='D', c=color_array)
plt.show()
```



Creating Advanced Scatterplots Showing Correlations

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.pylab as plb
%matplotlib inline

x1 = 15 * np.random.rand(50)
x2 = 15 * np.random.rand(50) + 15
x3 = 30 * np.random.rand(30)
x = np.concatenate((x1, x2, x3))

y1 = 15 * np.random.rand(50)
y2 = 15 * np.random.rand(50) + 15
y3 = 30 * np.random.rand(30)
y = np.concatenate((y1, y2, y3))

color_array = ['b'] * 50 + ['g'] * 50 + ['r'] * 25
plt.scatter(x, y, s=[90], marker='*', c=color_array)
z = np.polyfit(x, y, 1)
p = np.poly1d(z)
plb.plot(x, p(x), 'm-')

plt.show()
```

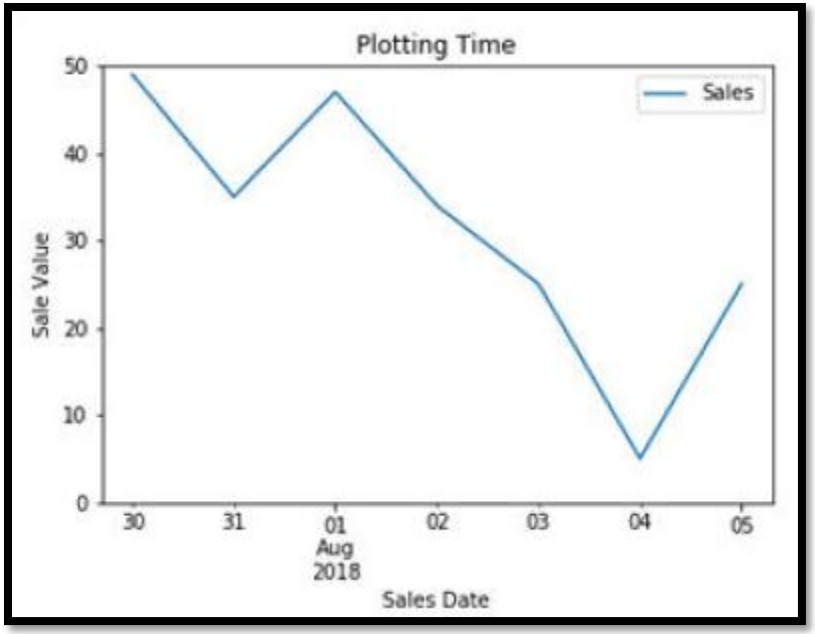


Creating Advanced Scatterplots Showing Correlations

- Adding a trendline means calling the NumPy polyfit() function with the data, which returns a vector of coefficients, p, that minimizes the least-squares error.
- Leastsquare regression is a method for finding a line that summarizes the relationship between two variables, x and y in this case, at least within the domain of the explanatory variable x. The third polyfit() parameter expresses the degree of the polynomial fit.
- The vector output of polyfit() is used as input to poly1d(), which calculates the actual y axis data points. The call to plot() creates the trendline on the scatterplot.

Plotting Time Series
Representing time on axes

```
import pandas as pd
import matplotlib.pyplot as plt
import datetime as dt
%matplotlib inline

start_date = dt.datetime(2018, 7, 30)
end_date = dt.datetime(2018, 8, 5)
daterange = pd.date_range(start_date, end_date)
sales = (np.random.rand(len(daterange)) * 50).astype(int)
df = pd.DataFrame(sales, index=daterange,
                  columns=['Sales'])

df.loc['Jul 30 2018':'Aug 05 2018'].plot()
plt.ylim(0, 50)
plt.xlabel('Sales Date')
plt.ylabel('Sale Value')
plt.title('Plotting Time')
plt.show()
```



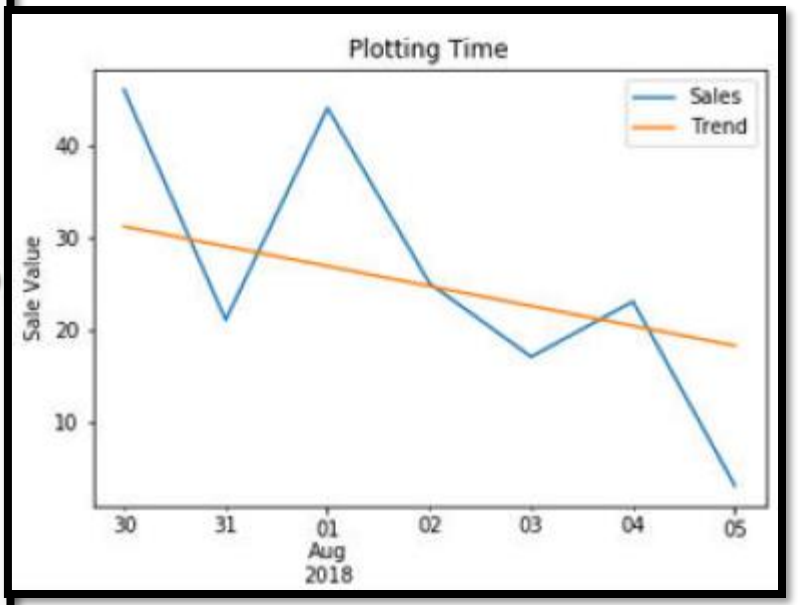
Plotting Time Series
Representing time on axes

- Using loc[] lets you select a range of dates from the total number of entries available.

Notice that this example uses only some of the generated data for output. It then adds some amplifying information about the plot and displays it onscreen. The call to plot() must specify the x and y values

Plotting Time Series
Plotting trends over time

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime as dt

start_date = dt.datetime(2018, 7, 29)
end_date = dt.datetime(2018, 8, 7)
daterange = pd.date_range(start_date, end_date)
sales = (np.random.rand(len(daterange)) * 50).astype(int)
df = pd.DataFrame(sales, index=daterange,
                  columns=['Sales'])

lr_coef = np.polyfit(range(0, len(df)), df['Sales'], 1)
lr_func = np.poly1d(lr_coef)
trend = lr_func(range(0, len(df)))
df['trend'] = trend
df.loc['Jul 30 2018':'Aug 05 2018'].plot()

plt.xlabel('Sales Date')
plt.ylabel('Sale Value')
plt.title('Plotting Time')
plt.legend(['Sales', 'Trend'])
plt.show()
```



if you print df after the call to df['trend'] = trend, you see trendline data similar to the values.

shown here:

Plotting Geographical Data

- Knowing where data comes from or how it applies to a specific place can be important. For example, if you want to know where food shortages have occurred and plan how to deal with them, you need to match the data you have to geographical locations.

- **Getting the Basemap Toolkit**

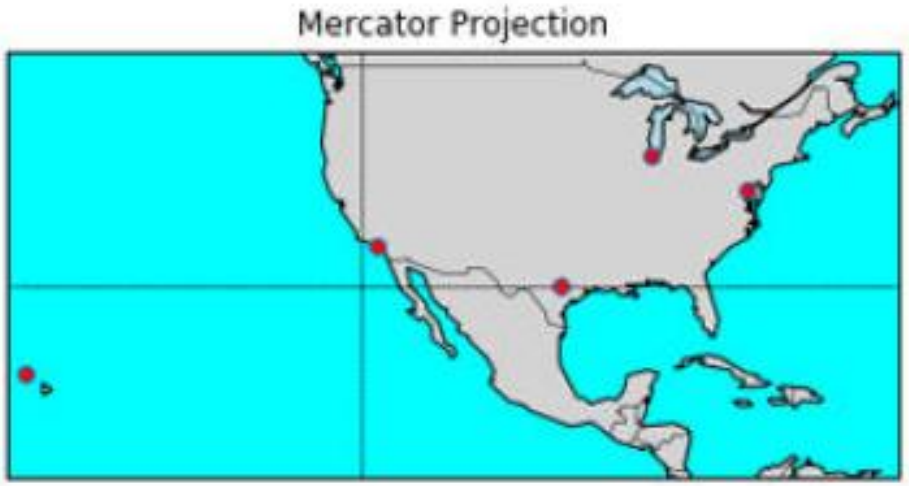
  https://matplotlib.org/basemap/users/intro.html

- **Dealing with deprecated library issues**

  https://github.com/matplotlib/basemap/issues/382

Using Basemap to plot Gepgraphic Data

```
austin = (-97.75, 30.25)
hawaii = (-157.8, 21.3)
washington = (-77.01, 38.90)
chicago = (-87.68, 41.83)
losangeles = (-118.25, 34.05)

m = Basemap(projection='merc',llcrnrlat=10,urcrnrlat=50,
            llcrnrlon=-160,urcrnrlon=-60)

m.drawcoastlines()
m.fillcontinents(color='lightgray',lake_color='lightblue')
m.drawparallels(np.arange(-90.,91.,30.))
m.drawmeridians(np.arange(-180.,181.,60.))
m.drawmapboundary(fill_color='aqua')

m.drawcountries()

x, y = m(*zip(*[hawaii, austin, washington,
                chicago, losangeles]))
m.plot(x, y, marker='o', markersize=6,
       markerfacecolor='red', linewidth=0)

plt.title("Mercator Projection")
plt.show()
```

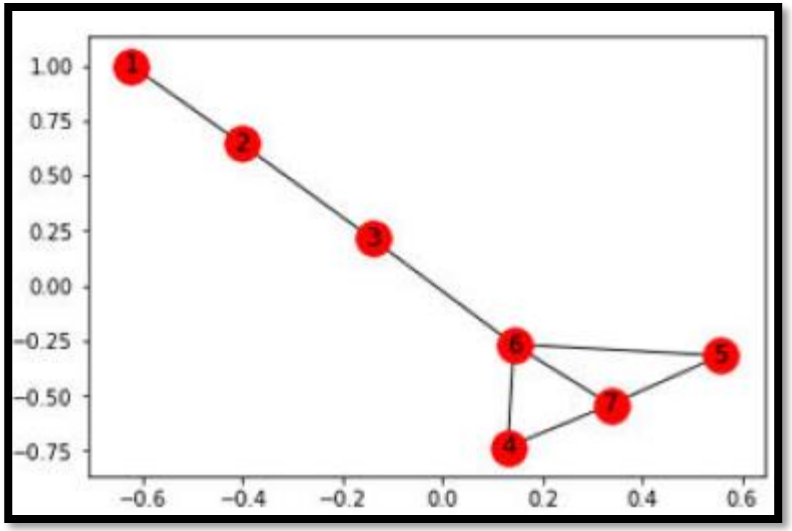


Visualizing Graph Developing Undirected Graphs

- An undirected graph simply shows connections between nodes. The output doesn't provide a direction from one node to the next. For example, when establishing connectivity between web pages, no direction is implied.

Visualizing Graphs
Developing Undirected Graph

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G = nx.Graph()
H = nx.Graph()
G.add_node(1)
G.add_nodes_from([2, 3])
G.add_nodes_from(range(4, 7))
H.add_node(7)
G.add_nodes_from(H)

G.add_edge(1, 2)
G.add_edge(1, 1)
G.add_edges_from([(2,3), (3,6), (4,6), (5,6)])
H.add_edges_from([(4,7), (5,7), (6,7)])
G.add_edges_from(H.edges())

nx.draw_networkx(G)
plt.show()
```



Visualizing Graph
Developing Directed Graphs

- You use directed graphs when you need to show a direction, say from a start point to an end point.

- When you get a map that shows you how to get from one specific point to another, the starting node and ending node are marked as such and the lines between these nodes (and all the intermediate nodes), show direction.

- The example begins by creating a directional graph using the **DiGraph()** constructor. You should note that the **NetworkX** package also supports **MultiGraph()** and **MultiDiGraph()** graph types. You can see a listing of all the graph types at https://networkx.lanl.gov/reference/classes.html.

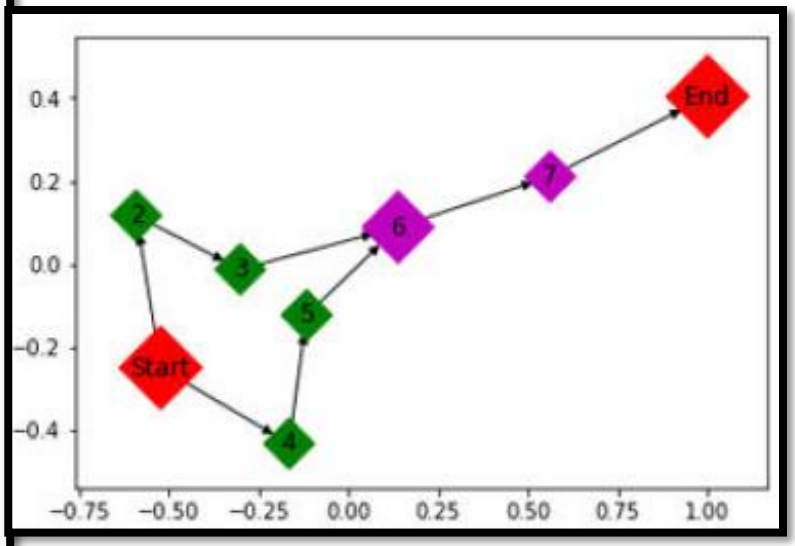
Developing Directed Graphs

- Adding nodes is much like working with an undirected graph. You can add single nodes using **add_node()** and multiple nodes using **add_nodes_from().** The **add_path()** call lets you create nodes and edges at the same time. The order of nodes in the call is important. The flow from one node to another is from left to right in the list supplied to the call.

- This example adds special node colors, labels, shape (only one shape is used), and sizes to the output. You still call on draw_networkx() to perform the task. However, adding the parameters shown changes the appearance of the graph. Note that you must set with_labels to True in order to see the labels provided by the labels parameter.

Developing Directed Graphs

```
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline

G = nx.DiGraph()

G.add_node(1)
G.add_nodes_from([2, 3])
G.add_nodes_from(range(4, 6))
G.add_path([6, 7, 8])

G.add_edge(1, 2)
G.add_edges_from([(1,4), (4,5), (2,3), (3,6), (5,6)])

colors = ['r', 'g', 'g', 'g', 'g', 'm', 'm', 'r']
labels = {1:'Start', 2:'2', 3:'3', 4:'4',
          5:'5', 6:'6', 7:'7', 8:'End'}
sizes = [800, 300, 300, 300, 300, 600, 300, 800]

nx.draw_networkx(G, node_color=colors, node_shape='D',
                 with_labels=True, labels=labels,
                 node_size=sizes)
plt.show()
```



THANK YOU …… !