CHENNAI INSTITUTE OF TECHNOLOGY

(Affiliated to Anna University, Approved by AICTE, Accredited by NAAC & NBA)
Sarathy Nagar, Kundrathur, Chennai – 600069, India.

Lecture Notes Unit V

DEPARTMENT OF INFORMATION TECHNOLOGY

Subject: CS 3353-Foundations of Data Science

Dr.A.R.Kavitha

II Year IT/III SEMESTER

UNIT V DATA VISUALIZATION

Importing Matplotlib – Line plots – Scatter plots – visualizing errors – density and contour plots – Histograms – legends – colors – subplots – text and annotation – customization – three dimensional plotting - Geographic Data with Basemap - Visualization with Seaborn.

Python Matplotlib. matplotlib.pyplot is a python package used for 2D graphics.

- What Is Python Matplotlib?
- What is Matplotlib used for?
- Types Of Plots
  - Bar Graph
  - Histogram
  - Scatter Plot
  - Area Plot
  - Pie Chart

What Is Python Matplotlib?

matplotlib.pyplot is a plotting library used for 2D graphics in python programming language. It can be used in python scripts, shell, web application servers and other graphical user interface toolkits.

What is Matplotlib used for?

Matploitlib is a Python Library used for plotting, this python library provides and objected-oriented APIs for integrating plots into applications.

Is Matplotlib Included in Python?

Matplotlib is not a part of the Standard Libraries which is installed by default when Python, there are several toolkits which are available that extend python matplotlib functionality. Some of them are separate downloads, others can be shipped with the matplotlib source code but have external dependencies.

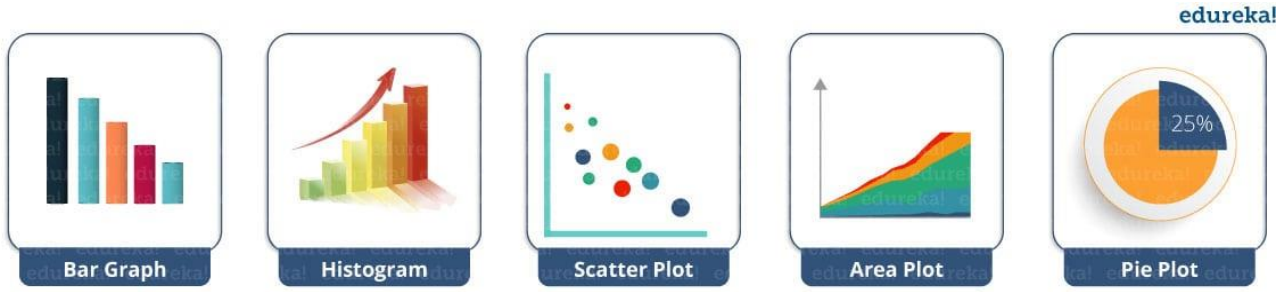
- **Basemap**: It is a map plotting toolkit with various map projections, coastlines and political boundaries.
- **Cartopy**: It is a mapping library featuring object-oriented map projection definitions, and arbitrary point, line, polygon and image transformation capabilities.
- **Excel tools**: Matplotlib provides utilities for exchanging data with Microsoft Excel.
- **Mplot3d**: It is used for 3-D plots.
- **Natgrid**: It is an interface to the natgrid library for irregular gridding of the spaced data.

You may go through this recording of Python Matplotlib where our instructor has explained how to download Matplotlib in Python and the topics in a detailed manner with examples that will help you to understand this concept better.

Next, let us move forward in this blog and explore different types of plots available in python matplotlib.

Python Matplotlib : Types of Plots

There are various plots which can be created using python matplotlib. Some of them are listed below:

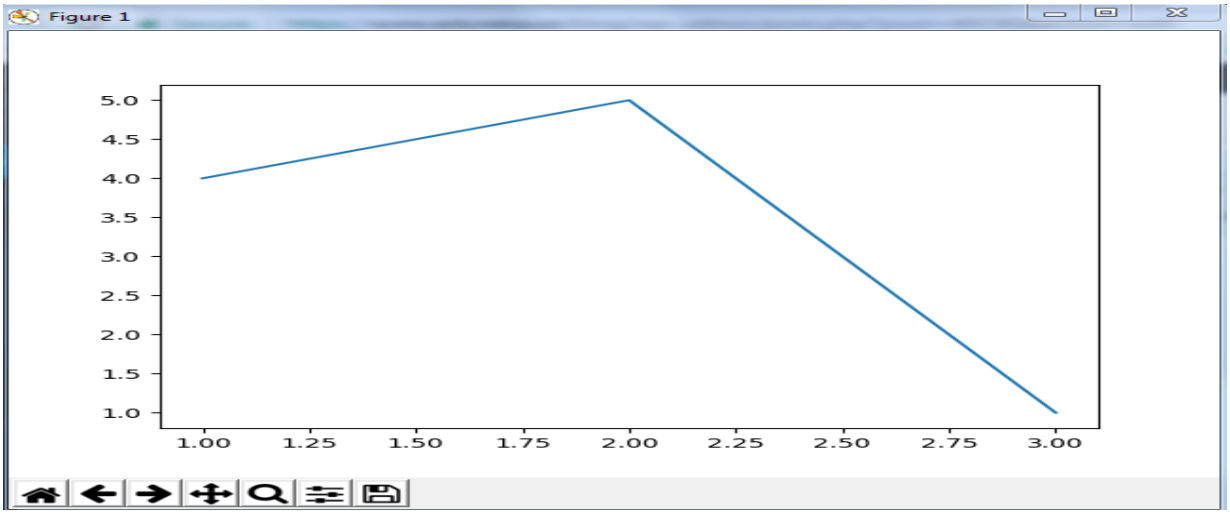


I will demonstrate each one of them in detail.

But before that, let me show you very basic codes in python matplotlib in order to generate a simple graph.

```
from matplotlib import pyplot as plt
 #Plotting to our canvas
 plt.plot([1,2,3],[4,5,1])
 #Showing what we plotted
 plt.show()
```

Output –

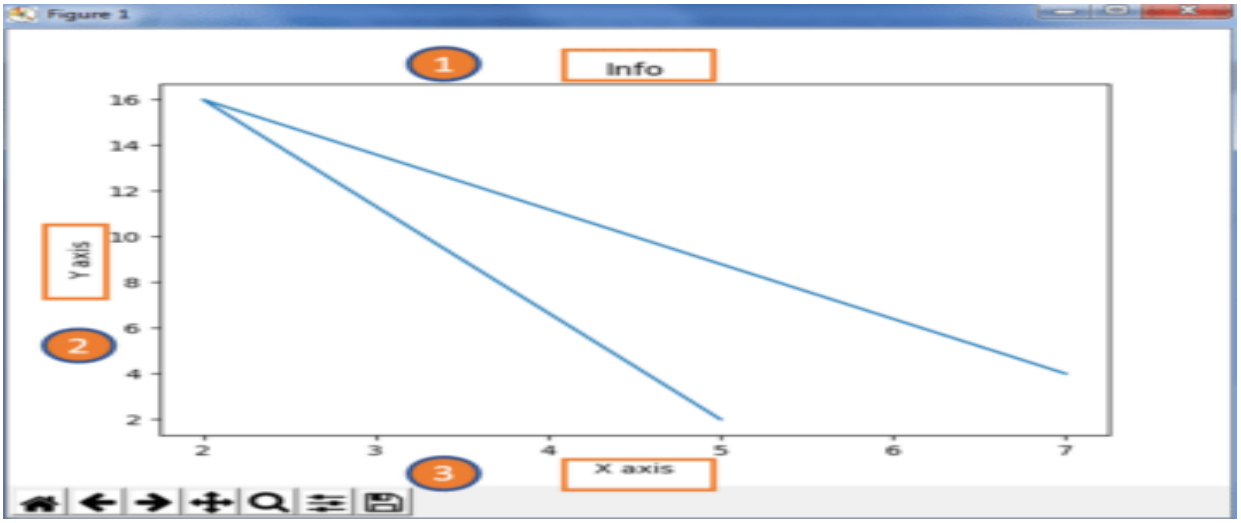


So, with three lines of code, you can generate a basic graph using python matplotlib. Simple, isn't it?

Let us see how can we add title, labels to our graph created by python matplotlib library to bring in more meaning to it. Consider the below example:

```
1   from matplotlib import pyplot as plt
2
3   x = [5,2,7]
4   y = [2,16,4]
5   plt.plot(x,y)
6   plt.title('Info')
7   plt.ylabel('Y axis')
8   plt.xlabel('X axis')
9   plt.show()
```

Output –



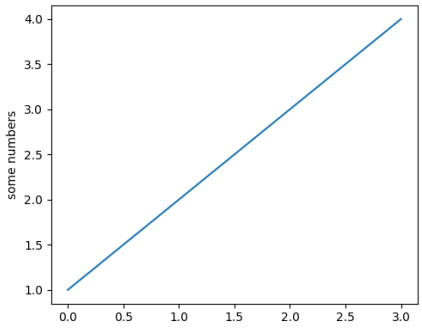
Pyplot

Pyplot

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 matplotlib.pyplot various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that “axes” here and in most places in the documentation refers to the *axes* part of a figure and not the strict mathematical term for more than one axis).

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
```

```
plt.show()
```

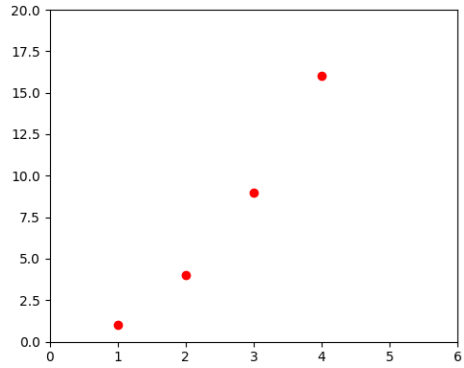


You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3].
plot() is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue

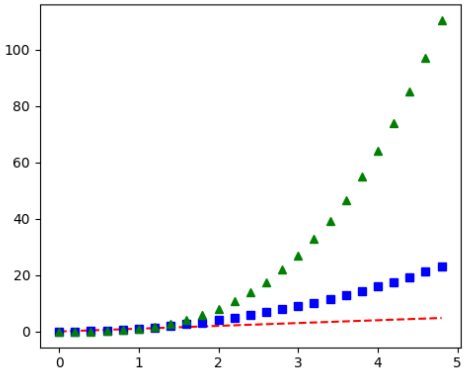
```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```



See the plot() documentation for a complete list of line styles and format strings. The axis() command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

If matplotlib 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. The example below illustrates a plotting several lines with different format styles in one command using arrays.

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

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

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```



Controlling line properties

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see matplotlib.lines.Line2D. There are several ways to set line properties

- Use keyword args:

```
plt.plot(x, y, linewidth=2.0)
```

- Use the setter methods of a Line2D instance. plot returns a list of Line2D objects; e.g., line1, line2 = plot(x1, y1, x2, y2). In the code below we will suppose that we have only one line so that the list returned is of length 1. We use tuple unpacking with line, to get the first element of that list:

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False) # turn off antialising
```

- Use the setp() command. The example below uses a MATLAB-style command to set multiple properties on a list of lines. setp works transparently with a list of objects or a single object. You can either use python keyword arguments or MATLAB-style string/value pairs:

```
lines = plt.plot(x1, y1, x2, y2)
# use keyword args
plt.setp(lines, color='r', linewidth=2.0)
# or MATLAB style string value pairs
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
```

Here are the available Line2D properties.

Property	Value Type
alpha	float
animated	[True | False]
antialiased or aa	[True | False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True | False]
clip_path	a Path instance and a Transform instance, a Patch
color or c	any matplotlib color
contains	the hit testing function
dash_capstyle	['butt' | 'round' | 'projecting']

Property	Value Type
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
data	(np.array xdata, np.array ydata)
figure	a matplotlib.figure.Figure instance
label	any string
linestyle or ls	['-' | '--' | '-.' | ':' | 'steps' | ...]
linewidth or lw	float value in points
lod	[True | False]
marker	['+' | ',' | '.' | '1' | '2' | '3' | '4']
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markersize or ms	float
markevery	[None | integer | (startind, stride)]
picker	used in interactive line selection
pickradius	the line pick selection radius
solid_capstyle	['butt' | 'round' | 'projecting']
solid_joinstyle	['miter' | 'round' | 'bevel']
transform	a matplotlib.transforms.Transform instance
visible	[True | False]
xdata	np.array
ydata	np.array
zorder	any number

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:

```
import matplotlib.pyplot as plt
```

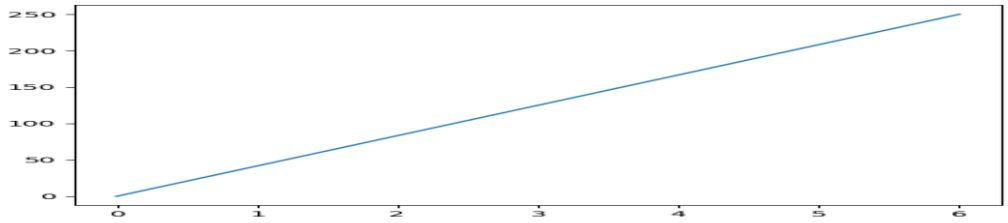
Now the Pyplot package can be referred to as plt.

Example

Draw a line in a diagram from position (0,0) to position (6,250):

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

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.show()
```

Result:



Creating Scatter Plots

With Pyplot, you can use the scatter() function to draw a scatter plot.

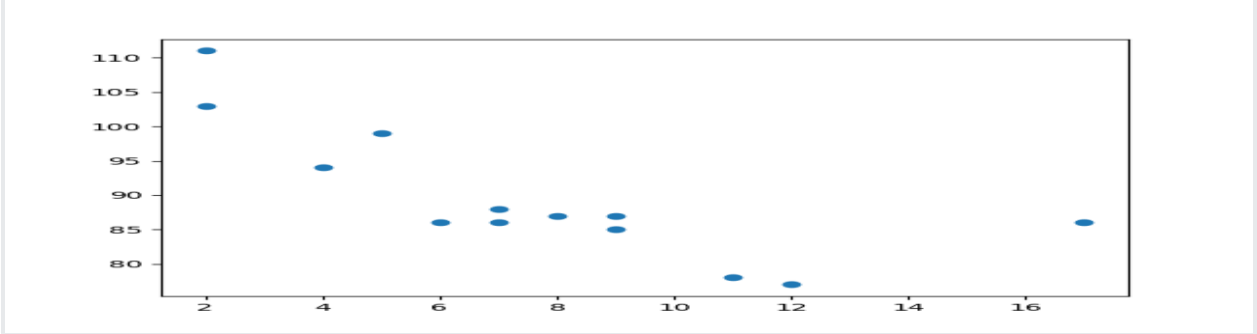
The scatter() function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

Example

A simple scatter plot:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])

plt.scatter(x, y)
plt.show()
```

Result:

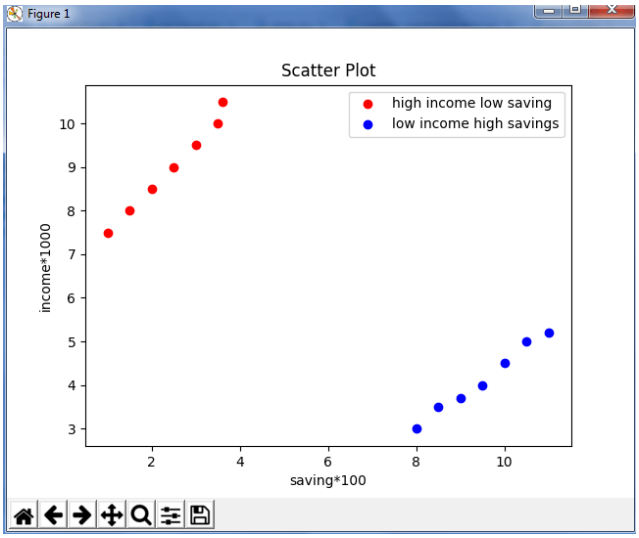


Usually we need scatter plots in order to compare variables, for example, how much one variable is affected by another variable to build a relation out of it. The data is displayed as a collection of points, each having the value of one variable which determines the position on the horizontal axis and the value of other variable determines the position on the vertical axis.

Consider the below example:

```
import matplotlib.pyplot as plt
x = [1,1.5,2,2.5,3,3.5,3.6]
y = [7.5,8,8.5,9,9.5,10,10.5]

x1=[8,8.5,9,9.5,10,10.5,11]
y1=[3,3.5,3.7,4,4.5,5,5.2]

plt.scatter(x,y, label='high income low saving',color='r')
plt.scatter(x1,y1,label='low income high savings',color='b')
plt.xlabel('saving*100')
plt.ylabel('income*1000')
plt.title('Scatter Plot')
plt.legend()
plt.show()
```

Output –



As you can see in the above graph, I have plotted two scatter plots based on the inputs specified in the above code. The data is displayed as a collection of points having 'high income low salary' and 'low income high salary'.

Creating Bars

First, let us understand why do we need a bar graph. A bar graph uses bars to compare data among different categories. It is well suited when you want to measure the changes over a period of time. It can be represented horizontally or vertically. Also, the important thing to keep in mind is that longer the bar, greater is the value. Now, let us practically implement it using python matplotlib.

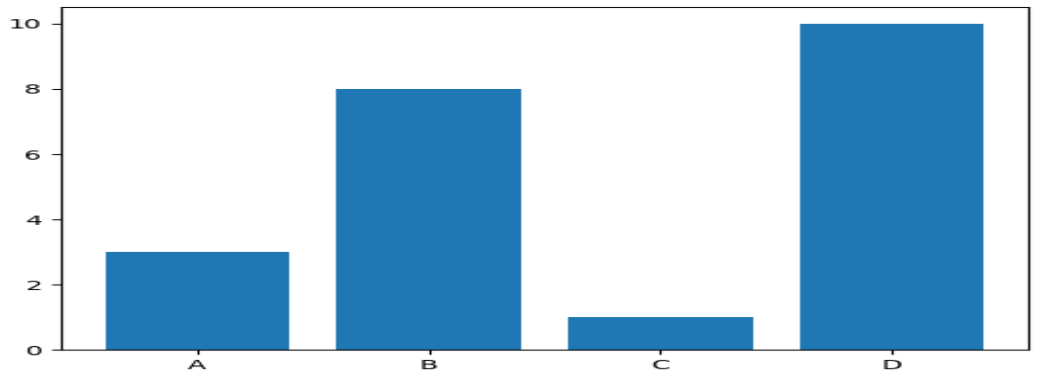
With Pyplot, you can use the bar() function to draw bar graphs:

Example

Draw 4 bars:

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```

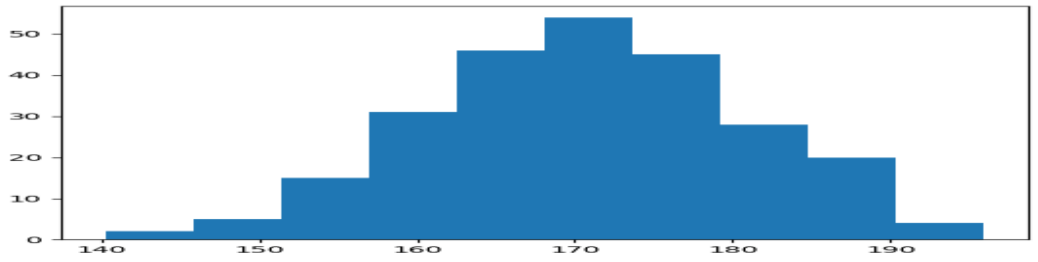
Result:



A simple histogram:

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

x = np.random.normal(170, 10, 250)

plt.hist(x)
plt.show()
```

Result:



Creating Pie Charts

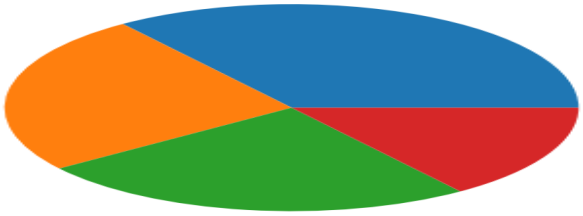
With Pyplot, you can use the pie() function to draw pie charts:

Example

A simple pie chart:

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

y = np.array([35, 25, 25, 15])

plt.pie(y)
plt.show()
```

Result:



Visualizing Errors

For any scientific measurement, accurate accounting for errors is nearly as important, if not more important, than accurate reporting of the number itself. For example, imagine that I am using some astrophysical observations to estimate the Hubble Constant, the local measurement of the expansion rate of the Universe. I know that the current literature suggests a value of around 71 (km/s)/Mpc, and I measure a value of 74 (km/s)/Mpc with my method. Are the values consistent? The only correct answer, given this information, is this: there is no way to know.

Suppose I augment this information with reported uncertainties: the current literature suggests a value of around 71 ±± 2.5 (km/s)/Mpc, and my method has measured a value of 74 ±± 5 (km/s)/Mpc. Now are the values consistent? That is a question that can be quantitatively answered.

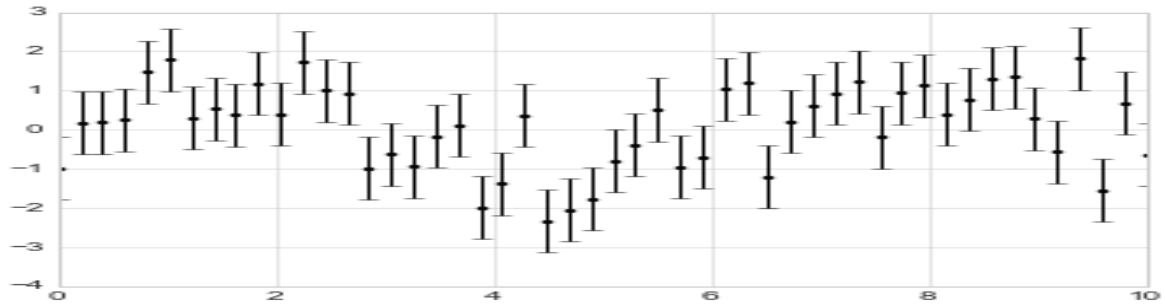
In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.

Basic Errorbars

A basic errorbar can be created with a single Matplotlib function call:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```
In [2]:

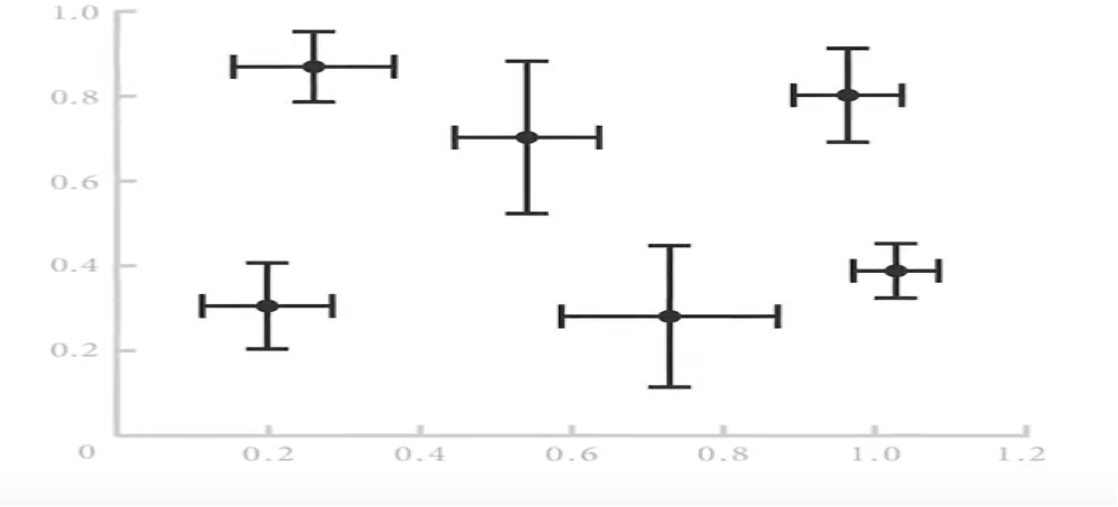
```
x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)

plt.errorbar(x, y, yerr=dy, fmt='.k');
```



Here the `fmt` is a format code controlling the appearance of lines and points, and has the same syntax as the shorthand used in `plt.plot`, outlined in Simple Line Plots and Simple Scatter Plots.

In addition to these basic options, the `errorbar` function has many options to fine-tune the outputs. Using these additional options you can easily customize the aesthetics of your errorbar plot. I often find it helpful, especially in crowded plots, to make the errorbars lighter than the points themselves:

Error bars always run parallel to a quantity of scale axis so they can be displayed either vertically or horizontally depending on whether the quantitative scale is on the y-axis or x-axis if there are two quantity of scales and two pairs of arrow bars can be used for both axes.



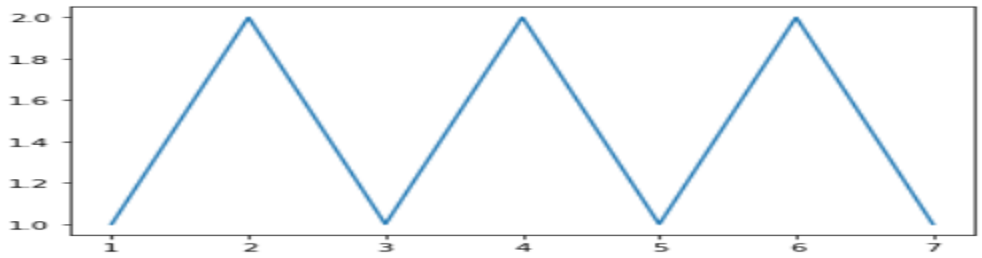
Let see an example of errorbar how it works.

Creating a Simple Graph.

- Python3

```
# importing matplotlib
import matplotlib.pyplot as plt

# making a simple plot
x =[1, 2, 3, 4, 5, 6, 7]
y =[1, 2, 1, 2, 1, 2, 1]

# plotting graph
plt.plot(x, y)
```

Output:



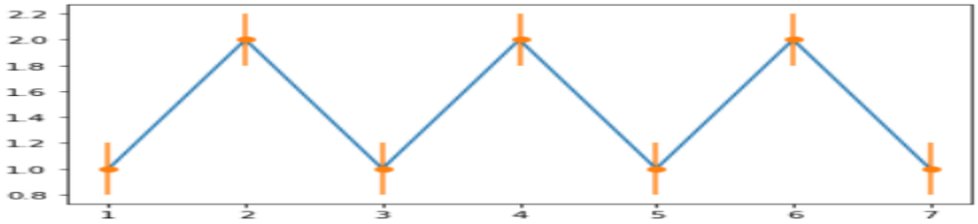
Example 1: Adding Some error in y value.

- Python3

```
# importing matplotlib
```

```
import matplotlib.pyplot as plt


# making a simple plot
x =[1, 2, 3, 4, 5, 6, 7]
y =[1, 2, 1, 2, 1, 2, 1]

# creating error
y_error = 0.2

# plotting graph
plt.plot(x, y)

plt.errorbar(x, y,
             yerr = y_error,
             fmt ='o')
```

Output:

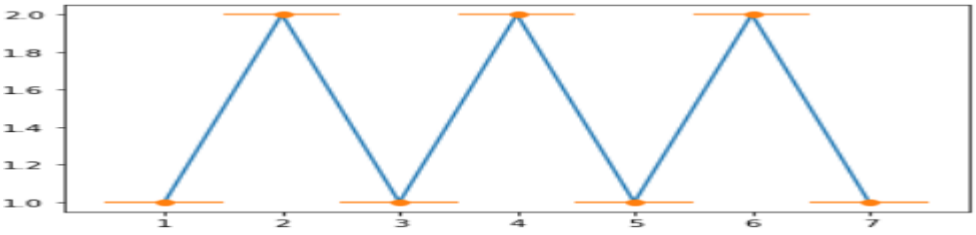


Example 2: Adding Some error in x value.

- Python3

```
# importing matplotlib
import matplotlib.pyplot as plt

# making a simple plot
x =[1, 2, 3, 4, 5, 6, 7]
y =[1, 2, 1, 2, 1, 2, 1]

# creating error
x_error = 0.5

# plotting graph
plt.plot(x, y)
plt.errorbar(x, y,
             xerr = x_error,
             fmt ='o')
```

Output:

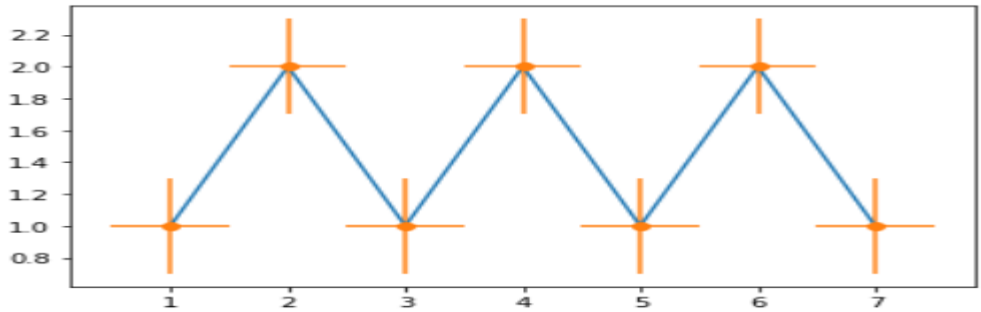


Example 3: Adding error in x & y

- Python3

```
# importing matplotlib
import matplotlib.pyplot as plt


# making a simple plot
x =[1, 2, 3, 4, 5, 6, 7]
y =[1, 2, 1, 2, 1, 2, 1]

# creating error
x_error = 0.5
y_error = 0.3

# plotting graph
plt.plot(x, y)
plt.errorbar(x, y,
             yerr = y_error,
             xerr = x_error,
             fmt ='o')
```

Output:

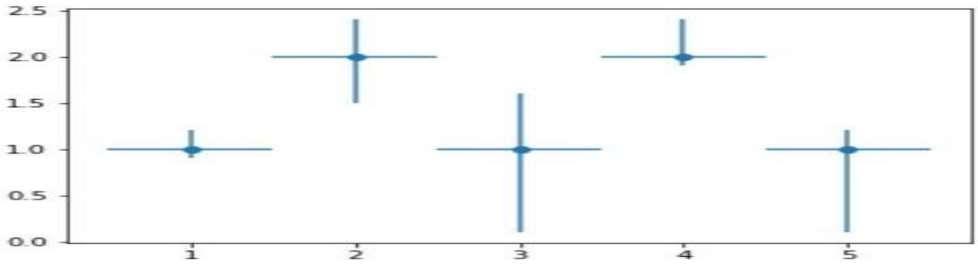


Example 4: Adding variable error in x and y

- Python3

```
# importing matplotlib
import matplotlib.pyplot as plt


# making a simple plot
x =[1, 2, 3, 4, 5]
y =[1, 2, 1, 2, 1]

# creating error
y_errormin =[0.1, 0.5, 0.9,
             0.1, 0.9]
y_errormax =[0.2, 0.4, 0.6,
             0.4, 0.2]

x_error = 0.5
y_error =[y_errormin, y_errormax]

# plotting graph
# plt.plot(x, y)
plt.errorbar(x, y,
             yerr = y_error,
             xerr = x_error,
             fmt ='o')
```

Output:



Colors

Matplotlib recognizes the following formats to specify a color:

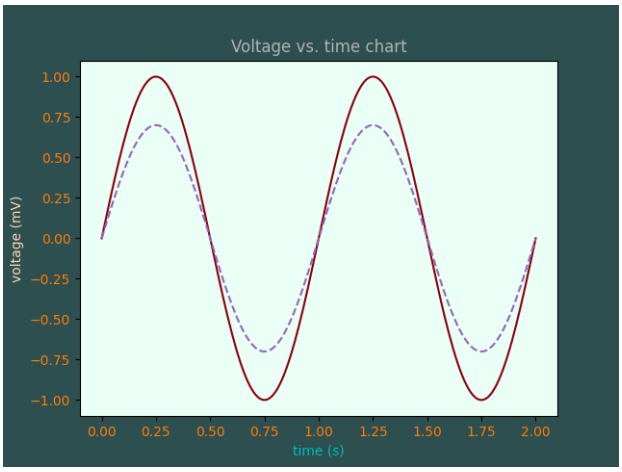
1. an RGB or RGBA tuple of float values in [0, 1] (e.g. (0.1, 0.2, 0.5) or (0.1, 0.2, 0.5, 0.3)). RGBA is short for Red, Green, Blue, Alpha;
2. a hex RGB or RGBA string (e.g., '#0F0F0F' or '#0F0F0F0F');
3. a shorthand hex RGB or RGBA string, equivalent to the hex RGB or RGBA string obtained by duplicating each character, (e.g., '#abc', equivalent to '#aabbcc', or '#abcd', equivalent to '#aabbccdd');
4. a string representation of a float value in [0, 1] inclusive for gray level (e.g., '0.5');

5. a single letter string, i.e. one of {'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}, which are short-hand notations for shades of blue, green, red, cyan, magenta, yellow, black, and white;
6. a X11/CSS4 ("html") color name, e.g. "blue";
7. a name from the xkcd color survey, prefixed with 'xkcd:' (e.g., 'xkcd:sky blue');
8. a "Cn" color spec, i.e. 'C' followed by a number, which is an index into the default property cycle (rcParams["axes.prop_cycle"] (default: cycler('color', ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']))); the indexing is intended to occur at rendering time, and defaults to black if the cycle does not include color.
9. one of {'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan'} which are the Tableau Colors from the 'tab10' categorical palette (which is the default color cycle);

For more information on colors in matplotlib see

- the **matplotlib.colors** API;
- the List of named colors example.

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

t = np.linspace(0.0, 2.0, 201)
s = np.sin(2 * np.pi * t)

# 1) RGB tuple:
fig, ax = plt.subplots(facecolor=(.18, .31, .31))
# 2) hex string:
ax.set_facecolor('#eafff5')
# 3) gray level string:
ax.set_title('Voltage vs. time chart', color='0.7')
# 4) single letter color string
ax.set_xlabel('time (s)', color='c')
# 5) a named color:
ax.set_ylabel('voltage (mV)', color='peachpuff')
# 6) a named xkcd color:
ax.plot(t, s, 'xkcd:crimson')
# 7) Cn notation:
ax.plot(t, .7*s, color='C4', linestyle='--')
# 8) tab notation:
ax.tick_params(labelcolor='tab:orange')
plt.show()
```



Matplotlib Subplot

Display Multiple Plots

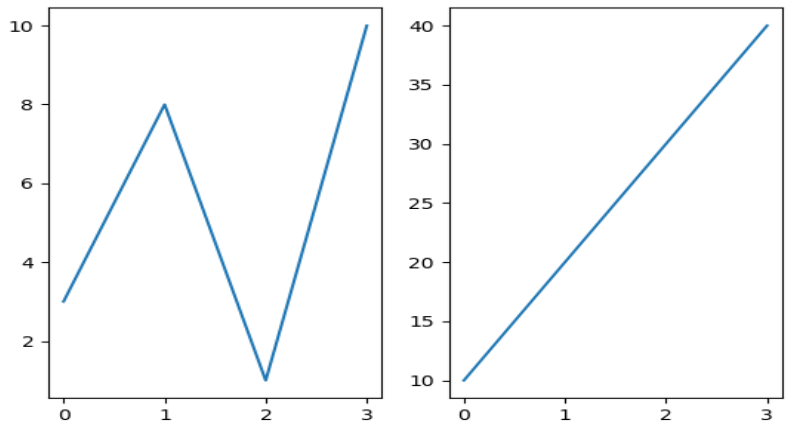
With the subplot() function you can draw multiple plots in one figure:

Example

Draw 2 plots:

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

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)

plt.show()
```

Result:



Text and Annotation

Creating a good visualization involves guiding the reader so that the figure tells a story. In some cases, this story can be told in an entirely visual manner, without the need for added text, but in others, small textual cues and labels are necessary. Perhaps the most basic types of annotations you will use are axes labels and titles, but the options go beyond this. Let's take a look at some data and how we might visualize and annotate it to help convey interesting information. We'll start by setting up the notebook for plotting and importing the functions we will use:

Perhaps the most basic types of annotations you will use are axes labels and titles, but the options go beyond this. Let's take a look at some data and how we might visualize and annotate it to help convey interesting information:

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import numpy as np
import pandas as pd
sns.set()
```

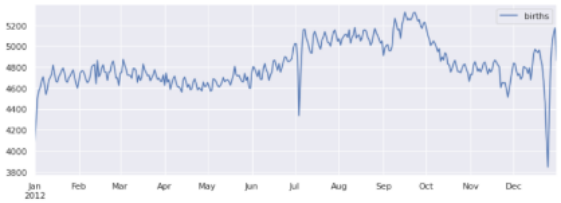
Let's return to some data we worked with earlier in Birthrate Analysis, where we generated a plot of average births over the course of the calendar year; as already mentioned, this data can be downloaded from here.

We'll start with the same cleaning procedure we used there, and plot the results:

```
births = pd.read_csv("births.csv")
births.head()
```

	year	month	day	gender	births
0	1969	1	1.0	F	4046
1	1969	1	1.0	M	4440
2	1969	1	2.0	F	4454
3	1969	1	2.0	M	4548
4	1969	1	3.0	F	4548

```
quartiles = np.percentile(births["births"],[25, 50, 75])
mean, sigma = quartiles[1], 0.74 * (quartiles[2] - quartiles[0])
births = births.query("(births > @mean - 5 * @sigma) & (births < @mean +
births["day"] = births["day"].astype(int)
births.index = pd.to_datetime(10000 * births.year + 100 * births.month +
birthsDate = births.pivot_table("births", [births.index.month, births.inc
birthsDate.index = [pd.datetime(2012, month, day) for (month, day) in bir
fig, ax = plt.subplots(figsize=(12, 4))
birthsDate.plot(ax=ax)
plt.show()
```



When we're communicating data like this, it is often useful to annotate certain features of the plot to draw the reader's attention. This can be done manually with the plt.text/ax.text command, which will place text at a particular x/y value:

```
fig, ax = plt.subplots(figsize=(10,4))
birthsDate.plot(ax=ax)
style = dict(size=10, color='black')
ax.text("2012-1-1", 3950, "New Year", **style)
ax.text("2012-7-14", 4250, "Independence Day", ha="center", **style)
ax.text("2012-9-4", 4850, "Labour Day", ha="center", **style)
ax.text("2012-10-31", 4600, "Halloween", ha="right", **style)
ax.text("2012-12-25", 4450, "Thanksgiving", ha="center", **style)
ax.text("2012-12-25", 3850, "Christmas", ha="right", **style)
# Labeling the axes
ax.set(title= "USA births by day", ylabel="average daily births")
# Format the x axis with centerd month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter("%h"))
plt.show()
```

The ax.text method takes an x position, a y position, a string, and then optional keywords specifying the color, size, style, alignment, and other properties of the text. Here we used ha='right' and ha='center', where ha is short for horizonal alignment.

Transforms and Text Position

It is helpful knowledge to have when considering the placement of text on a figure. There are three predefined transforms that can be useful in this situation: **ax.transData**: Transform associated with data coordinates **ax.transAxes**: Transform associated with the axes (in units of axes dimensions) **fig.transFigure**: Transform associated with the figure (in units of figure dimensions.

Let's demonstrate several of the possible options using the birthrate plot from before:

```
fig, ax = plt.subplots(figsize=(12, 4))
births_date.plot(ax=ax)

# Add labels to the plot
ax.annotate("New Year's Day", xy=('2012-1-1', 4100), xycoords='data',
xytext=(50, -30), textcoords='offset points',
arrowprops=dict(arrowstyle="->",
connectionstyle="arc3,rad=-0.2"))

ax.annotate("Independence Day", xy=('2012-7-4', 4250), xycoords='data',
bbox=dict(boxstyle="round", fc="none", ec="gray"),xytext=(10, -40), textc
arrowprops=dict(arrowstyle="->"))

ax.annotate('Labor Day', xy=('2012-9-4', 4850), xycoords='data', ha='cent
xytext=(0, -20), textcoords='offset points')
ax.annotate('', xy=('2012-9-1', 4850), xytext=('2012-9-7', 4850),
xycoords='data', textcoords='data',
arrowprops={'arrowstyle': '|-|,widthA=0.2,widthB=0.2', })

ax.annotate('Halloween', xy=('2012-10-31', 4600), xycoords='data',
xytext=(-80, -40), textcoords='offset points',
arrowprops=dict(arrowstyle="fancy",
fc="0.6", ec="none",
connectionstyle="angle3,angleA=0,angleB=-90"))
```

```
ax.annotate('Thanksgiving', xy=('2012-11-25', 4500), xycoords='data',
xytext=(-120, -60), textcoords='offset points',
bbox=dict(boxstyle="round4,pad=.5", fc="0.9"),
arrowprops=dict(arrowstyle="->",
connectionstyle="angle,angleA=0,angleB=80,rad=20"))

ax.annotate('Christmas', xy=('2012-12-25', 3850), xycoords='data',
xytext=(-30, 0), textcoords='offset points',
size=13, ha='right', va="center",
bbox=dict(boxstyle="round", alpha=0.1),
arrowprops=dict(arrowstyle="wedge,tail_width=0.5", alpha=0.1))

# Label the axes
ax.set(title='USA births by day of year (1969-1988)',
ylabel='average daily births')

# Format the x axis with centered month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));
ax.set_ylim(3600, 5400)
plt.show()
```

How to change the text alignment?

We specify the `xy` coordinates for the text, but of course, the text can't fit on a single point.

So is the text centered on the point, or is the first letter in the text positioned on that point? Let's see.

```
fig, ax = plt.subplots()
ax.set_title("Different horizonal alignment options when x = .5")
ax.text(.5, .8, 'ha left', fontsize = 12, color = 'red', ha = 'left')
ax.text(.5, .6, 'ha right', fontsize = 12, color = 'green', ha = 'right')
ax.text(.5, .4, 'ha center', fontsize = 12, color = 'blue', ha = 'center')
ax.text(.5, .2, 'ha default', fontsize = 12)
```

```
Text(0.5, 0.2, 'ha default')
```

Creating a text box

The `fontdict` dictionary object allows you to customize the font. Similarly, passing the `bbox` dictionary object allows you to set the properties for a box around the text. Color values between 0 and 1 determine the shade of gray, with 0 being totally black and 1 being totally white. We can also use `boxstyle` to determine the shape of the box. If the `facecolor` is too dark, it can be lightened by trying a value of alpha closer to 0.

```
fig, ax = plt.subplots()
x, y, text = .5, .7, "Text in grey box with\nrectangular box corners."
ax.text(x, y, text,bbox={'facecolor': '.9', 'edgecolor':'blue', 'boxstyle':'square'})
x, y, text = .5, .5, "Text in blue box with\nrounded corners and alpha of .1."
ax.text(x, y, text,bbox={'facecolor': 'blue', 'edgecolor':'none', 'boxstyle':'round', 'alpha' : 0.05})
x, y, text = .1, .3, "Text in a circle.\nalpha of .5 darker\nthan alpha of .1"
ax.text(x, y, text,bbox={'facecolor': 'blue', 'edgecolor':'black', 'boxstyle':'circle', 'alpha' : 0.5})
Text(0.1, 0.3, 'Text in a circle.\nalpha of .5 darker\nthan alpha of .1')
```

Basic annotate method example

Like we said earlier, often you'll want the text to be below or above the point it's labeling. We could do this with the text method, but annotate makes it easier to place text relative to a point. The annotate method allows us to specify two pairs of coordinates. One xy coordinate specifies the point we wish to label. Another xy coordinate specifies the position of the label itself. For example, here we plot a point at (.5,.5) but put the annotation a little higher, at (.5,.503).

```
fig, ax = plt.subplots()
x, y, annotation = .5, .5, "annotation"
ax.title.set_text = "Annotating point (.5,.5) with label located at (.5,.503)"
ax.scatter(x,y)
ax.annotate(annotation,xy=(x,y),xytext=(x,y+.003))
Text(0.5, 0.503, 'annotation')
```

Annotate with an arrow

Okay, so we have a point at xy and an annotation at `xytext`. How can we connect the two?

Can we draw an arrow from the annotation to the point? Absolutely! What we've done with annotate so far looks the same as if we'd just used the text method to put the point at (.5, .503). But annotate can also draw an arrow connecting the label to the point. The arrow is styled by passing a dictionary to `arrowprops`.

```
fig, ax = plt.subplots()
x, y, annotation = .5, .5, "annotation"
ax.scatter(x,y)
ax.annotate(annotation,xy=(x,y),xytext=(x,y+.003),arrowprops={'arrowstyle' : 'simple'})
```
Text(0.5, 0.503, 'annotation')

How can we annotate all the points on a scatter plot?

We can first create 15 test points with associated labels. Then loop through the points and use the annotate method at each point to add a label.

```
import random
random.seed(2)
x = range(15)
y = [element * (2 + random.random()) for element in x]
n = ['label for ' + str(i) for i in x]
fig, ax = plt.subplots()
ax.scatter(x, y)
texts = []
for i, txt in enumerate(n):
ax.annotate(txt, xy=(x[i], y[i]), xytext=(x[i],y[i]+.3))
```

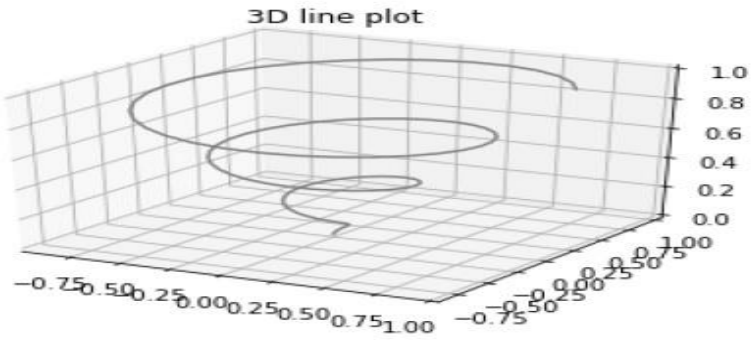
Matplotlib - Three-dimensional Plotting

Even though Matplotlib was initially designed with only two-dimensional plotting in mind, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display in later versions, to provide a set of tools for three-dimensional data visualization. Three-dimensional plots are enabled by importing the **mplot3d toolkit**, included with the Matplotlib package.

A three-dimensional axes can be created by passing the keyword projection='3d' to any of the normal axes creation routines.

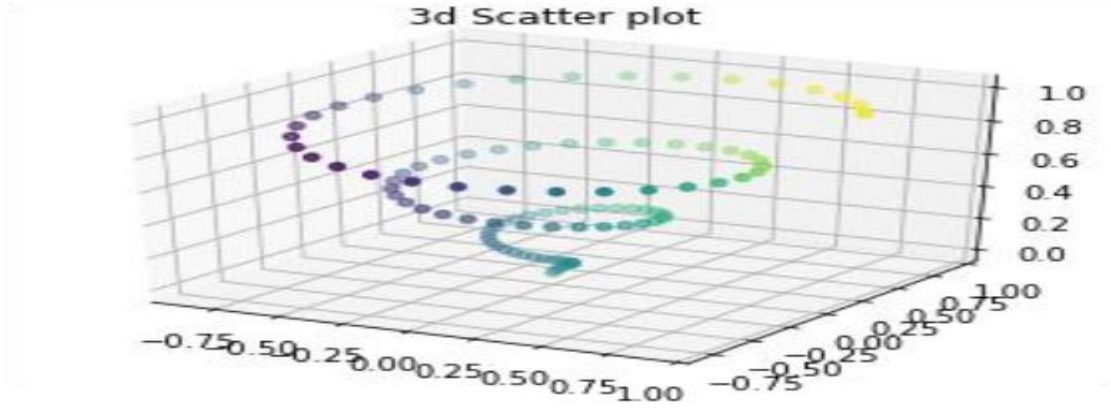
```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(projection='3d')
z = np.linspace(0, 1, 100)
x = z * np.sin(20 * z)
y = z * np.cos(20 * z)
ax.plot3D(x, y, z, 'gray')
ax.set_title('3D line plot')
plt.show()
```

We can now plot a variety of three-dimensional plot types. The most basic three-dimensional plot is a **3D line plot** created from sets of (x, y, z) triples. This can be created using the ax.plot3D function.



3D scatter plot is generated by using the **ax.scatter3D** function.

```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(projection='3d')
z = np.linspace(0, 1, 100)
x = z * np.sin(20 * z)
y = z * np.cos(20 * z)
c = x + y
ax.scatter(x, y, z, c=c)
ax.set_title('3d Scatter plot')
plt.show()
```



Geographic Data with Basemap

One common type of visualization in data science is that of geographic data. Matplotlib's main tool for this type of visualization is the Basemap toolkit, which is one of several

Matplotlib toolkits which lives under the mpl_toolkits namespace. Admittedly, Basemap feels a bit clunky to use, and often even simple visualizations take much longer to render than you might hope. More modern solutions such as leaflet or the Google Maps API may be a better choice for more intensive map visualizations. Still, Basemap is a useful tool for Python users to have in their virtual toolbelts. In this section, we'll show several examples of the type of map visualization that is possible with this toolkit.

Installation of Basemap is straightforward; if you're using conda you can type this and the package will be downloaded:

```
$ conda install basemap
```

We add just a single new import to our standard boilerplate:

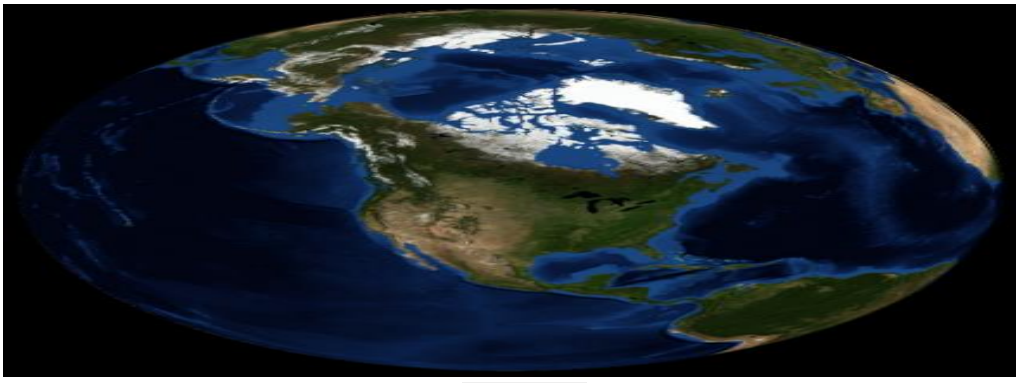
In [1]:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
```

Once you have the Basemap toolkit installed and imported, geographic plots are just a few lines away (the graphics in the following also requires the PIL package in Python 2, or the pillow package in Python 3):

In [2]:

```
plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None, lat_0=50, lon_0=-100)
m.bluemarble(scale=0.5);
```



The meaning of the arguments to Basemap will be discussed momentarily.

The useful thing is that the globe shown here is not a mere image; it is a fully-functioning Matplotlib axes that understands spherical coordinates and which allows us to easily overplot data on the map! For example, we can use a different map projection, zoom-in to North America and plot the location of Seattle. We'll use an etopo image (which shows topographical features both on land and under the ocean) as the map background:

In [3]:

```
fig = plt.figure(figsize=(8, 8))
```

```
m = Basemap(projection='lcc', resolution=None,
        width=8E6, height=8E6,
        lat_0=45, lon_0=-100,)
m.etopo(scale=0.5, alpha=0.5)

# Map (long, lat) to (x, y) for plotting
x, y = m(-122.3, 47.6)
plt.plot(x, y, 'ok', markersize=5)
plt.text(x, y, ' Seattle', fontsize=12);
```

This gives you a brief glimpse into the sort of geographic visualizations that are possible with just a few lines of Python. We'll now discuss the features of Basemap in more depth, and provide several examples of visualizing map data. Using these brief examples as building blocks, you should be able to create nearly any map visualization that you desire.

Map Projections

The first thing to decide when using maps is what projection to use. You're probably familiar with the fact that it is impossible to project a spherical map, such as that of the Earth, onto a flat surface without somehow distorting it or breaking its continuity. These projections have been developed over the course of human history, and there are a lot of choices! Depending on the intended use of the map projection, there are certain map features (e.g., direction, area, distance, shape, or other considerations) that are useful to maintain.

The Basemap package implements several dozen such projections, all referenced by a short format code. Here we'll briefly demonstrate some of the more common ones.

Cylindrical projections

The simplest of map projections are cylindrical projections, in which lines of constant latitude and longitude are mapped to horizontal and vertical lines, respectively. This type of mapping represents equatorial regions quite well, but results in extreme distortions near the poles. The spacing of latitude lines varies between different cylindrical projections, leading to different conservation properties, and different distortion near the poles. In the following

figure we show an example of the *equidistant cylindrical projection*, which chooses a latitude scaling that preserves distances along meridians. Other cylindrical projections are the Mercator (`projection='merc'`) and the cylindrical equal area (`projection='cea'`) projections.

In [5]:

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='cyl', resolution=None,
        llcrnrlat=-90, urcrnrlat=90,
        llcrnrlon=-180, urcrnrlon=180, )
draw_map(m)
```

The additional arguments to Basemap for this view specify the latitude (`lat`) and longitude (`lon`) of the lower-left corner (`llcrnr`) and upper-right corner (`urcrnr`) for the desired map, in units of degrees.

Pseudo-cylindrical projections

Pseudo-cylindrical projections relax the requirement that meridians (lines of constant longitude) remain vertical; this can give better properties near the poles of the projection. The Mollweide projection (`projection='moll'`) is one common example of this, in which all meridians are elliptical arcs. It is constructed so as to preserve area across the map: though there are distortions near the poles, the area of small patches reflects the true area. Other pseudo-cylindrical projections are the sinusoidal (`projection='sinu'`) and Robinson (`projection='robin'`) projections.

In [6]:

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='moll', resolution=None,
        lat_0=0, lon_0=0)
draw_map(m)
```

The extra arguments to Basemap here refer to the central latitude (lat_0) and longitude (lon_0) for the desired map.

Perspective projections

Perspective projections are constructed using a particular choice of perspective point, similar to if you photographed the Earth from a particular point in space (a point which, for some projections, technically lies within the Earth!). One common example is the orthographic projection (projection='ortho'), which shows one side of the globe as seen from a viewer at a very long distance. As such, it can show only half the globe at a time. Other perspective-based projections include the gnomonic projection (projection='gnom') and stereographic projection (projection='stere'). These are often the most useful for showing small portions of the map.

Here is an example of the orthographic projection:

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
        lat_0=50, lon_0=0)
draw_map(m);
```

The following are some of the available drawing functions that you may wish to explore using IPython's help features:

- **Physical boundaries and bodies of water**
  - drawcoastlines(): Draw continental coast lines
  - drawlsmask(): Draw a mask between the land and sea, for use with projecting images on one or the other
  - drawmapboundary(): Draw the map boundary, including the fill color for oceans.
  - drawrivers(): Draw rivers on the map
  - fillcontinents(): Fill the continents with a given color; optionally fill lakes with another color
- **Political boundaries**
  - drawcountries(): Draw country boundaries
  - drawstates(): Draw US state boundaries
  - drawcounties(): Draw US county boundaries
- **Map features**
  - drawgreatcircle(): Draw a great circle between two points
  - drawparallels(): Draw lines of constant latitude
  - drawmeridians(): Draw lines of constant longitude
  - drawmapscale(): Draw a linear scale on the map
- **Whole-globe images**
  - bluemarble(): Project NASA's blue marble image onto the map

- shadedrelief(): Project a shaded relief image onto the map
- etopo(): Draw an etopo relief image onto the map
- warpimage(): Project a user-provided image onto the map

Plotting Data on Maps

Perhaps the most useful piece of the Basemap toolkit is the ability to over-plot a variety of data onto a map background. For simple plotting and text, any plt function works on the map; you can use the Basemap instance to project latitude and longitude coordinates to (x, y) coordinates for plotting with plt, as we saw earlier in the Seattle example.

In addition to this, there are many map-specific functions available as methods of the Basemap instance. These work very similarly to their standard Matplotlib counterparts, but have an additional Boolean argument latlon, which if set to True allows you to pass raw latitudes and longitudes to the method, rather than projected (x, y) coordinates.

Some of these map-specific methods are:

- contour()/contourf() : Draw contour lines or filled contours
- imshow(): Draw an image
- pcolor()/pcolormesh() : Draw a pseudocolor plot for irregular/regular meshes
- plot(): Draw lines and/or markers.
- scatter(): Draw points with markers.
- quiver(): Draw vectors.
- barbs(): Draw wind barbs.
- drawgreatcircle(): Draw a great circle.

We'll see some examples of a few of these as we continue. For more information on these functions, including several example plots, see the online Basemap documentation.

Example: California Cities

Recall that in Customizing Plot Legends, we demonstrated the use of size and color in a scatter plot to convey information about the location, size, and population of California cities. Here, we'll create this plot again, but using Basemap to put the data in context.

We start with loading the data, as we did before:

In [10]:

```
import pandas as pd
cities = pd.read_csv('data/california_cities.csv')

# Extract the data we're interested in
lat = cities['latd'].values
lon = cities['longd'].values
population = cities['population_total'].values
area = cities['area_total_km2'].values
```

Next, we set up the map projection, scatter the data, and then create a colorbar and legend:

In [11]:

```
# 1. Draw the map background
```

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution='h',
            lat_0=37.5, lon_0=-119,
            width=1E6, height=1.2E6)
m.shadedrelief()
m.drawcoastlines(color='gray')
m.drawcountries(color='gray')
m.drawstates(color='gray')

# 2. scatter city data, with color reflecting population
# and size reflecting area
m.scatter(lon, lat, latlon=True,
          c=np.log10(population), s=area,
          cmap='Reds', alpha=0.5)

# 3. create colorbar and legend
plt.colorbar(label=r'$\log_{10}({\rm population})$')
plt.clim(3, 7)

# make legend with dummy points
for a in [100, 300, 500]:
    plt.scatter([], [], c='k', alpha=0.5, s=a,
                label=str(a) + ' km$^2$')
plt.legend(scatterpoints=1, frameon=False,
           labelspacing=1, loc='lower left');
```

This shows us roughly where larger populations of people have settled in California: they are clustered near the coast in the Los Angeles and San Francisco areas, stretched along the highways in the flat central valley, and avoiding almost completely the mountainous regions along the borders of the state.

Visualization with Seaborn

Seaborn provides an API on top of Matplotlib that offers sane choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas DataFrames.

To be fair, the Matplotlib team is addressing this: it has recently added the plt.style tools discussed in Customizing Matplotlib: Configurations and Style Sheets, and is starting to handle Pandas data more seamlessly. The 2.0 release of the library will include a new default stylesheet that will improve on the current status quo. But for all the reasons just discussed, Seaborn remains an extremely useful addon.

Seaborn Versus Matplotlib

Here is an example of a simple random-walk plot in Matplotlib, using its classic plot formatting and colors. We start with the typical imports:

In [1]:

```
import matplotlib.pyplot as plt
plt.style.use('classic')
%matplotlib inline
import numpy as np
import pandas as pd
```

Now we create some random walk data:

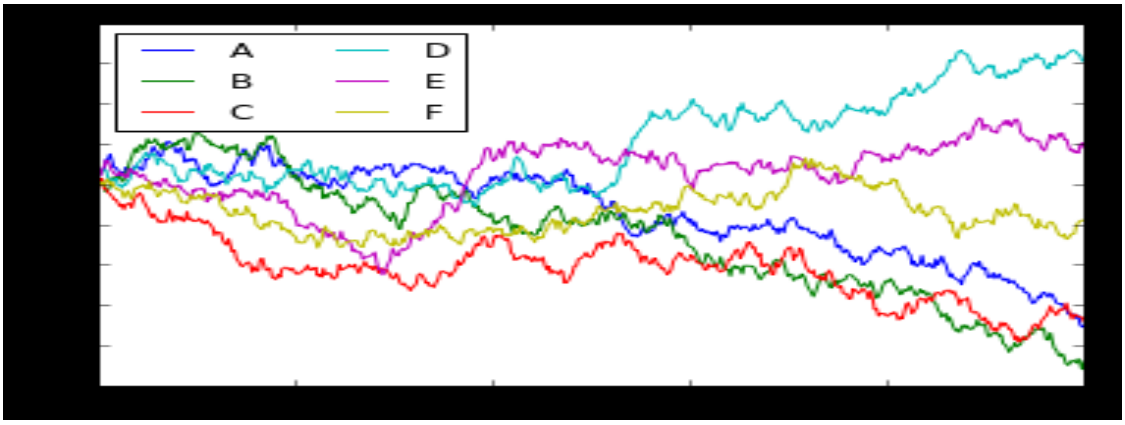
In [2]:

```
# Create some data
rng = np.random.RandomState(0)
x = np.linspace(0, 10, 500)
y = np.cumsum(rng.randn(500, 6), 0)
```

And do a simple plot:

In [3]:

```
# Plot the data with Matplotlib defaults
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left');
```



Although the result contains all the information we'd like it to convey, it does so in a way that is not all that aesthetically pleasing, and even looks a bit old-fashioned in the context of 21st-century data visualization.

Now let's take a look at how it works with Seaborn. As we will see, Seaborn has many of its own high-level plotting routines, but it can also overwrite Matplotlib's default parameters and in turn get even simple Matplotlib scripts to produce vastly superior output. We can set the style by calling Seaborn's `set()` method. By convention, Seaborn is imported as `sns`:

```
import seaborn as sns
sns.set()
```

Now let's rerun the same two lines as before:

In [5]:

```
# same plotting code as above!
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left');
```

Exploring Seaborn Plots

The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

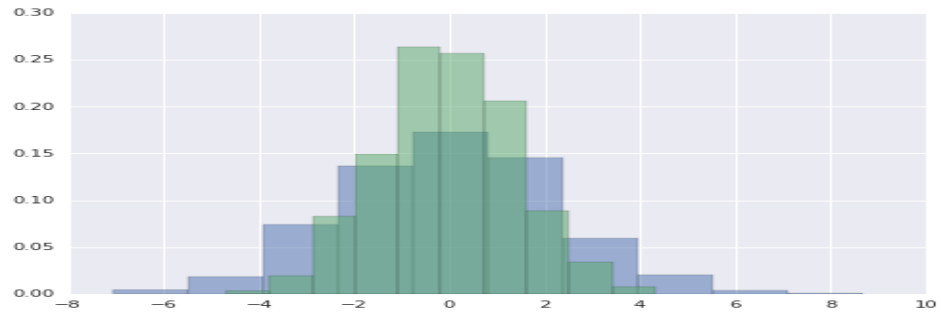
Let's take a look at a few of the datasets and plot types available in Seaborn. Note that all of the following *could* be done using raw Matplotlib commands (this is, in fact, what Seaborn does under the hood) but the Seaborn API is much more convenient.

Histograms, KDE, and densities

Often in statistical data visualization, all you want is to plot histograms and joint distributions of variables. We have seen that this is relatively straightforward in Matplotlib:

In [6]:

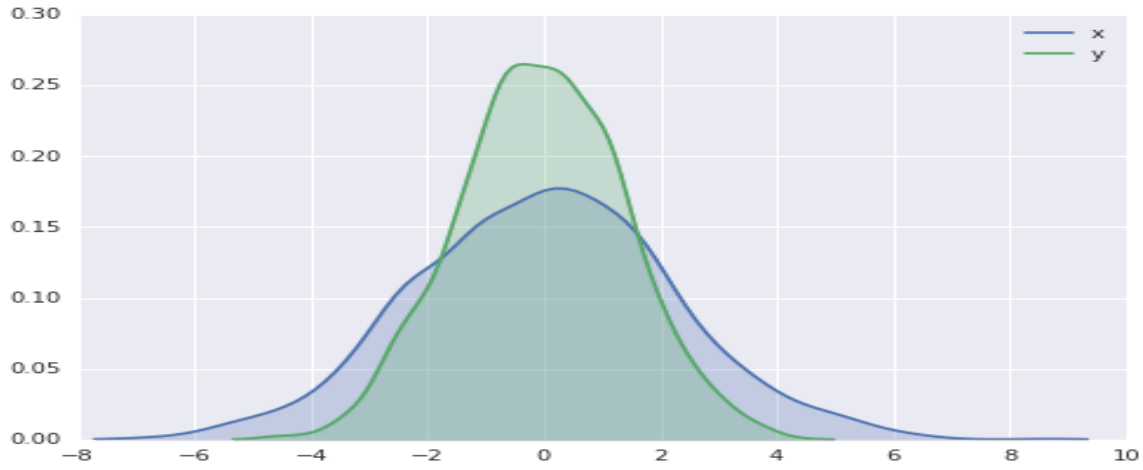
```
data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
data = pd.DataFrame(data, columns=['x', 'y'])
for col in 'xy':
    plt.hist(data[col], normed=True, alpha=0.5)
```



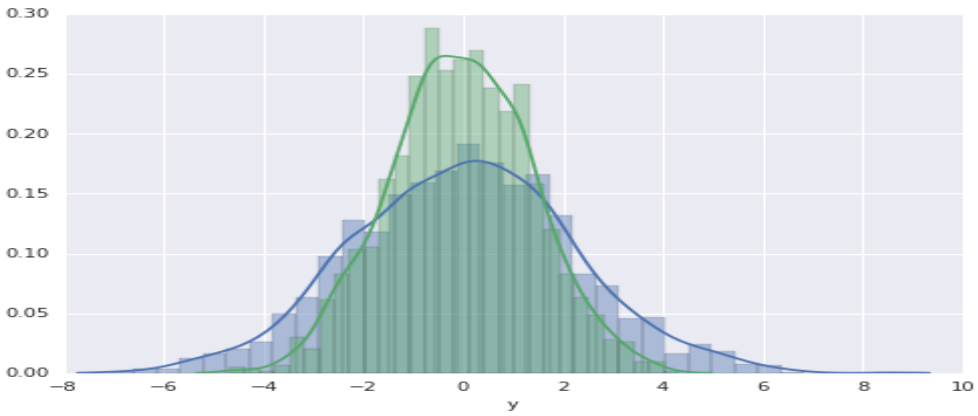
Rather than a histogram, we can get a smooth estimate of the distribution using a kernel density estimation, which Seaborn does with `sns.kdeplot`:

In [7]:

```
for col in 'xy':
  sns.kdeplot(data[col], shade=True)
```



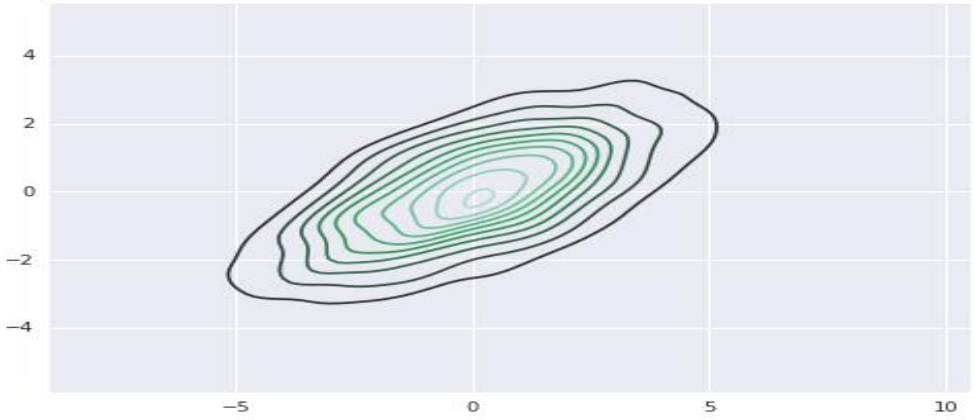

```
sns.distplot(data['x'])
sns.distplot(data['y']);
```



If we pass the full two-dimensional dataset to `kdeplot`, we will get a two-dimensional visualization of the data:

In [9]:

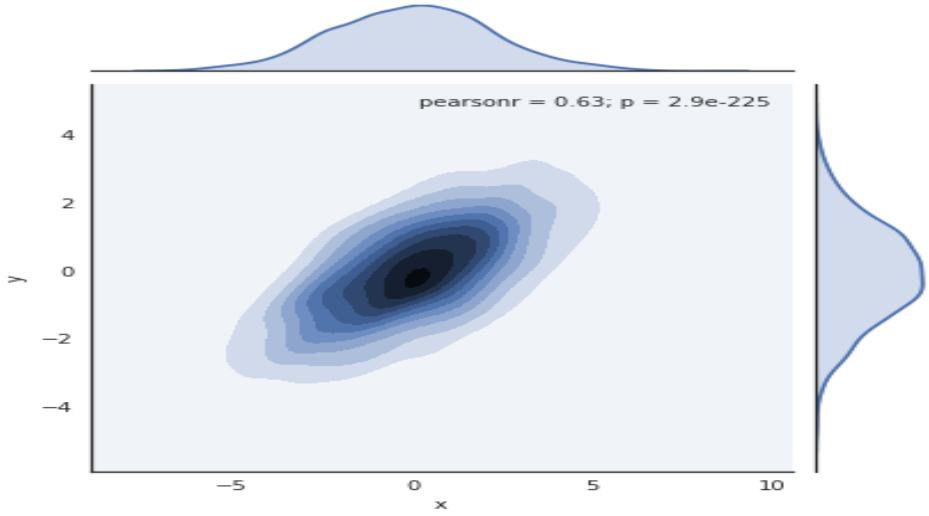
```
sns.kdeplot(data);
```



We can see the joint distribution and the marginal distributions together using sns.jointplot. For this plot, we'll set the style to a white background:

In [10]:

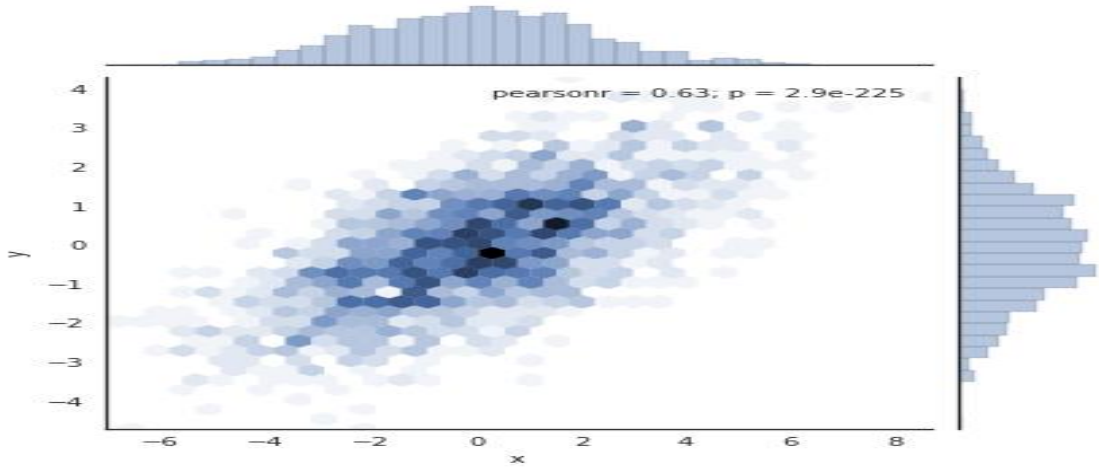
```
with sns.axes_style('white'):
  sns.jointplot("x", "y", data, kind='kde');
```



here are other parameters that can be passed to jointplot—for example, we can use a hexagonally based histogram instead:

In [11]:

```
with sns.axes_style('white'):
  sns.jointplot("x", "y", data, kind='hex')
```



Part – A

1. What is the use of matplotlib and Pyplot?
2. Define pie chart.
3. Explain about Bar chart
4. What is histogram? Give example
5. List the types of plots.
6. Write a python code for plotting a line.
7. Write a python code for drawing a scatter plot.
8. List any five 2D line properties.
9. Define legend. State how legends are made.
10. State the importance of subplots.
11. What is legend handler?
12. Write the code for drawing multiple plots.
13. Write the syntax for specifying colors in matplotlib.
14. What is the role of text method?
15. Write the code for changing the font size and font color.
16. How to change the text alignment?
17. Give example for basic annotate method.
18. How can we annotate all the points on a scatter plot?
19. How to prevent overlapping annotations
20. Write the code for 3D line plot.

21. Write the code snippet for plotting a 3D scatter plot.
22. Define basemap.
23. List various map projections.
24. Define Seaborn.
25. Compare matplotlib with seaborn.

Part – B

1. Write a python program to plot histogram and box plot.
2. Write a python program to plot scatter plot and area plot.
3. Write a program to construct multiple plots.
4. Explain in detail about all the attributed of 2D line.
5. Discuss in detail about Matplotlib.pyplot package with suitable code.
6. Explain about Text and annotations with suitable code.
7. Write about Three dimensional plotting with suitable examples.
8. Write the code for customization of line chart with color, style and width.
9. Discuss in detail about various projections of map with suitable code.
10. Discuss about Data Visualization using seaborn API.