Matplotlib

Lecturer: Ben Galili

Experis KickStart
ManpowerGroup

Matplotlib

- Matplotlib is a multi-platform plotting and data visualization library for Python
- It provides a means of producing graphical plots that can be embedded into applications, displayed on the screen or output as high-quality image files for publication
- Conceived by John Hunter in 2002
- Matplotlib is designed to be as usable as MATLAB, with the ability to use Python, and the advantage of being free and open-source

Importing Matplotlib

- Just as we use the np shorthand for NumPy, we will use a standard shorthands for Matplotlib import:

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

- We will use the plt.style directive to choose appropriate aesthetic styles for our figures
- Here we will set the classic style, which ensures that the plots we create use the classic Matplotlib style:

```
plt.style.use('classic')
```

Plotting from a Script

- If you are using Matplotlib from within a script, the function **plt.show()** is your friend
- plt.show() starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display your figure or figures
- For example, you may have a file called *myplot.py* containing the following:

```
# myplot.py
import matplotlib.pyplot as plt
import numpy as np
plt.style.use('classic')

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--')

plt.show()
```

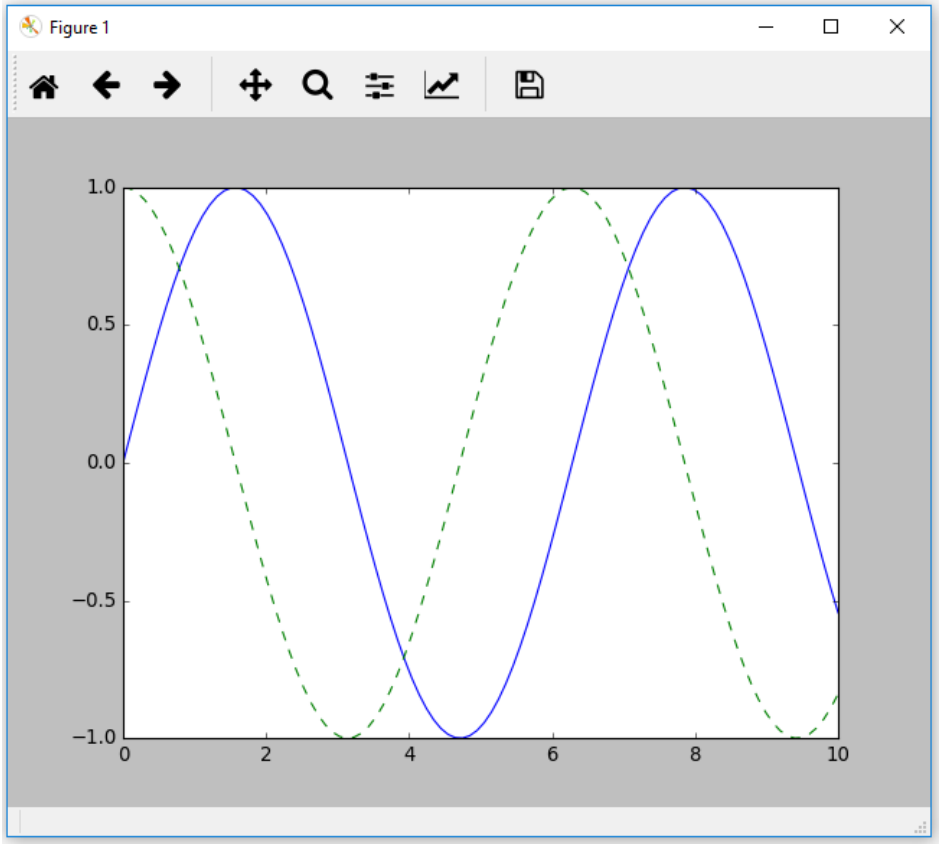
plt.plot() creates the necessary figure and axes to plot y vs. x as lines and/or markers

The plt.show() command should be used only once per Python session, and is most often seen at the very end of the script

Plotting from a Script

- You can then run this script from the command-line prompt, which will result in a window opening with your figure displayed:

```
C:\PycharmProjects\Matplotlib>python myplot.py
```



Plotting from an IPython Notebook

- Plotting interactively within an IPython notebook can be done with the %matplotlib magic command:

```
%matplotlib
```

```
Using matplotlib backend: Qt5Agg
```

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

- At this point, any plt.plot() command will cause a figure window to open, and further commands can be run to update the plot
- Using plt.show() in matplotlib mode is not required
- Some changes (such as modifying properties of lines that are already drawn) will not draw automatically: to force an update, use plt.draw()

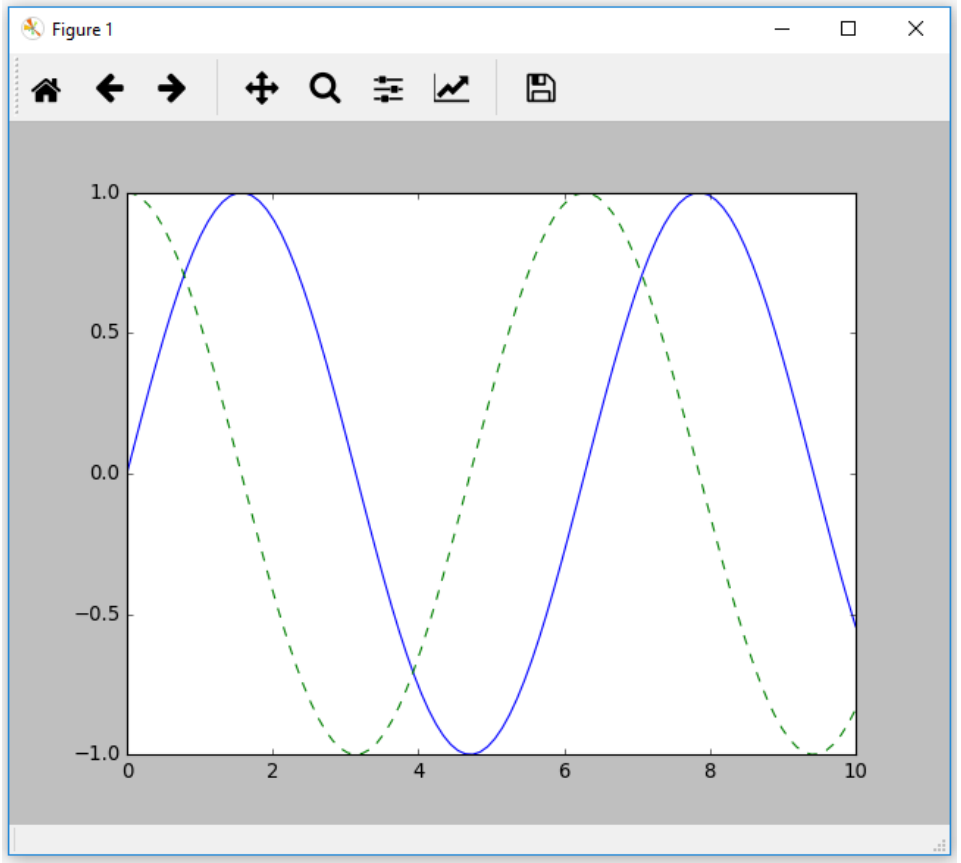
Plotting from an IPython Notebook

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

```
Using matplotlib backend: Qt5Agg
```

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--')
```

```
[<matplotlib.lines.Line2D at 0x1cf5c595278>]
```



Plotting from an IPython Notebook

Experis KickStart ManpowerGroup

- In the IPython notebook, you also have the option of embedding graphics directly in the notebook, with two possible options:
  - **%matplotlib notebook** will lead to *interactive* plots embedded within the notebook
  - **%matplotlib inline** will lead to *static* images of your plot embedded in the notebook
- We'll usually use the inline option:

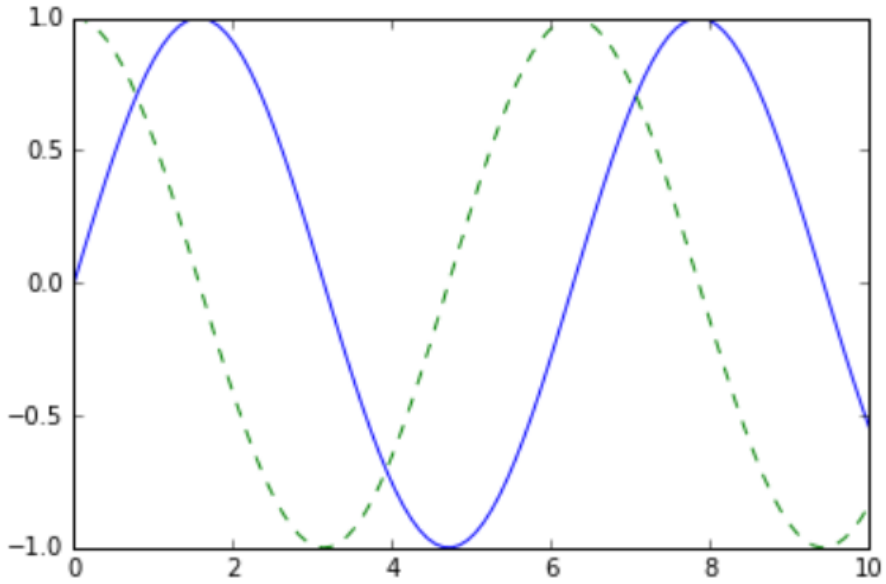
```
%matplotlib inline
```

- After running this command, any cell within the notebook that creates a plot will embed a PNG image of the resulting graphic
- This command needs to be done only once per kernel/session
- By default, when the IPython kernel is started it internally runs %matplotlib inline
  - However, it's better to specify this option explicitly, in case that another Notebook used a different option and was executed before your notebook

Plotting from an IPython Notebook

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

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
```



The ; at the end of the cell allows you to suppress the printing of the output

Saving Figures to File

- Saving a figure can be done using the **savefig()** command of the Figure object
- First you have to store the figure object in a variable (before calling plot)
- For example, to save the previous figure as a PNG file, you can run this:

```
x = np.linspace(0, 10, 100)

fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--')

fig.savefig('my_figure.png');
```

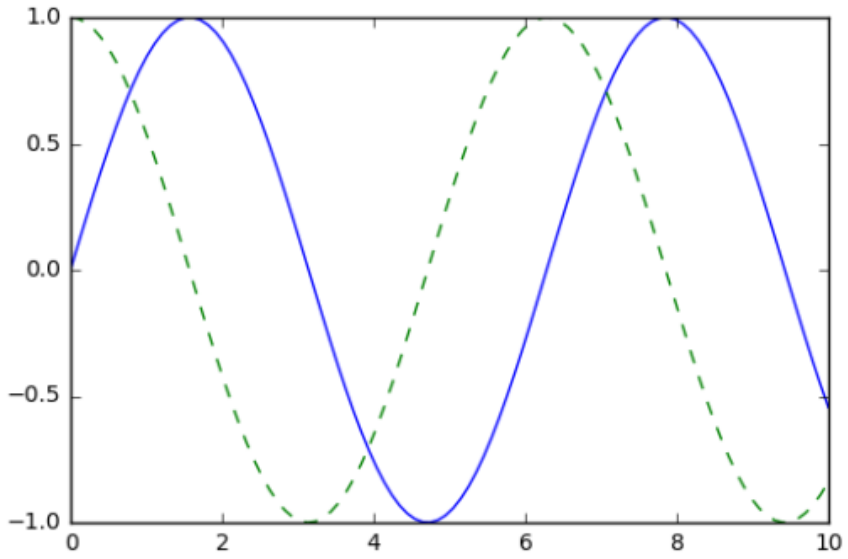
- We now have a file called my_figure.png in the current working directory:

```
C:\Notebooks\Matplotlib>dir my_figure.png
 Volume in drive C has no label.
 Volume Serial Number is 5CAF-CB37

 Directory of C:\Notebooks\Matplotlib

09/09/2018  07:32 AM            26,306 my_figure.png
               1 File(s)         26,306 bytes
               0 Dir(s)  31,436,197,888 bytes free
```

Saving Figures to File

- To confirm that it contains what we think it contains, let's use the IPython Image object to display the contents of this file:

```
from IPython.display import Image
Image('my_figure.png')
```



Saving Figures to File

- In savefig(), the file format is inferred from the extension of the given filename
- Depending on the backends you have, many different file formats are available
- The list of supported file types can be found for your system by using the following method of the figure canvas object:

```
fig.canvas.get_supported_filetypes()
```

```
{'eps': 'Encapsulated Postscript',
 'jpeg': 'Joint Photographic Experts Group',
 'jpg': 'Joint Photographic Experts Group',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'ps': 'Postscript',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}
```

Two Matplotlib Interfaces

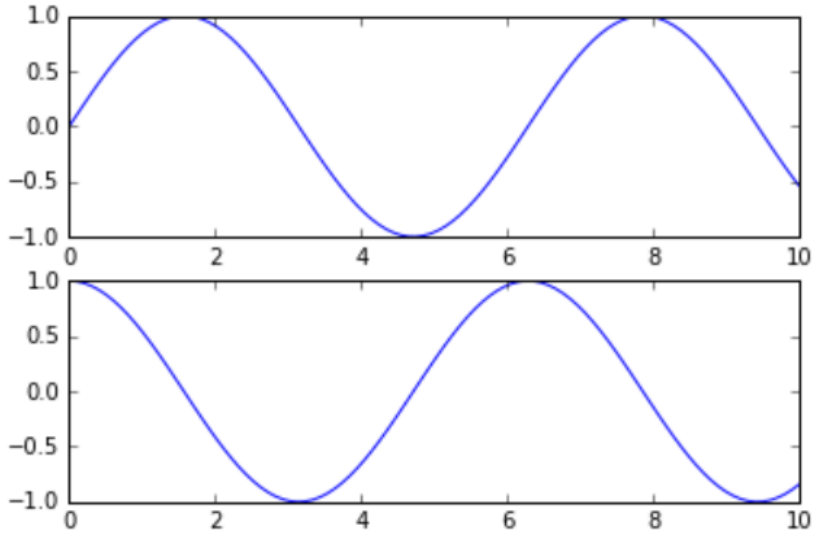
Experis KickStart ManpowerGroup

- A potentially confusing feature of Matplotlib is its dual interfaces:
  - a convenient procedural, MATLAB-style state-based interface, which supports many MATLAB functions in Matlab with the same names and arguments
  - a more powerful object-oriented interface, which allows more advanced and customizable use

MATLAB-Style Interface

- The MATLAB-style tools are contained in the **pyplot** (plt) interface
- In this mode, there is only one active figure and one axis at each time point
  - Each pyplot function makes some change to that figure, e.g., plots a line on some axis
  - The state of the figure is preserved across function calls
- For example, the following code will probably look familiar to MATLAB users:

```
# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```



Object-Oriented Interface

- The object-oriented interface is available for these more complicated situations, and for when you want more control over your figure
- Rather than depending on some notion of an "active" figure or axes, in the OO interface the plotting functions are *methods* of explicit Figure and Axes objects
- To re-create the previous plot using this style of plotting, you might do the following:

```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```

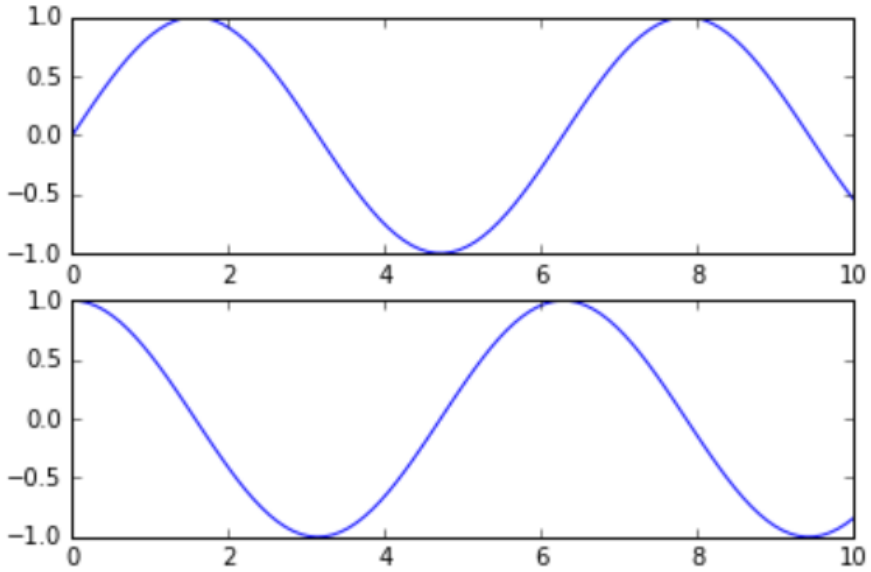


Figure and Axes

- For all Matplotlib plots, we start by creating a figure and an axes
- Start by setting up the notebook for plotting and importing the packages we will use:

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

- The *figure* (an instance of plt.Figure) is the top-level object, containing all the elements of the plot including axes, graphics, text, and labels
- The *axes* (an instance of plt.Axes) is a bounding box with ticks and labels, which contains the plot elements that make up our visualization

Figure and Axes

- In their simplest form, a figure and axes can be created as follows:

```
fig = plt.figure()
ax = plt.axes()
```

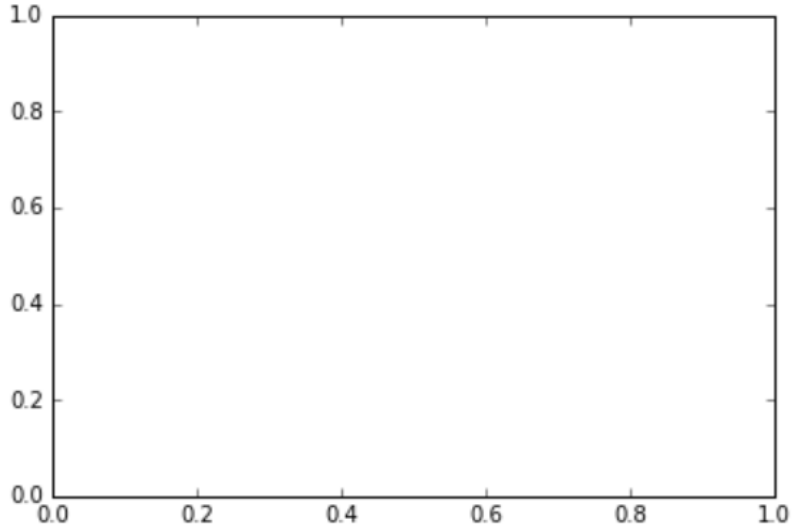


Figure Properties

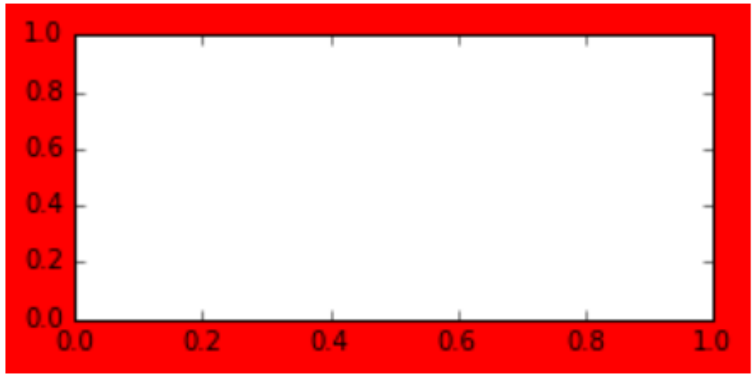
- The figure object can be customized using the following properties:

Argument	Description
num	An identifier for the figure – if none is provided, an integer, starting at 1, is used and incremented with each figure created. Alternatively using a string will set the window title to that string when the figure is displayed with plt.show().
figsize	A tuple of figure (width, height), in inches.
dpi	Figure resolution in dots-per-inch
facecolor	Figure background color
edgecolor	Figure border color

Figure Properties

- For example, to create a small figure with a red background:

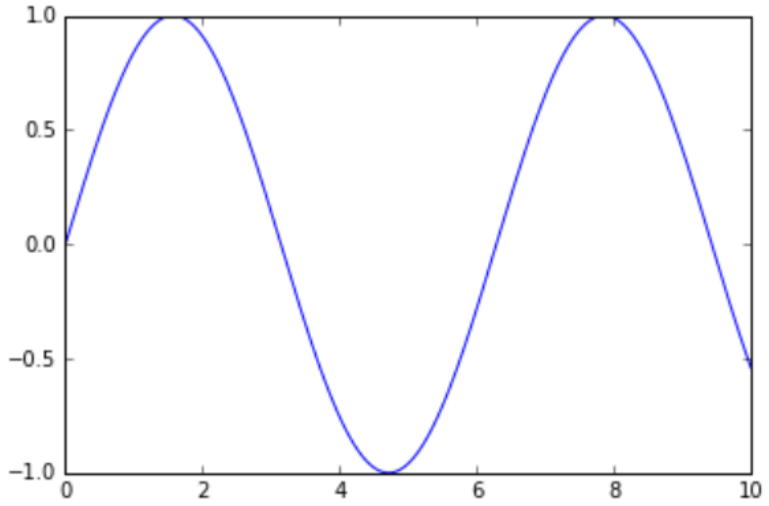
```
fig = plt.figure('Population density', figsize=(4.5, 2),
                 facecolor='red')
ax = plt.axes()
```



Line Plots

- Once we have created an axes, we can use the **ax.plot()** function to plot some data
- Perhaps the simplest of all plots is the visualization of a single function $y = f(x)$
- Let's start with a simple sinusoid:

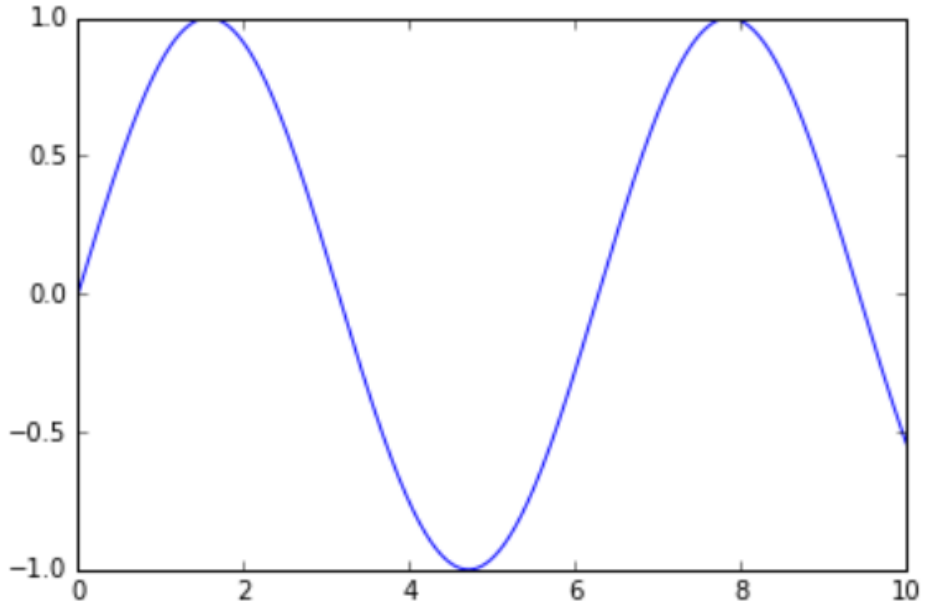
```
fig = plt.figure()
ax = plt.axes()

x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```



Line Plots

- Alternatively, we can use the **pylab** interface and let the figure and axes be created for us in the background, by calling **plt.plot()**:

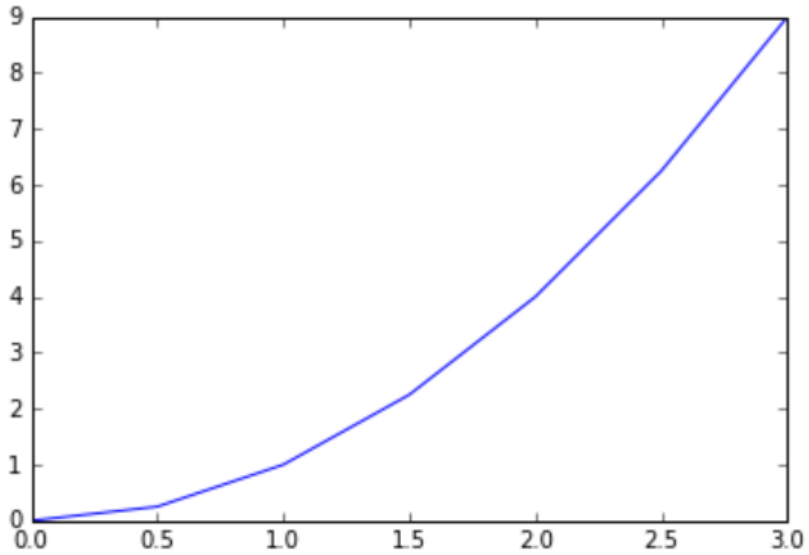
```
plt.plot(x, np.sin(x));
```



Line Plots

- The plt.plot() function accepts any two iterable objects of the same length (typically lists or NumPy arrays)

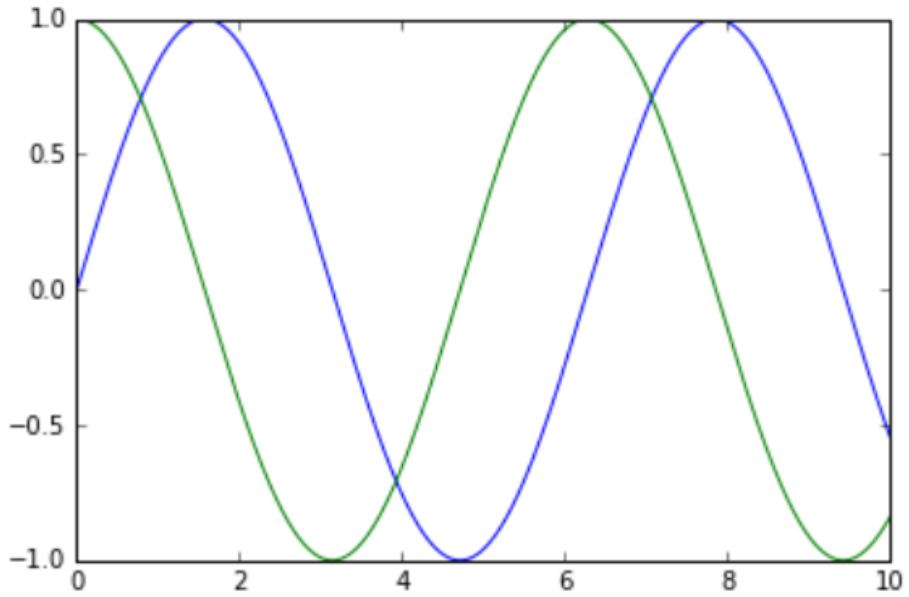
```
x = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
y = [0.0, 0.25, 1.0, 2.25, 4.0, 6.25, 9.0]

plt.plot(x, y);
```



Line Plots

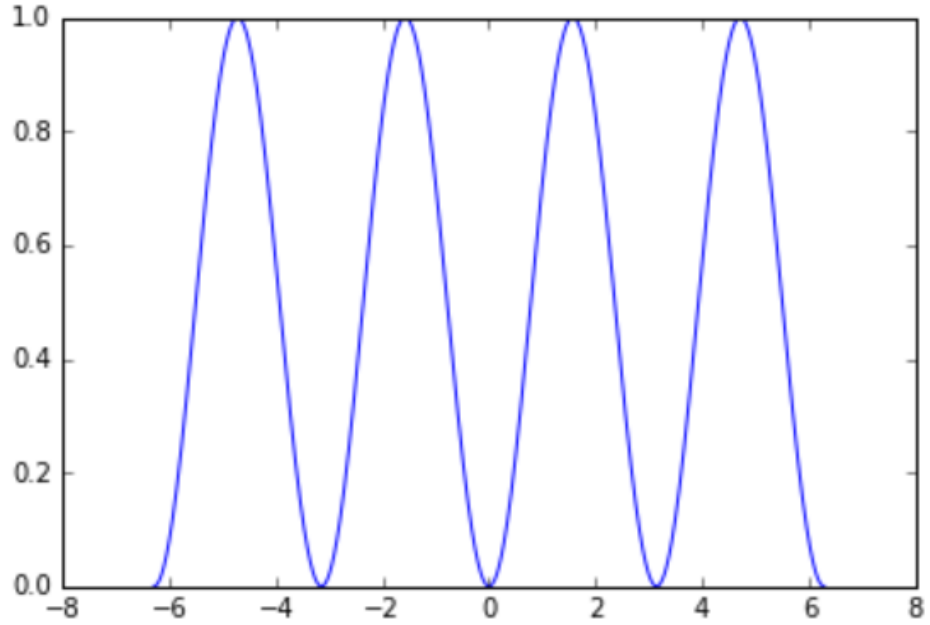
- If we want to create a single figure with multiple lines, we can simply call the plot() function multiple times:

```
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```

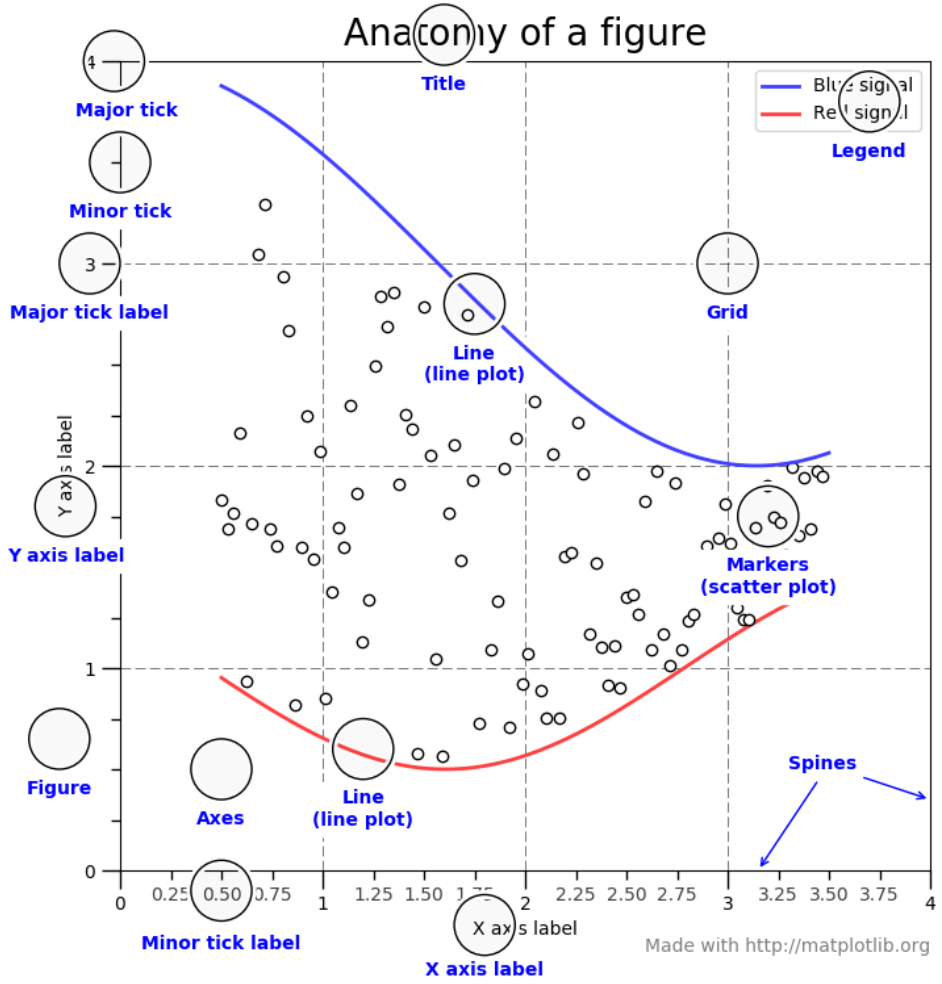


Exercise

- Plot the function $f(x) = \sin^2 x$ for 1,000 points across the range $-2\pi \leq x \leq 2\pi$
- Save the figure to a file



The Anatomy of a Figure



Adjusting the Plot: Line Colors and Styles

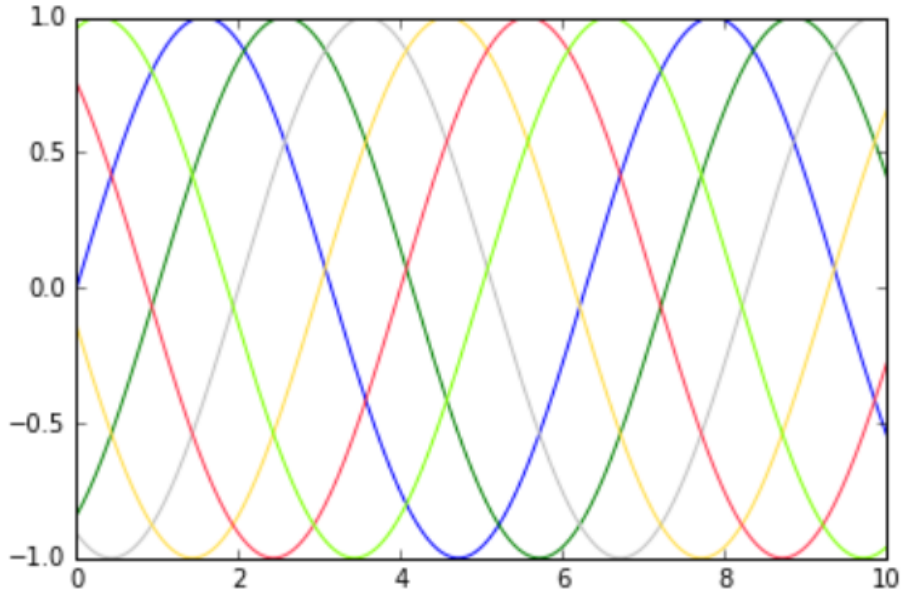
- The first adjustment you might wish to make is to control the line colors and styles
- The plt.plot() function takes additional arguments that can be used to specify these
- To adjust the color, you can use the **color** keyword, which accepts a string argument
- The color be specified in a variety of ways:
  - The name of the color, e.g. “blue”
  - Short color code (rgbcmyk)
  - Grayscale between 0 and 1
  - Hex code (RRGGBB from 00 to FF)
  - RGB tuple, values 0 to 1, e.g. (0.5, 0, 0) is a dark red color
  - All HTML color names supported

Code	Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

- If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines

Line Colors

```
plt.plot(x, np.sin(x - 0), color='blue')        # specify color by name
plt.plot(x, np.sin(x - 1), color='g')           # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')        # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # Hex code (RRGGBB)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 to 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # HTML color names supported
```



Line Styles

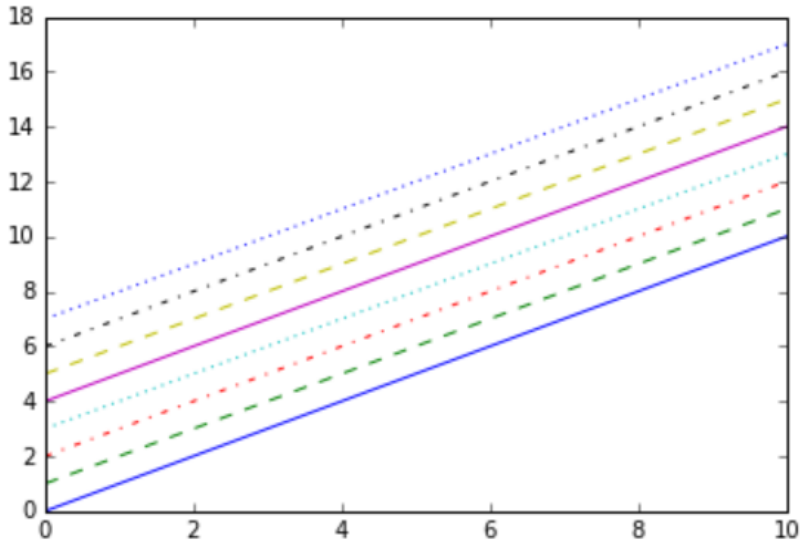
- Similarly, the line style can be adjusted using the **linestyle** keyword (also a string)
- Some of the possible line styles settings are given in the following table:

Code	Line style
-	solid
--	dashed
:	dotted
-.	dash-dot

- To draw no line at all, set linestyle=’’ (the empty string)
- The thickness of a line can be specified in points by passing a float to **linewidth**
- The default plot line style is a solid line of weight 1.0 pt

Line Styles

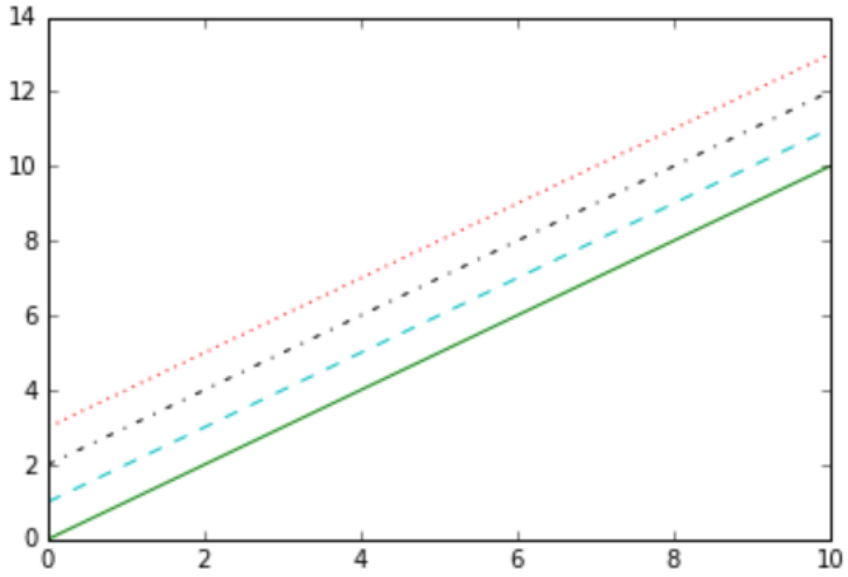
```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');

# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':');  # dotted
```



Line Color and Styles

- If you would like to be extremely terse, these linestyle and color codes can be combined into a single non-keyword argument to the plt.plot() function:

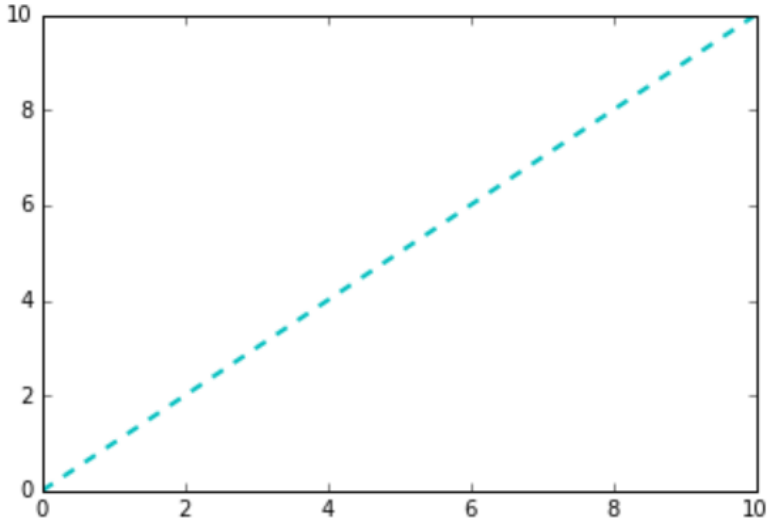
```
plt.plot(x, x + 0, '-g')  # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r'); # dotted red
```



Line Color and Styles

- The following abbreviations for the plot line properties are also valid:
  - c for color
  - ls for linestyle
  - lw for linewidth

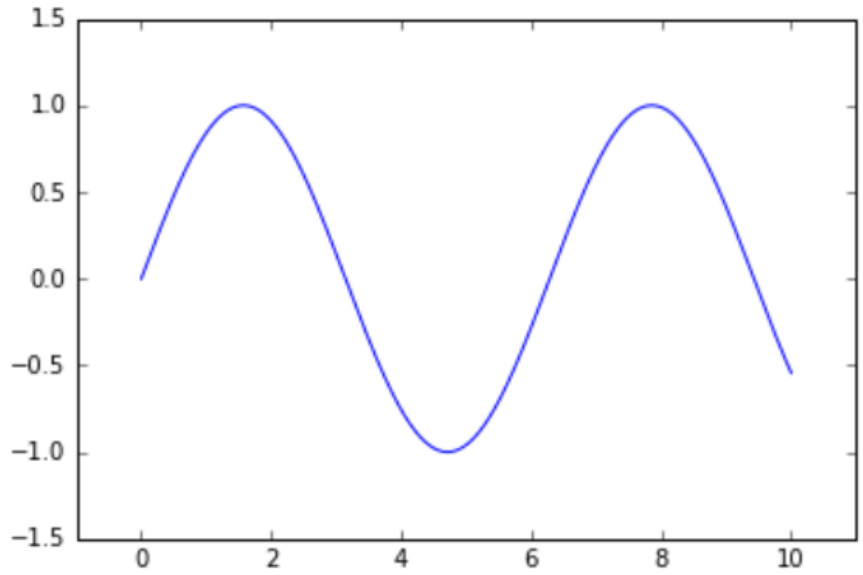
```
plt.plot(x, x + 0, c='c', ls='--', lw=2); # a thick, cyan, dashed line
```



Axes Limits

- Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes it's nice to have finer control
- You can adjust axis limits by using the **plt.xlim()** and **plt.ylim()** methods:

```
plt.plot(x, np.sin(x))

plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```

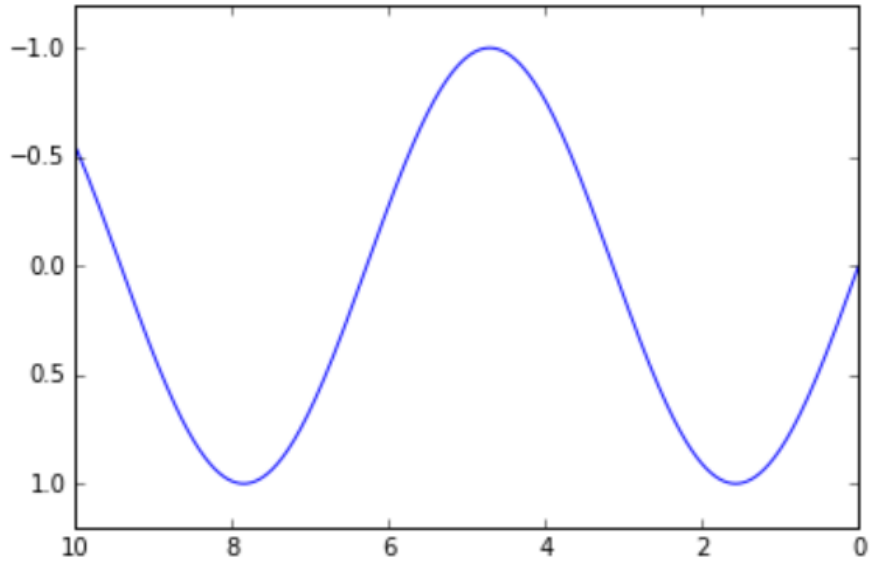


Axes Limits

- If for some reason you'd like either axis to be displayed in reverse, you can simply reverse the order of the arguments:

```
plt.plot(x, np.sin(x))

plt.xlim(10, 0)
plt.ylim(1.2, -1.2)
```

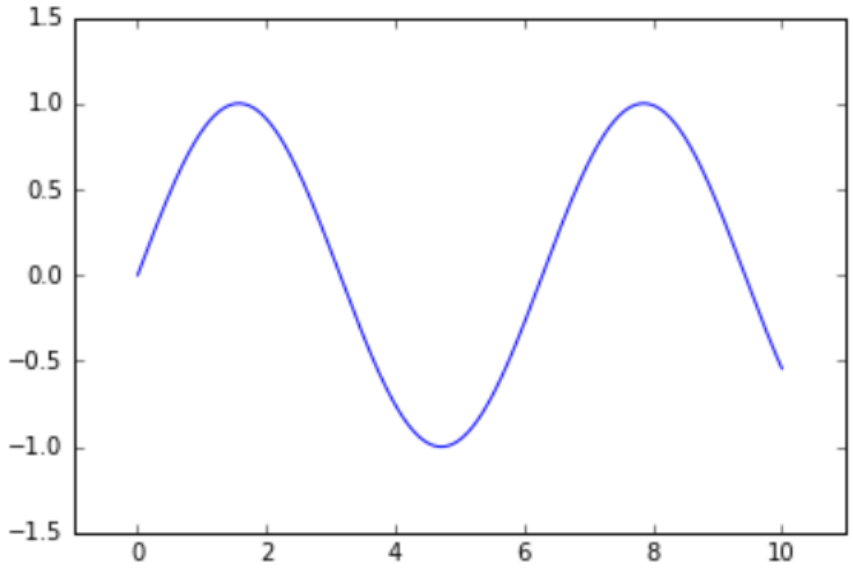
```
(1.2, -1.2)
```



Axes Limits

- A useful related method is **plt.axis()** (note here the potential confusion between *axes* with an *e*, and *axis* with an *i*)
- The plt.axis() method allows you to set the *x* and *y* limits with a single call, by passing a list which specifies [xmin, xmax, ymin, ymax]:

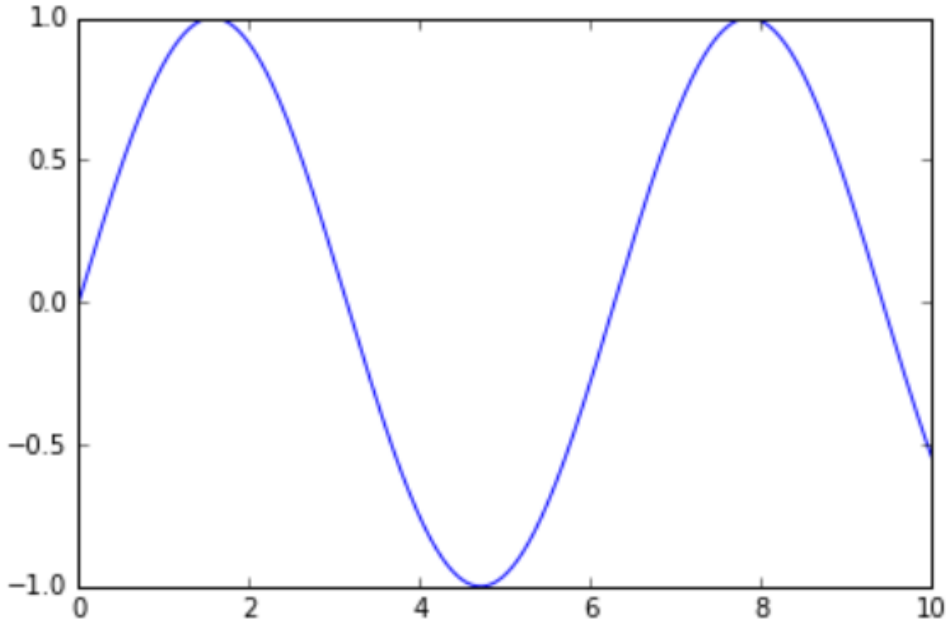
```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```



Axes Limits

- The plt.axis() method goes even beyond this, allowing you to do things like automatically tighten the bounds around the current plot:

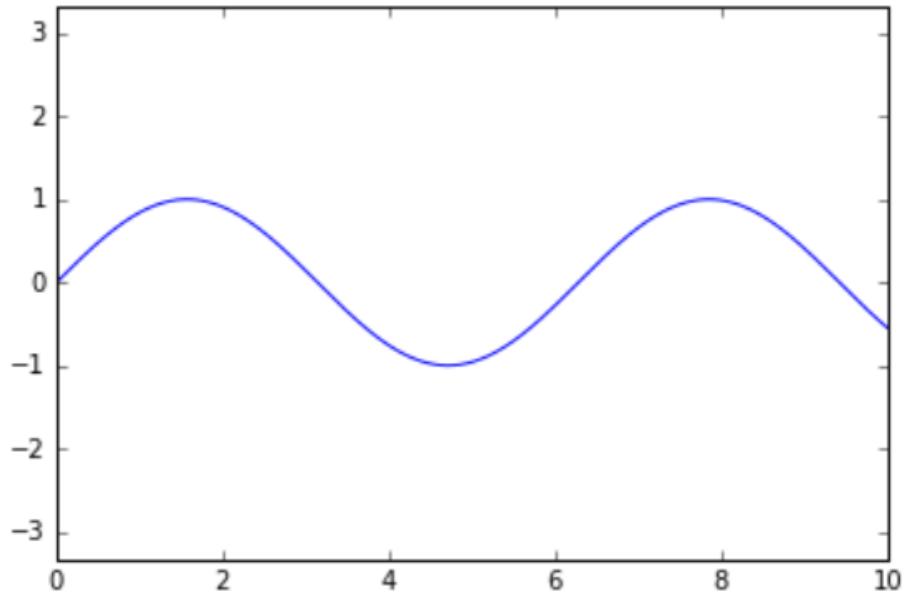
```
plt.plot(x, np.sin(x))
plt.axis('tight');
```



Axes Limits

- It allows even higher-level specifications, such as ensuring an equal aspect ratio so that on your screen, one unit in x is equal to one unit in y:

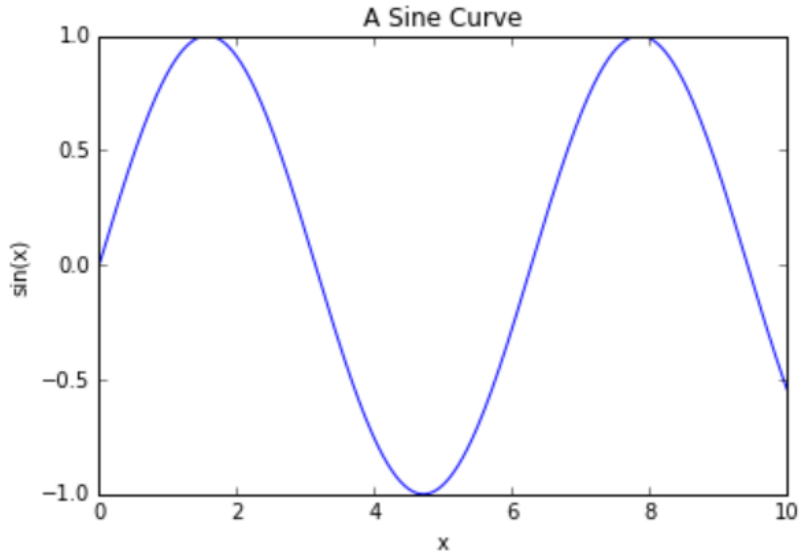
```
plt.plot(x, np.sin(x))
plt.axis('equal');
```



Labeling Plots

- A plot can be given a title above the axes by passing a string to the **title()** function
- Similarly, the methods **xlabel()** and **ylabel()** control the labeling of the *x*- and *y*-axes
- The position, size, and style of these labels can be adjusted using optional arguments

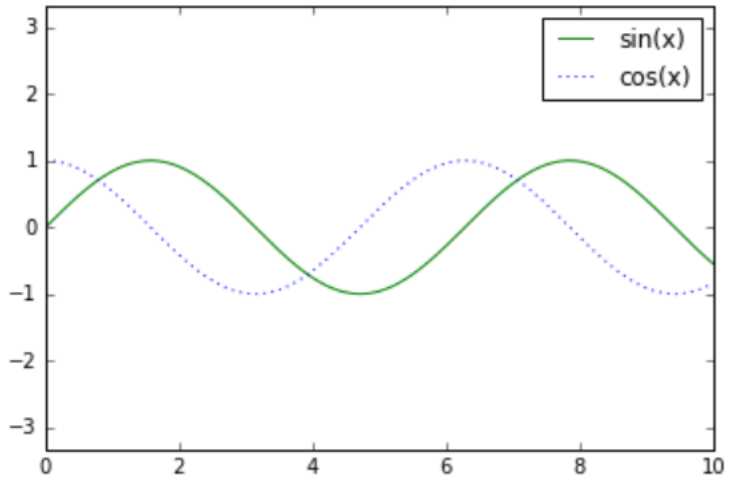
```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```



Legend

- When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type
- Each line on a plot can be given a label by passing a string to its **label** argument
- The labels appear on the plot when you call the function **plt.legend()**:

```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.axis('equal')

plt.legend();
```

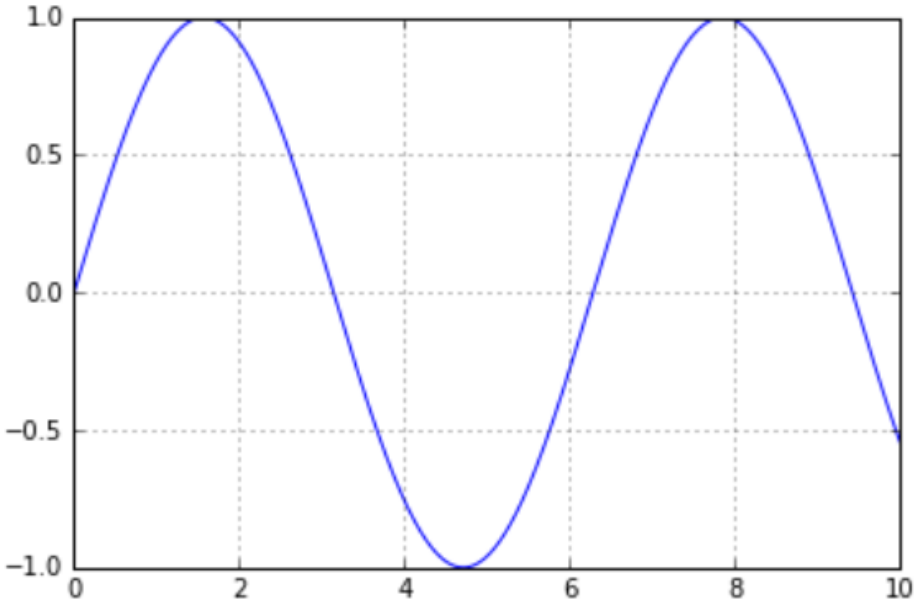


Gridlines

- Gridlines are vertical (for the *x*-axis) and horizontal (for the *y*-axis) lines running across the plot to aid with locating the numerical values of data points
- By default no gridlines are drawn, but they may be turned on by calling **plt.grid()**
- You can set the *axis* argument to ‘x’, ‘y’, or ‘both’ (default) to control which set of gridlines are drawn
- The line properties of the gridlines are set with the linestyle, linewidth, color, etc. arguments, as for plot lines

Gridlines

```
plt.plot(x, np.sin(x))
plt.grid()
```



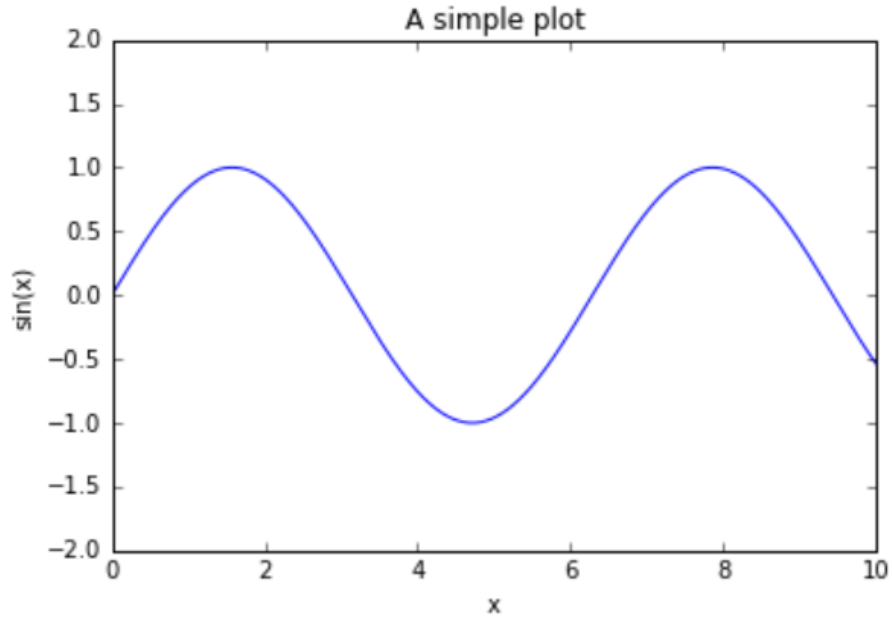
Matplotlib Two Interfaces

- While most plt functions translate directly to ax methods (e.g., plt.plot() → ax.plot(), plt.legend() → ax.legend()), this is not the case for all commands
- In particular, functions to set limits, labels, and titles are slightly modified
- For transitioning between MATLAB-style functions and object-oriented methods, make the following changes:
  - plt.xlabel() → ax.set_xlabel()
  - plt.ylabel() → ax.set_ylabel()
  - plt.xlim() → ax.set_xlim()
  - plt.ylim() → ax.set_ylim()
  - plt.title() → ax.set_title()

Matplotlib Two Interfaces

- In the OO interface to plotting, rather than calling these functions individually, it is often more convenient to use the **ax.set()** method to set all these properties at once:

```
ax = plt.axes()
ax.plot(x, np.sin(x))
ax.set(xlim=(0, 10), ylim=(-2, 2),
       xlabel='x', ylabel='sin(x)',
       title='A simple plot');
```

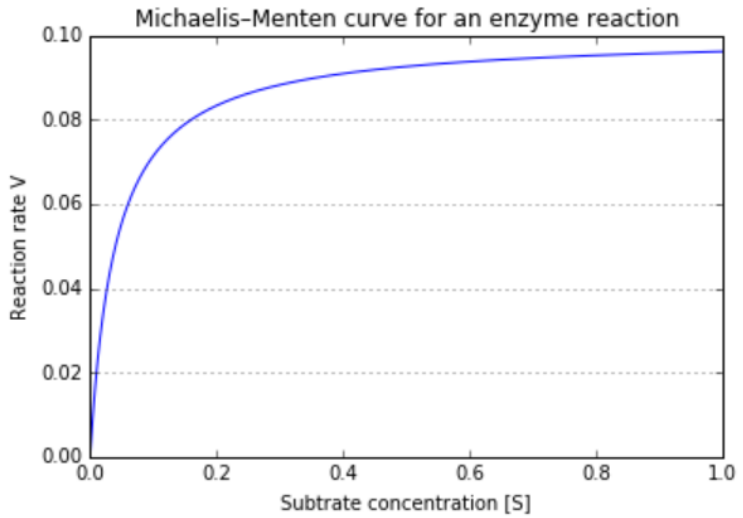


Exercise

- The Michaelis-Menten equation models the kinetics of enzymatic reactions as

$$v = \frac{d[P]}{dt} = \frac{V_{\max}[S]}{K_m + [S]}$$

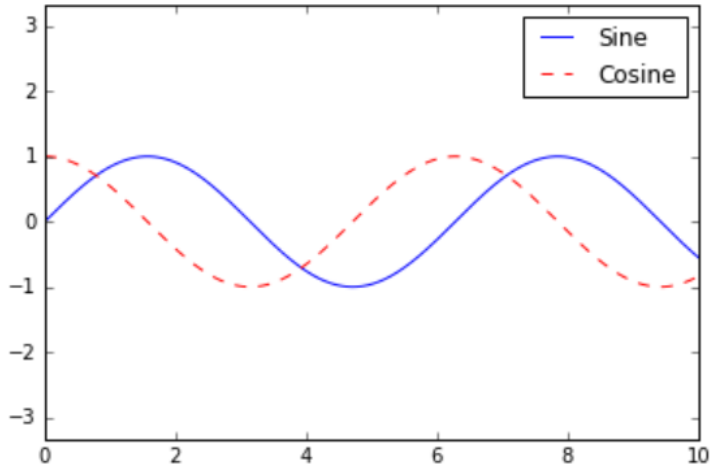
  - v is the rate of the reaction converting the substrate, S, to product P, catalyzed by the enzyme
  - V_{max} is the maximum rate (when all the enzyme is bound to S)
  - K_m is the substrate concentration at which the reaction rate is at half its maximum value
- Plot v against [S] for a reaction with K_m = 0.04 M and V_{max} = 0.1 Ms^{-1}



Customizing Plot Legends

- Legends give meaning to the various plot elements
- The simplest legend can be created with the **plt.legend()** command, which automatically creates a legend for any labeled plot elements:

```
x = np.linspace(0, 10, 1000)
fig = plt.figure()
ax = plt.axes()

ax.plot(x, np.sin(x), '-b', label='Sine')
ax.plot(x, np.cos(x), '--r', label='Cosine')
ax.axis('equal')
ax.legend();
```

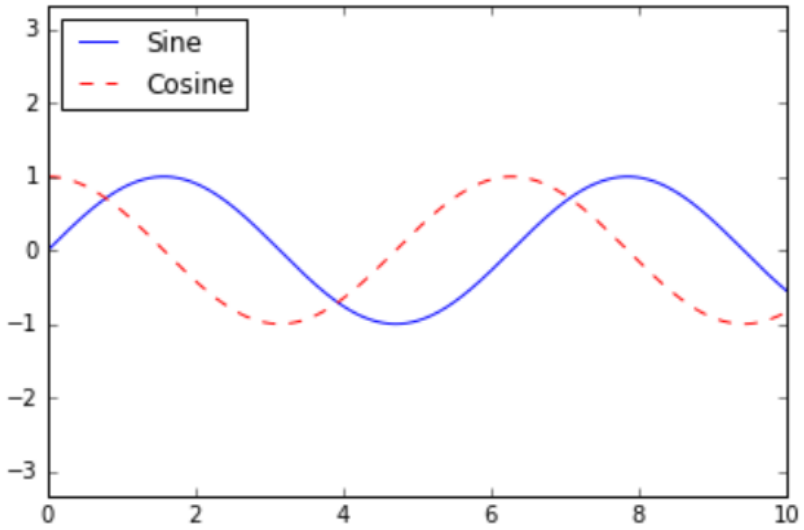


Legend Location

- The location of the legend is, by default, the top right-hand corner of the plot
- It can be customized by setting the **loc** argument to the legend method to either the string or integer values given in the following table:

String	Integer
best	0
upper right	1
upper left	2
lower left	3
lower right	4
right	5
center left	6
center right	7
lower center	8
upper center	9
center	10

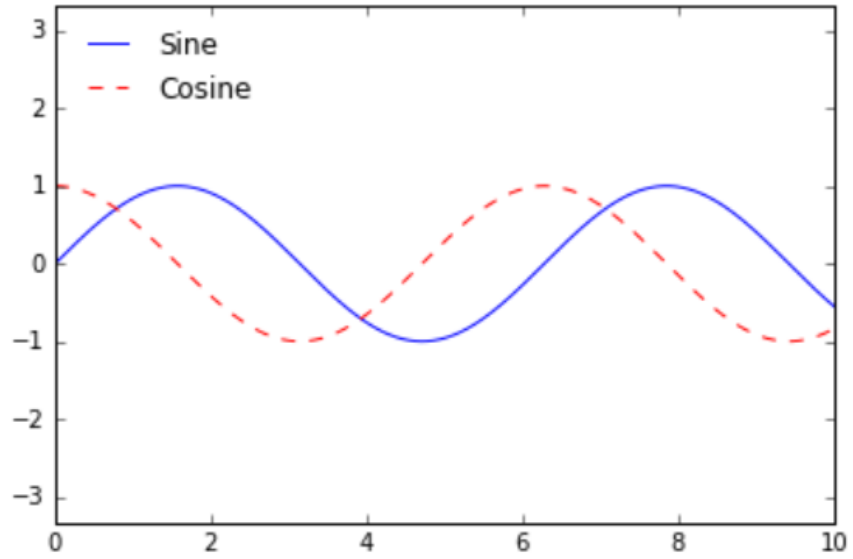
```
ax.legend(loc='upper left')
fig
```



Changing the Frame

- You can turn off the frame:

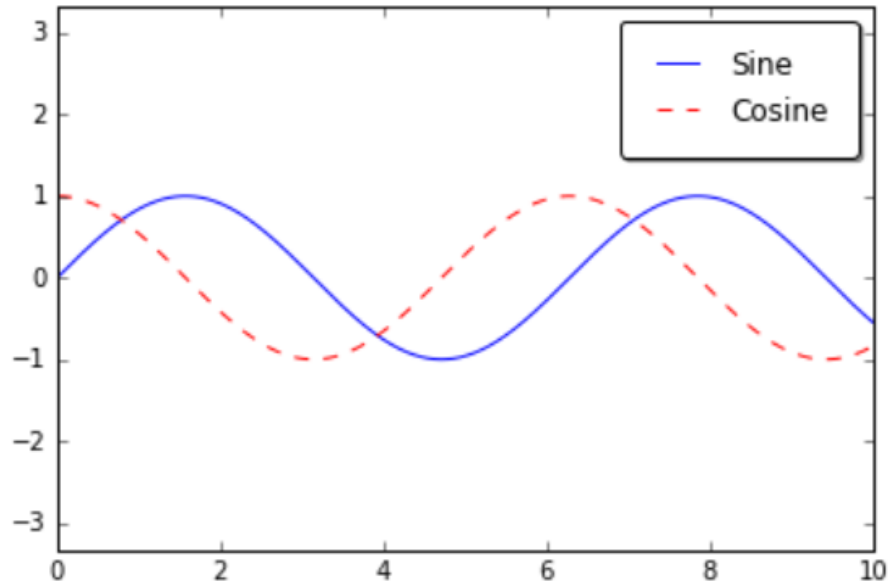
```
ax.legend(frameon=False, loc='upper left')
fig
```



Changing the Frame

- We can use a rounded box (fancybox) or add a shadow, change the transparency (alpha value) of the frame, or change the padding around the text:

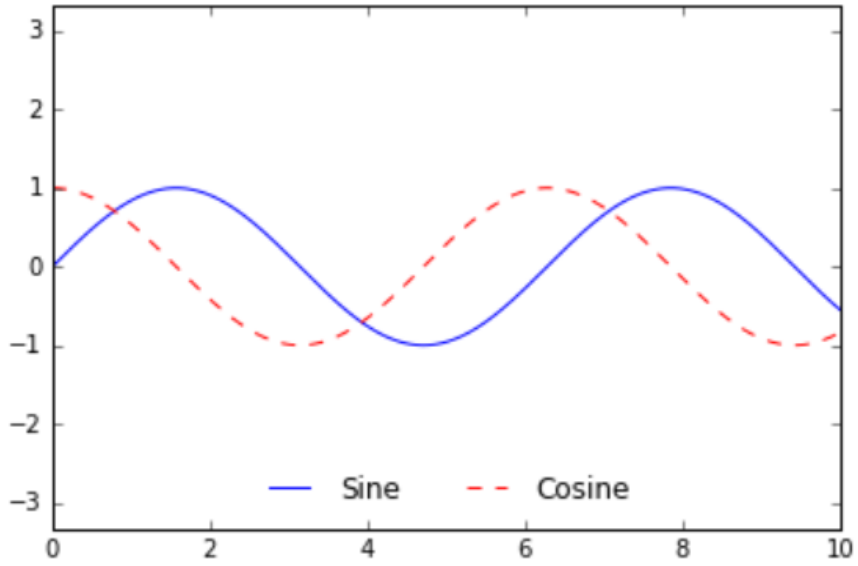
```
ax.legend(fancybox=True, framealpha=1, shadow=True, borderpad=1)
fig
```



Number of Columns

- We can use the **ncol** argument to specify the number of columns in the legend:

```
ax.legend(frameon=False, loc='lower center', ncol=2)
fig
```

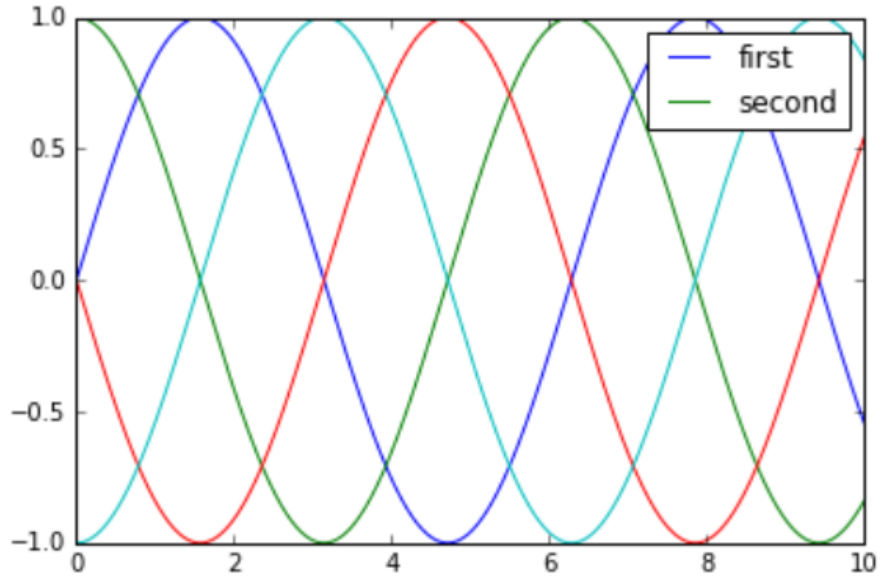


Choosing Elements for the Legend

- The legend includes all labeled elements by default
- If this is not what is desired, we can fine-tune which elements and labels appear in the legend by using the objects returned by plot commands
- The plt.plot() command is able to create multiple lines at once, and returns a list of created line instances
- Passing any of these to plt.legend() will tell it which to identify, along with the labels we'd like to specify

Choosing Elements for the Legend

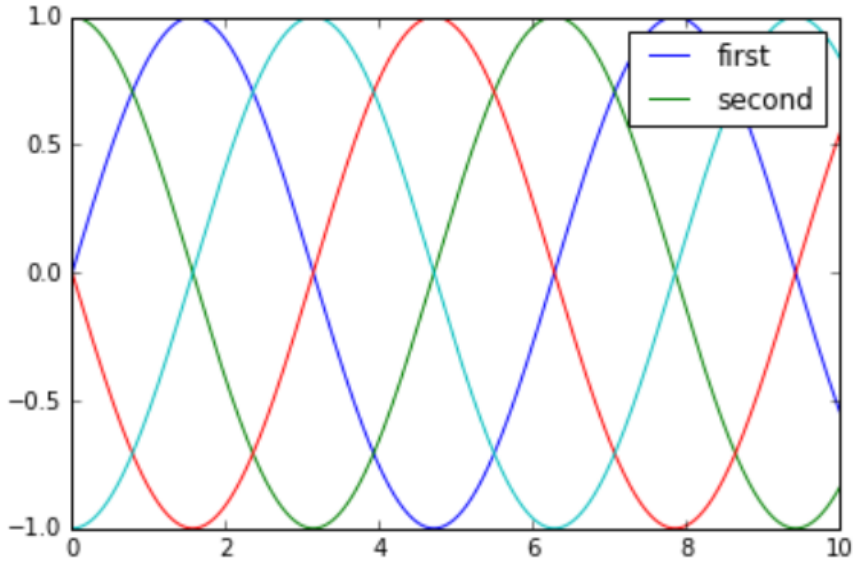
```
y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
lines = plt.plot(x, y)

# lines is a list of plt.Line2D instances
plt.legend(lines[:2], ['first', 'second']);
```



Choosing Elements for the Legend

- The same effect can be achieved by applying labels to the plot elements you'd like to show on the legend:

```
plt.plot(x, y[:, 0], label='first')
plt.plot(x, y[:, 1], label='second')
plt.plot(x, y[:, 2:])
plt.legend();
```



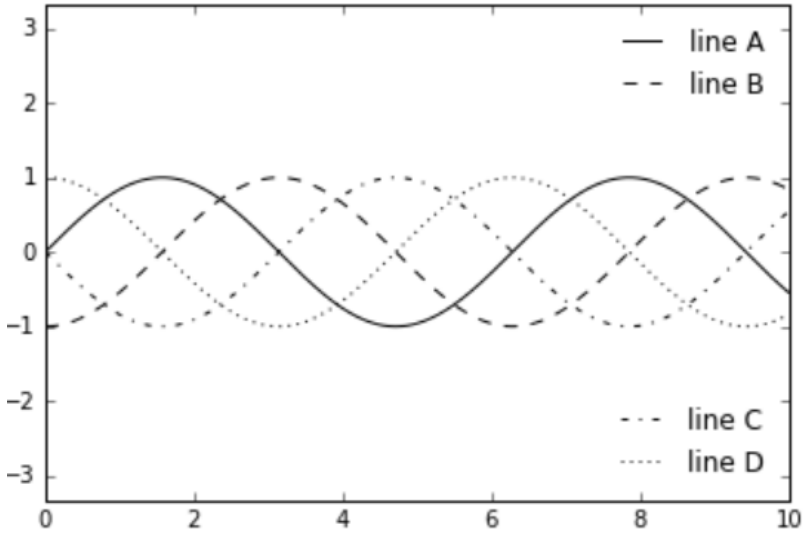
Multiple Legends

Experis KickStart ManpowerGroup

- Sometimes when designing a plot you'd like to add multiple legends to the same axes
- Unfortunately, Matplotlib does not make this easy: via the standard legend() interface, it is only possible to create a single legend for the entire plot
- If you try to create a second legend using plt.legend() or ax.legend(), it will simply override the first one
- We can work around this by creating a new legend artist from scratch, and then using the lower-level ax.add_artist() method to manually add the second artist to the plot

Multiple Legends

```
fig = plt.figure()
ax = plt.axes()

lines = []
styles = ['-', '--', '-.', ':']
x = np.linspace(0, 10, 1000)

for i in range(4):
    lines += ax.plot(x, np.sin(x - i * np.pi / 2),
                     styles[i], color='black')
ax.axis('equal')

# specify the lines and labels of the first legend
ax.legend(lines[:2], ['line A', 'line B'],
          loc='upper right', frameon=False)

# Create the second legend and add the artist manually.
from matplotlib.legend import Legend
leg = Legend(ax, lines[2:], ['line C', 'line D'],
             loc='lower right', frameon=False)
ax.add_artist(leg);
```



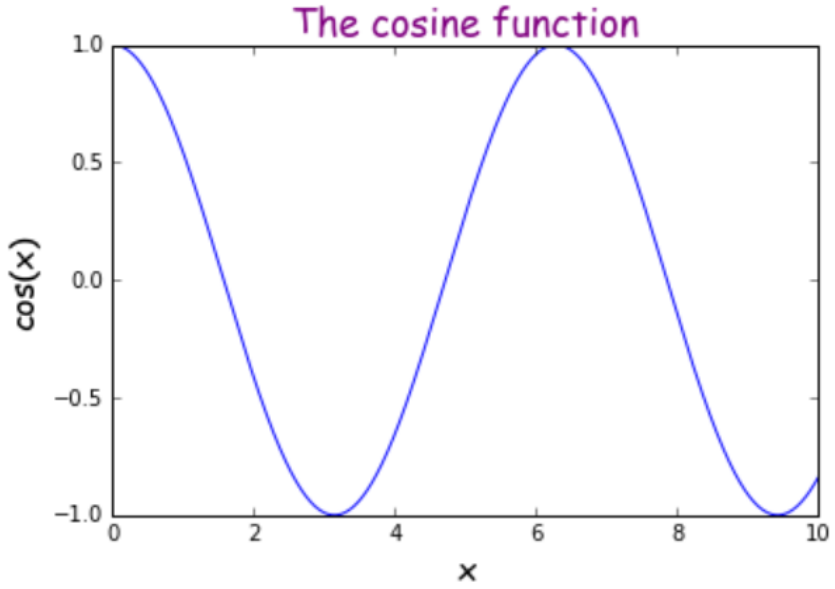
Font Properties

- The text elements of a plot (titles, legend, axis labels, etc.) can be customized with the arguments given in the following table:

Argument	Description
fontsize	The size of the font in points (e.g., 12, 16)
fontname	The font name (e.g., 'Courier', 'Arial')
family	The font family (e.g., 'sans-serif', 'cursive', 'monospace')
fontweight	The font weight (e.g., 'normal', 'bold')
fontstyle	The font style (e.g., 'normal', 'italic')
color	Any Matplotlib color specifier (e.g., 'r', '#ff00ff')

Font Properties

```
x = np.linspace(0, 10, 1000)
f = np.cos(x)

plt.plot(x, f)
plt.title('The cosine function', fontsize=18,
          fontname='Comic Sans MS', color='purple')

plt.xlabel('x', fontsize=16, fontname='Comic Sans MS')
plt.ylabel('cos(x)', fontsize=16, fontname='Comic Sans MS');
```



Scatter Plots

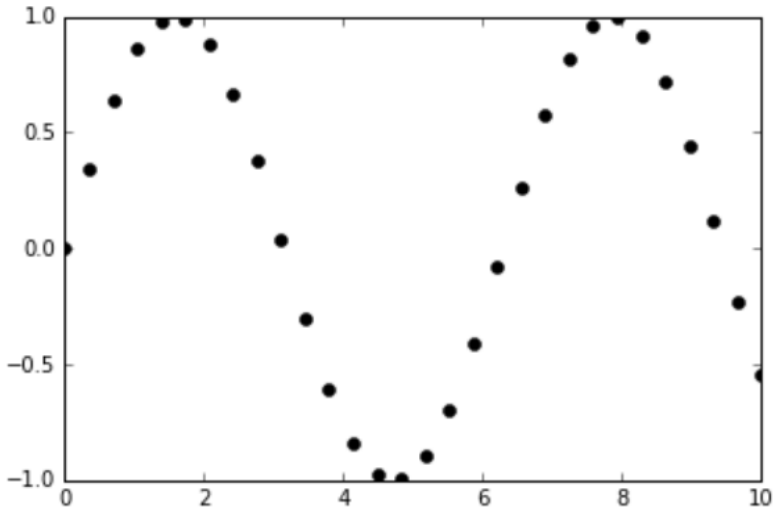
Experis KickStart ManpowerGroup

- Another commonly used plot type is the scatter plot, a close cousin of the line plot
- Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape
- You can create scatter plots in Matplotlib by using one of the following methods:
  - The familiar plt.plot()/ax.plot() can produce scatter plots as well
  - A second, more powerful method of creating scatter plots is plt.scatter()

Scatter Plots with plt.plot()

- The third argument in plt.plot() is a character that represents the type of symbol used for the plotting
- Just as you can specify options such as '-', '--' to control the line style, you can also use it to set a marker on each point of the plotted data

```
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, 'o', color='black');
```



Markers

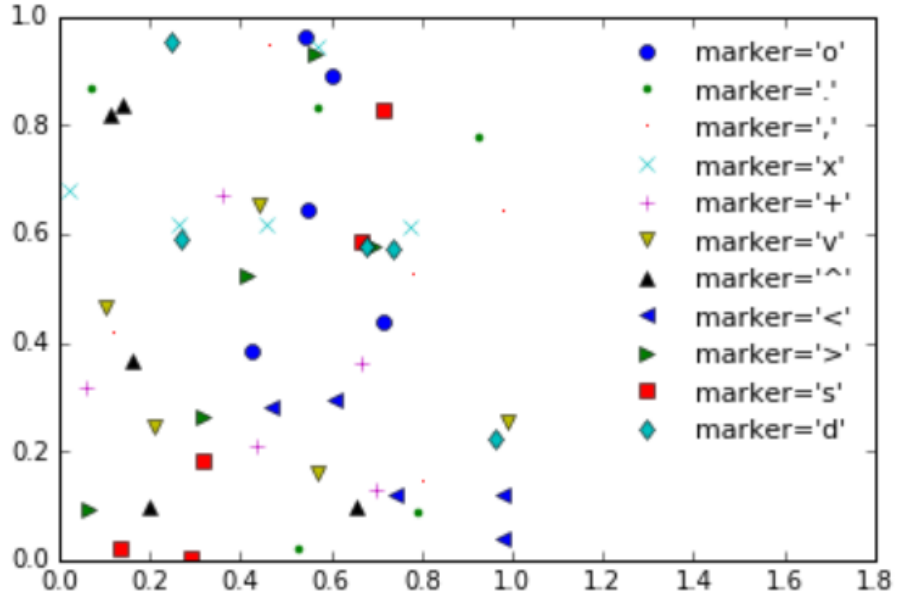
- The more useful markers are listed in the following table:

Code	Marker	Description
.	·	point
o	o	circle
+	+	plus
x	×	x
D	◇	diamond
v	▽	downward triangle
^	△	upward triangle
s	□	square
*	★	star

- The full list of markers can be found at https://matplotlib.org/api/markers_api.html

Markers

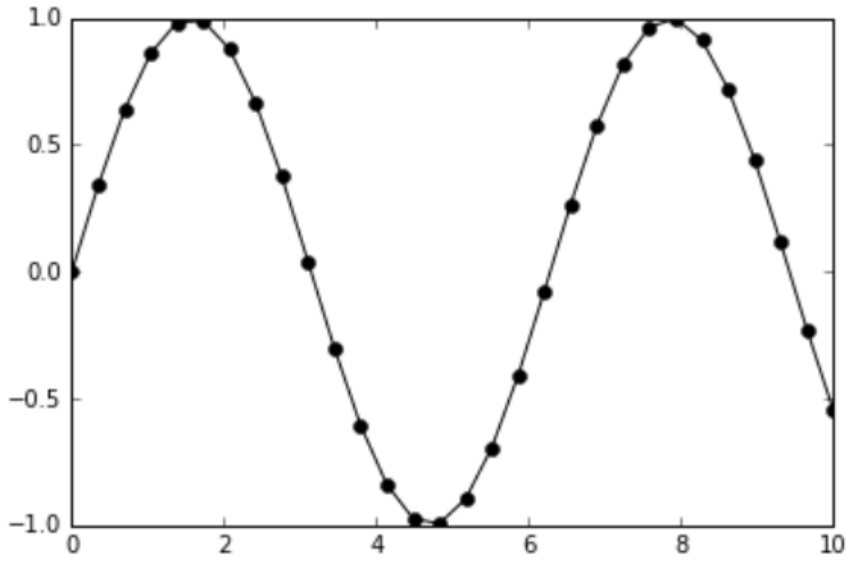
```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker, label=f"marker='{marker}'")
plt.legend(numpoints=1, frameon=False, fontsize=11)
plt.xlim(0, 1.8);
```



Markers

- For even more possibilities, the marker codes can be used together with line and color codes to plot points along with a line connecting them:

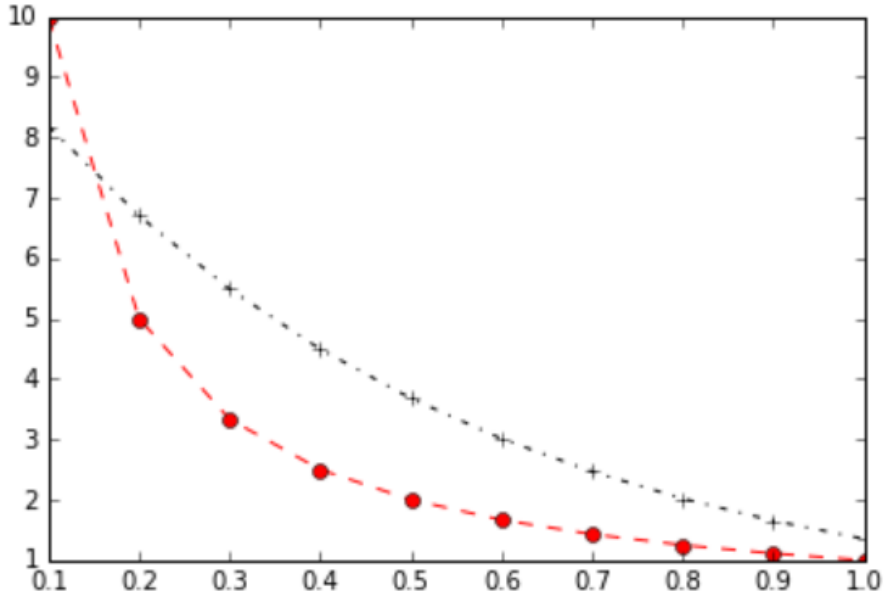
```
plt.plot(x, y, '-ok');  # a solid, black line with circle markers
```



Markers

- Multiple lines can be plotted using a sequence of x, y, format arguments:

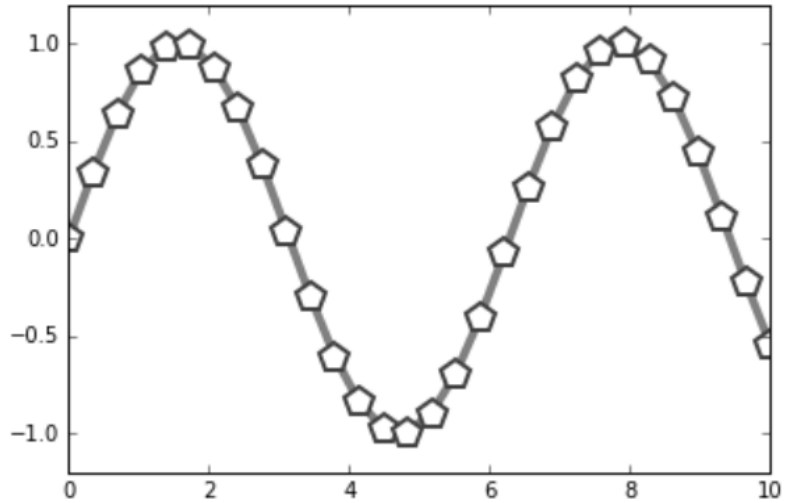
```
x = np.linspace(0.1, 1, 10)
y1 = 1 / x
y2 = 10 * np.exp(-2 * x)

plt.plot(x, y1, 'r--o', x, y2, 'k-.+');
```



Markers

- Additional keyword arguments to plt.plot() specify a wide range of properties of the lines and markers:

```
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, '-p', color='gray', linewidth=4,
         markersize=15,
         markerfacecolor='white',
         markeredgecolor='0.25',
         markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```



Exercise

- *Moore's Law* is the observation that the number of transistors on CPUs approximately doubles every two years
- Write a program that illustrates Moore's Law by comparing between the actual number of transistors on high-end CPUs from between 1972 and 2012, and that predicted by Moore's Law which may be stated mathematically as:

$$n_i = n_0 2^{(y_i - y_0)/2}$$

  - where n_0 is the number of transistors in some reference year, y_0
- Because the data cover 40 years, the values of n_i span many orders of magnitude, and it is convenient to apply Moore's Law to its logarithm, which shows a linear dependence on y:

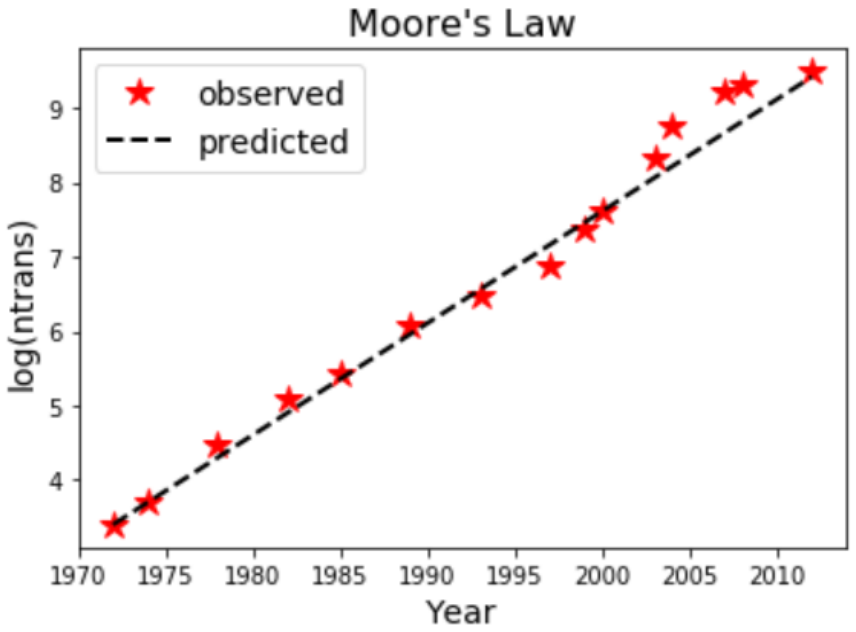
$$\log_{10} n_i = \log_{10} n_0 + \frac{y_i - y_0}{2} \log_{10} 2$$

Exercise

- Use the following data to create a graph showing the observed vs. the predicted number of CPU transistors for each year in logarithmic scale

Year	CPU transistors (in millions)
1972	0.0025
1974	0.005
1978	0.029
1982	0.12
1985	0.275
1989	1.18
1993	3.1
1997	7.5
1999	24.0
2000	42.0

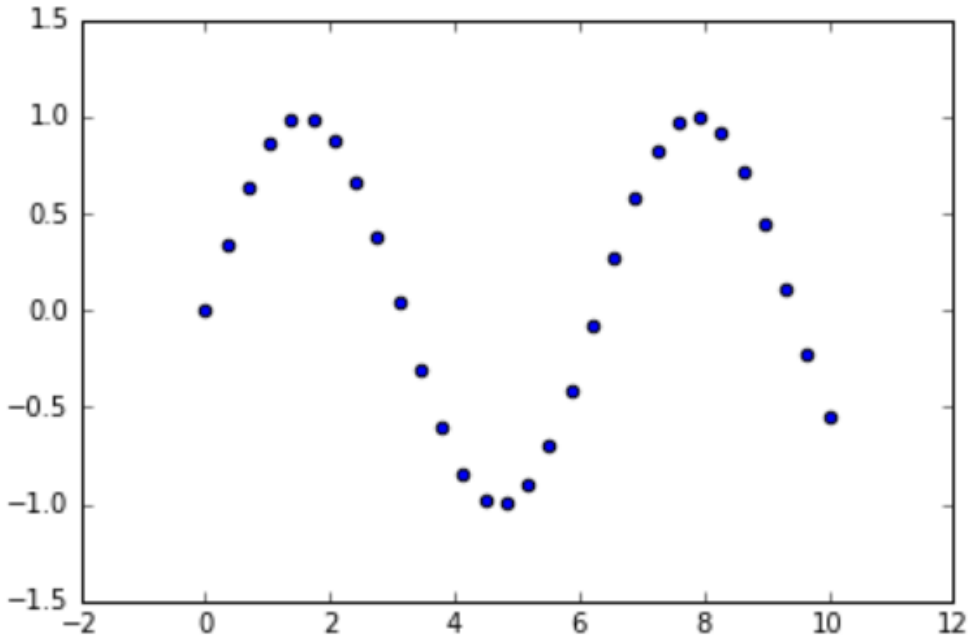
Year	CPU transistors (in millions)
2003	220.0
2004	592.0
2007	1720.0
2008	2046.0
2012	3100.0



Scatter Plots with plt.scatter()

- A second, more powerful method of creating scatter plots is the **plt.scatter()** function, which can be used very similarly to the plt.plot() function:

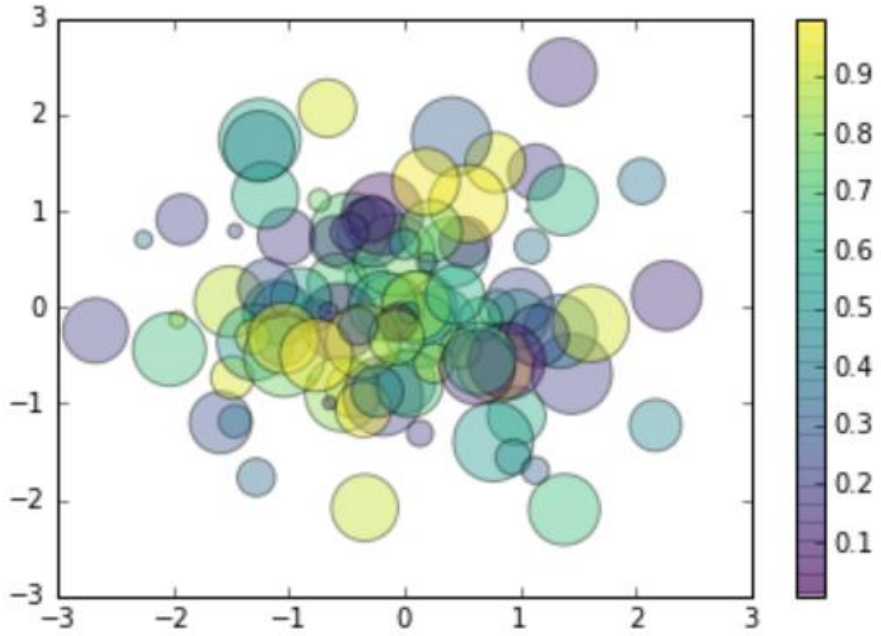
```
plt.scatter(x, y, marker='o');
```



Scatter Plots with plt.scatter()

- The primary difference of plt.scatter() from plt.plot() is that it can be used to control or map to data the properties of each individual point (size, color, etc.)
- Let's show this by creating a random scatter plot with points of many colors and sizes

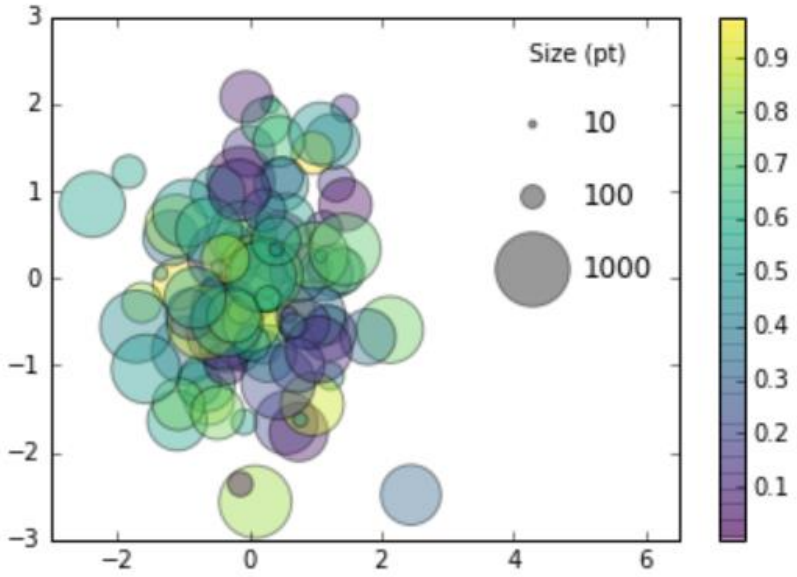
```
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100) # in points (100pt~3.5cm)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.4,
            cmap='viridis')
plt.colorbar();   # show color scale
```



Scatter Plots with plt.scatter()

- We can create a legend that specifies the scale of the sizes of the points, by plotting some labeled data with no entries:

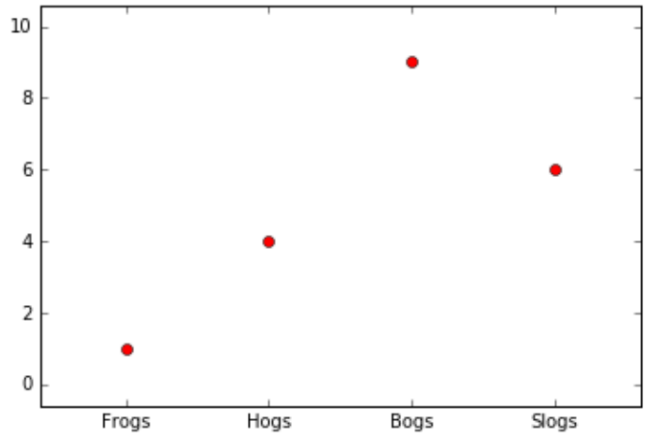
```
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100) # in points

plt.scatter(x, y, c=colors, s=sizes, alpha=0.4,
            cmap='viridis')
plt.colorbar();   # show color scale

# Here we create a legend:
# we'll plot empty lists with the desired size and label
for size in [10, 100, 1000]:
    plt.scatter([], [], c='k', alpha=0.4, s=size,
                label=str(size))

plt.legend(scatterpoints=1, frameon=False, labelspacing=1.5,
           title='Size (pt)')
plt.xlim(-3, 6.5);
```



Customizing Ticks

- Matplotlib does its best to label representative values (*tick marks)* on each axis
- However, there are some occasions when you want to customize them, for example, to make the tick marks more or less frequent, or to label them differently
- You can change the tick marks by calling the methods **plt.xticks**(*locs*, *labels*) and **plt.yticks**(*locs*, *labels*), supplying a list of positions at which ticks should be placed, and optionally a list of explicit labels to place at those locations
- To remove the tick marks altogether, set them to the empty list, e.g., yticks([])

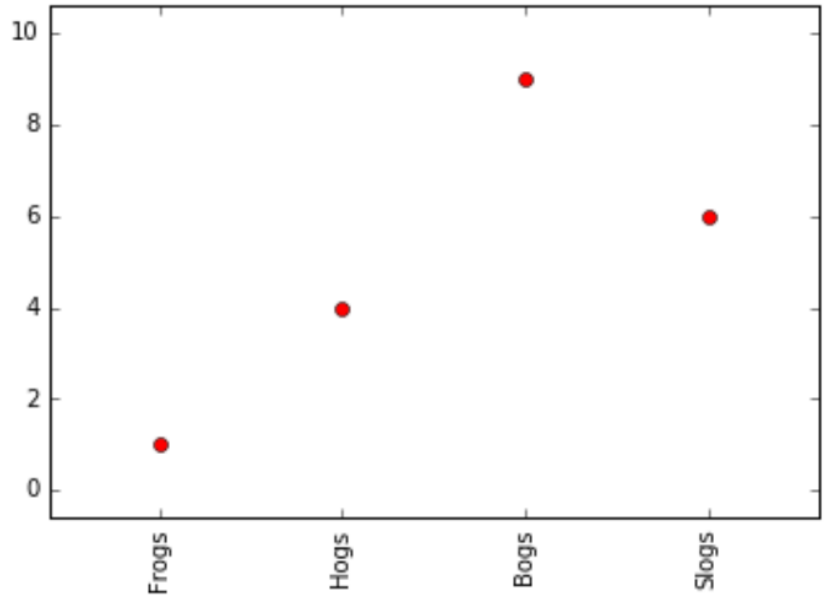
```
x = [1, 2, 3, 4]
y = [1, 4, 9, 6]
labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']

plt.plot(x, y, 'ro')
plt.xticks(x, labels)

# Pad margins so that markers don't get clipped by the axes
plt.margins(0.2);
```



Rotating the Tick Marks

- You can specify a rotation for the tick labels using the **rotation** keyword
  - The rotation can be specified in degrees or with keywords

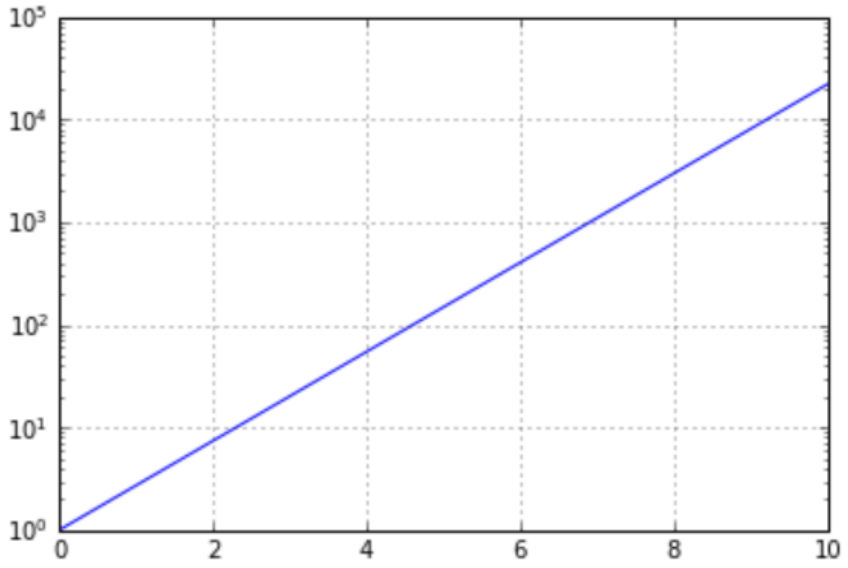
```
x = [1, 2, 3, 4]
y = [1, 4, 9, 6]
labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']

plt.plot(x, y, 'ro')
plt.xticks(x, labels, rotation='vertical')

# Pad margins so that markers don't get clipped by the axes
plt.margins(0.2);
```



Major and Minor Ticks

- Within each axis, there is the concept of a *major* tick mark, and a *minor* tick mark
- Each major tick shows a large tickmark and a label, while each minor tick shows a smaller tickmark with no label
- By default, Matplotlib rarely makes use of minor ticks, but one place you can see them is within logarithmic plots:

```
fig = plt.figure()
x = np.linspace(0, 10, 1000)
y = np.e ** x

ax = plt.axes(yscale='log')
ax.plot(x, y)
ax.grid()
```



Multiple Subplots

Experis KickStart ManpowerGroup

- Sometimes it is helpful to compare different views of data side by side
- To this end, Matplotlib has the concept of *subplots*: groups of smaller axes that can exist together within a single figure
- These subplots might be insets, grids of plots, or other more complicated layouts
- We'll explore four routines for creating subplots in Matplotlib

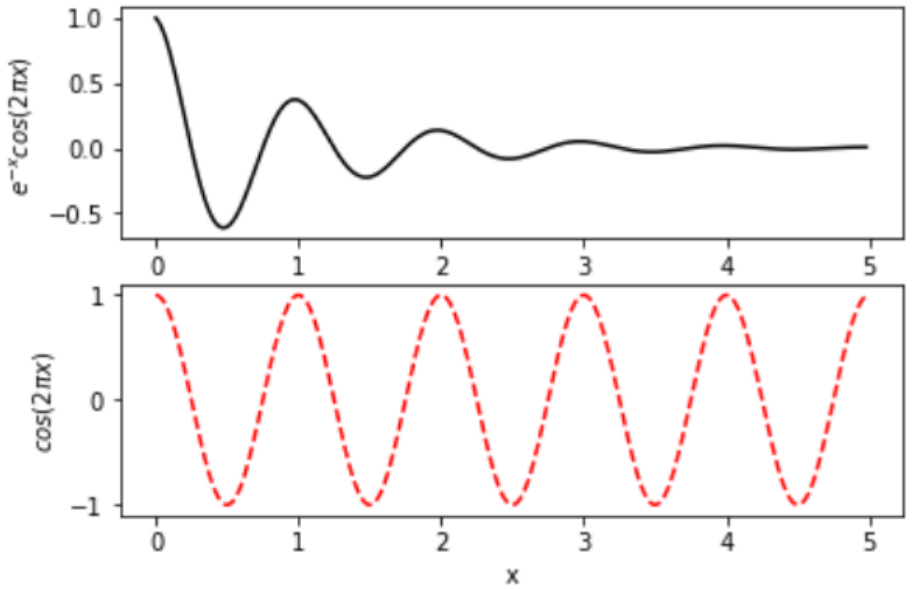
Multiple Subplots

Experis KickStart ManpowerGroup

- The **subplot(numrows, numscols, fignum)** function specifies the number of rows, number of columns and the subplot number
  - fignum ranges from 1 to numrows * numcols
- The subplot number increases along the columns in each row and then down the rows
- The commas in the subplot command are optional if numrows * numcols < 10
  - e.g., subplot(211) is idential to subplot(2, 1, 1)
- On the next slide there is a script to create two subplots

Multiple Subplots

```
x = np.arange(0.0, 5.0, 0.02)
y1 = np.exp(-x) * np.cos(2 * np.pi * x)
y2 = np.cos(2 * np.pi * x)

plt.subplot(211)
plt.plot(x, y1, 'k')
plt.ylabel('$e^{-x} cos(2 \pi x)$')

plt.subplot(212)
plt.plot(x, y2, 'r--')
plt.xlabel('x')
plt.ylabel('$cos(2 \pi x)$')

plt.show()
```



The Whole Grid in One Go

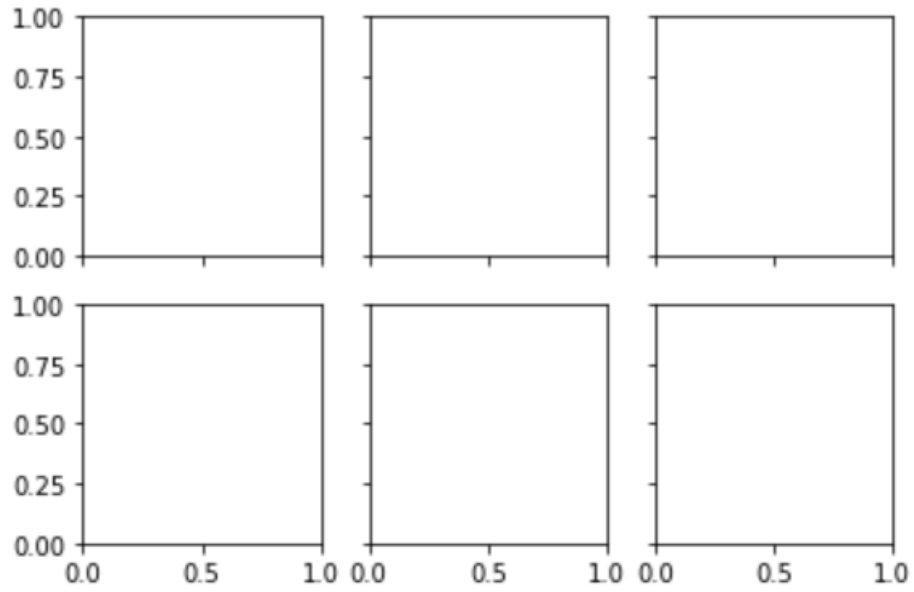
Experis KickStart ManpowerGroup

- The approach just described can become quite tedious when creating a large grid of subplots, especially if you'd like to hide the x- and y-axis labels on the inner plots
- The function **plt.subplots()** (note the s at the end of subplots) creates a full grid of subplots in a single line, returning them in a NumPy array
- The arguments are the number of rows and number of columns, along with optional keywords **sharex** and **sharey**, which allow you to specify the relationships between different axes

The Whole Grid in One Go

- Here we'll create a 2×3 grid of subplots, where all axes in the same row share their y-axis scale, and all axes in the same column share their x-axis scale:

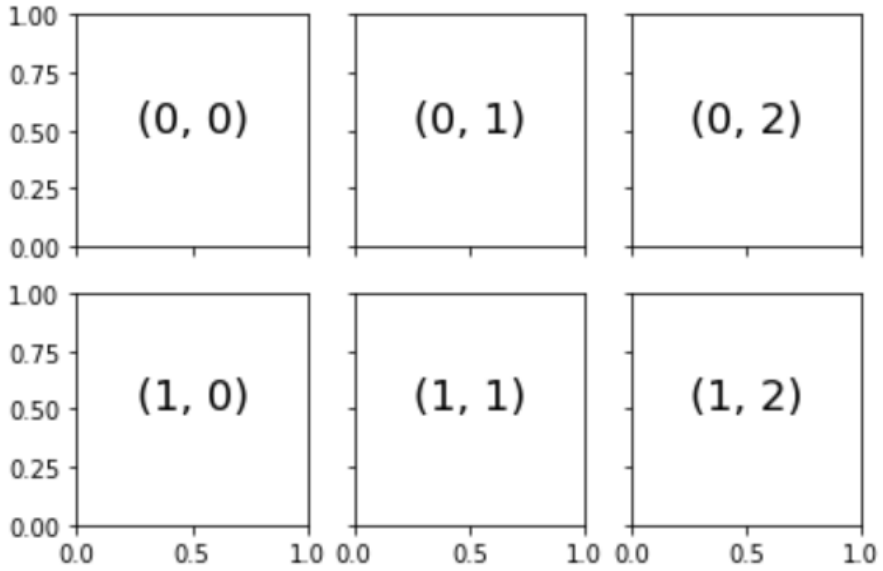
```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```



The Whole Grid in One Go

- We can now use a standard array indexing notation to perform an operation on all the subplots:

```
# axes are in a 2D array, indexed by [row, col]
for i in range(2):
    for j in range(3):
        ax[i, j].text(0.5, 0.5, str((i, j)),
                      fontsize=18, ha='center')
fig
```



More Complex Arrangements

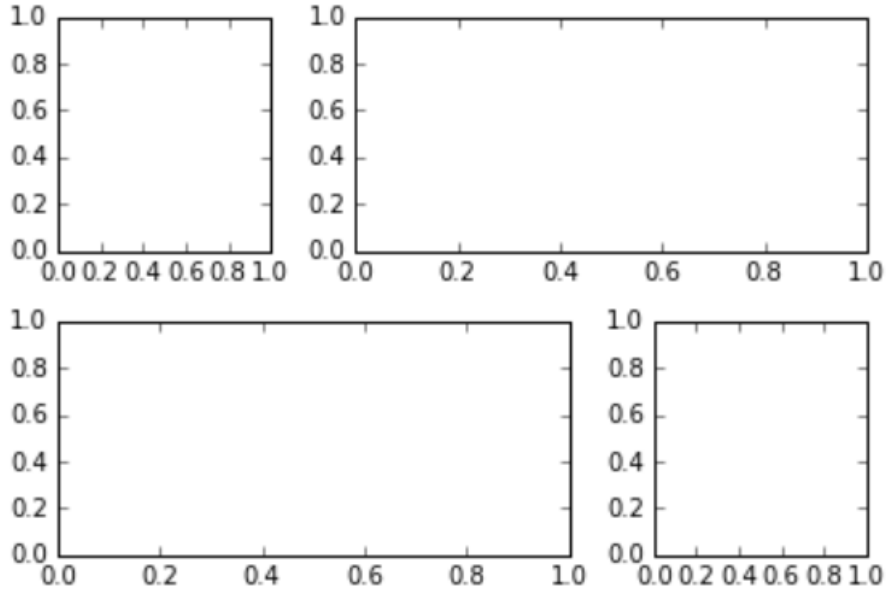
- To go beyond a regular grid to subplots that span multiple rows and columns, **plt.GridSpec()** is the best tool
- The plt.GridSpec() object does not create a plot by itself
- It is simply a convenient interface that is recognized by the plt.subplot() command
- For example, a gridspec for a grid of two rows and three columns with some specified width and height space looks like this:

```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
```

More Complex Arrangements

- From this we can specify subplot locations and extents using the familiar Python slicing syntax:

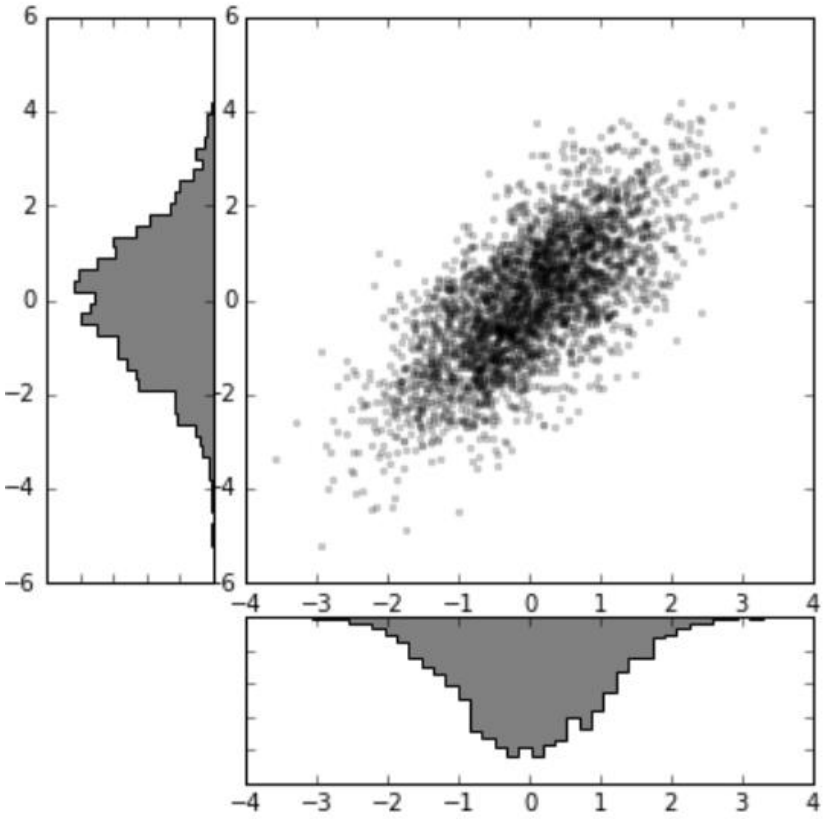
```
plt.subplot(grid[0, 0])
plt.subplot(grid[0, 1:])
plt.subplot(grid[1, :2])
plt.subplot(grid[1, 2]);
```



More Complex Arrangements

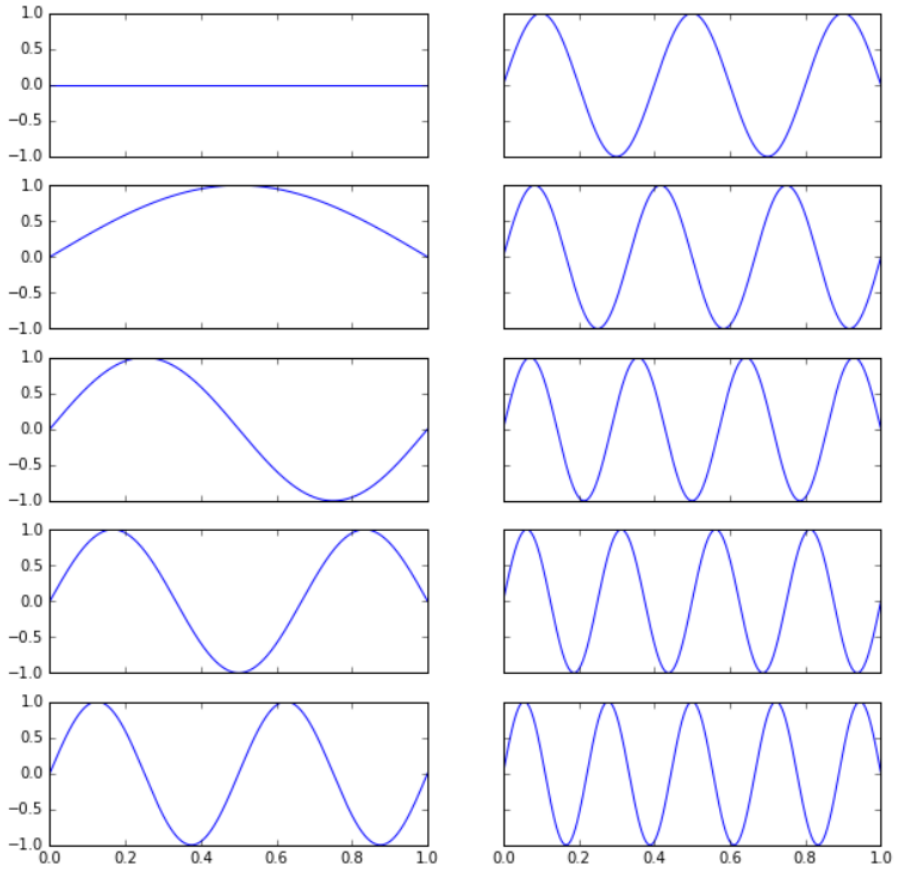
- This type of arrangement is useful for creating multi-axes histogram plots like this one:

```
# Create some normally distributed data
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 3000).T

# Set up the axes with gridspec
fig = plt.figure(figsize=(6, 6))
grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)
main_ax = fig.add_subplot(grid[:-1, 1:])
y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=main_ax)
x_hist = fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=main_ax)

# scatter points on the main axes
main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)

# histogram on the attached axes
x_hist.hist(x, 40, histtype='stepfilled', orientation='vertical',
            color='gray')
x_hist.invert_yaxis()

y_hist.hist(y, 40, histtype='stepfilled', orientation='horizontal',
            color='gray')
y_hist.invert_xaxis()
```



Exercise

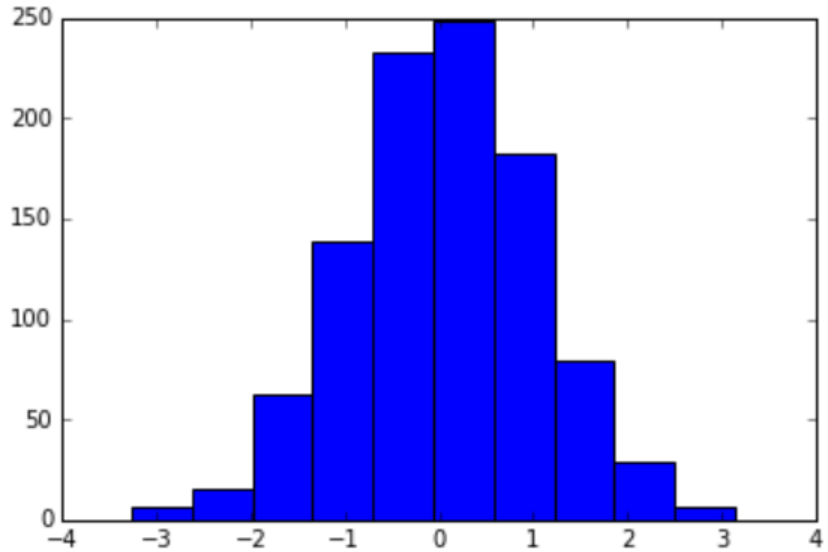
- Generate a figure of 10 subplots depicting the graph of $\sin(n\pi x)$ for $n = 1, 2, \ldots, 10$, and $x \in [0, 1]$. Display them using 5 rows and 2 columns:



Histograms

- A simple histogram can be a great first step in understanding a dataset
- A histogram displays the distribution of data as a series of vertical bars with lengths in proportion to the number of data items falling into predefined ranges (bins)
- **plt.hist()** produces a histogram from a sequence of data values

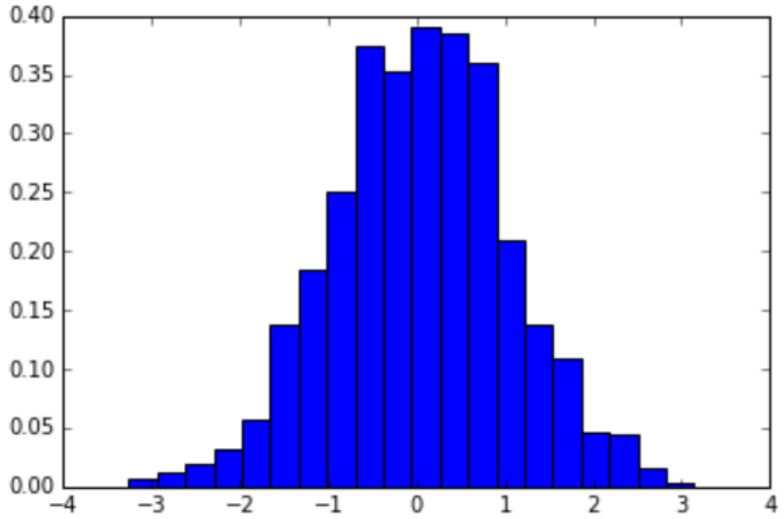
```
data = np.random.randn(1000)
plt.hist(data);
```



Histograms

- The number of bins can be passed as an optional argument **bins** (default is 10)
- Also by default the height of the histogram bars are absolute counts of the data in the corresponding bin
- Setting the attribute **normed=True** normalizes the histogram so that the total areas of the bins equals to 1

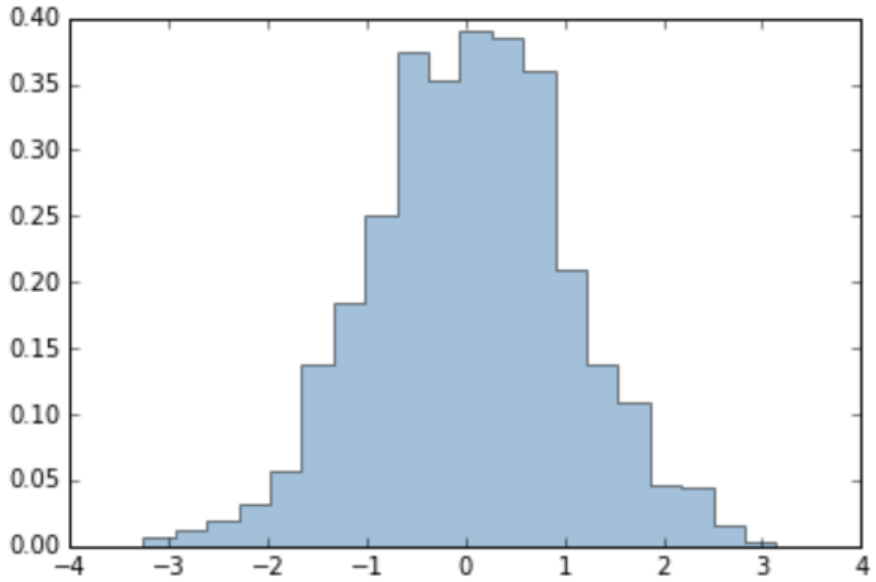
```
plt.hist(data, bins=20, normed=True);
```



Histograms

- The hist() function has many options to tune both the calculation and the display
- Here's an example of a more customized histogram:

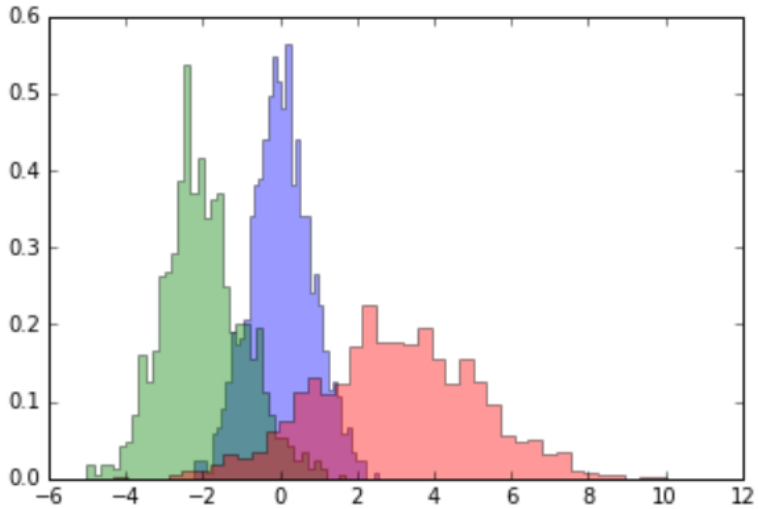
```
plt.hist(data, bins=20, normed=True, alpha=0.5,
         histtype='stepfilled', color='steelblue');
```



Histograms

- The combination of histtype='stepfilled' along with some transparency alpha can be very useful when comparing histograms of several distributions:

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

kwargs = dict(histtype='stepfilled', alpha=0.4, normed=True, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```



Histograms

- If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the **np.histogram()** function is available:

```
counts, bin_edges = np.histogram(data, bins=5)
print(counts)
```

```
[ 22 201 481 261  35]
```

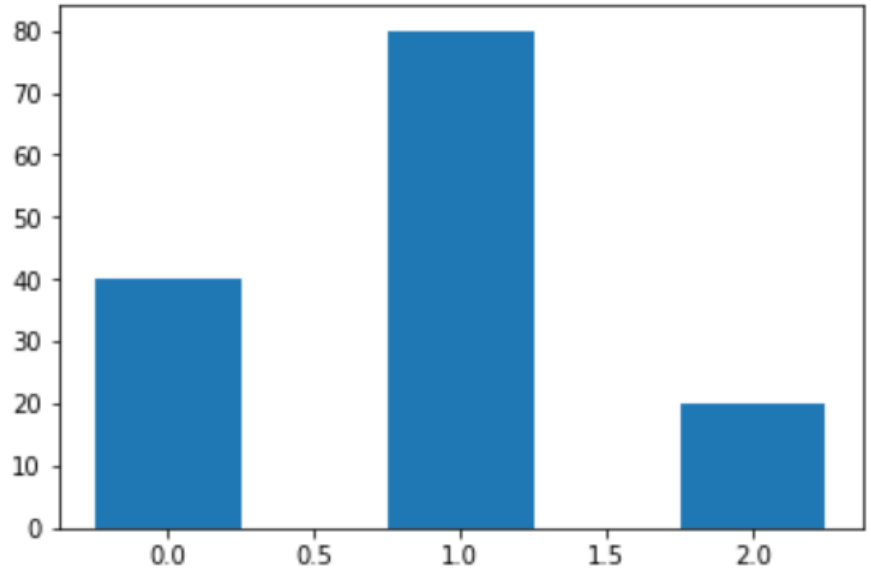
Bar Charts

- With **bar charts**, each column represents a group defined by a categorical variable whereas with histograms, each column represents a group defined by a continuous, quantitative variable
- The basic pyplot function for plotting a bar chart is **plt.bar()**, which makes a plot of rectangular bars defined by their *centers* and *height*
- For example `plt.bar([0, 1, 2], [40, 80, 20])`
- The width of the rectangles is, by default, 0.8 but can be set with the (third) width argument
- By default, plt.bar() produces a vertical bar chart
- Horizontal bar charts are catered for either by setting orientation='horizontal' or by using the analogous **plt.barh()** method

Bar Charts

- For example:

```
x = np.array([0, 1, 2])
y = np.array([40, 80, 20])
w = 0.5

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



Bar Charts

Experis KickStart ManpowerGroup

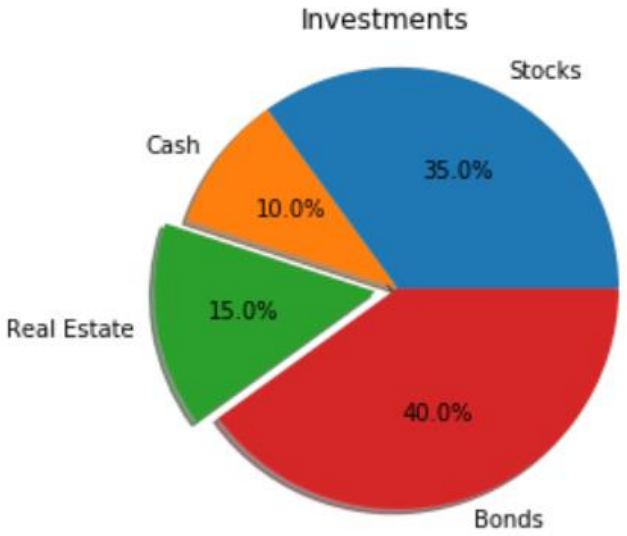
- Additional arguments for the bar charts are given in the following table:

Argument	Description
x	A sequence of *x*-coordinates of the centers of the bars
height	A sequence of heights for the bars
width	Width of the bars. If a scalar, all bars have the same width; can be array-like for variable widths
bottom	The *y*-coordinates of the bottom of the bars
color	Colors of the bar faces (scalar or array-like)
edgecolor	Colors of the bar edges (scalar or array-like)
linewidth	Line widths of the bar edges, in points (scalar or array-like)
xerr, yerr	Error bar limits, as for errorbar (scalar or array-like)
align	The default, 'center', centers the bars on this axis; 'edge' aligns the bars by their left edges (for vertical bars) or bottom edges (for horizontal bars) instead
log	Set to True to use a logarithmic axis scale
orientation	'vertical' (the default) or 'horizontal'

Pie Charts

- It is easy to draw a pie chart in Matplotlib by passing an array to **pyplot.pie()**
- The values will be normalized by their sum if this sum is greater than 1, or otherwise treated directly as fractions
  - The resulting pie will have an empty wedge of size 1 - sum(*x*)
- The wedges are plotted counterclockwise, by default starting from the x-axis

```
labels = ['Stocks', 'Cash', 'Real Estate', 'Bonds']
sizes = [35, 10, 15, 40]
explode = (0, 0, 0.1, 0)  # only "explode" the 3rd slice

plt.pie(sizes, explode=explode, labels=labels,
        autopct='%1.1f%%', shadow=True)
plt.axis('equal')   # so our pie looks round!
plt.title('Investments');
```



Pie Charts

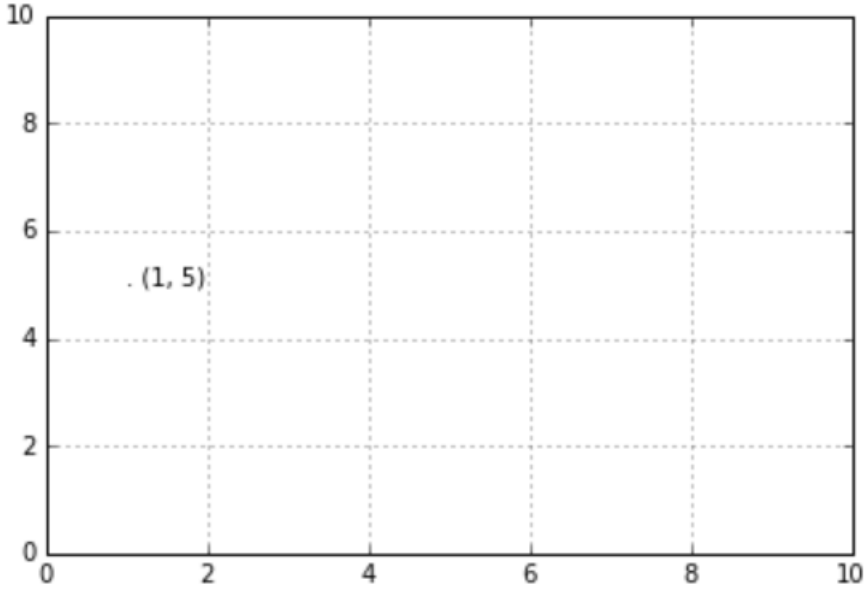
- Common arguments to pie.chart():

Argument	Description
colors	A sequence of Matplotlib color specifiers for coloring the segments
labels	A sequence of strings for labeling the segments
explode	A sequence of values specifying the fraction of the pie chart radius to offset each wedge by (0 for no explode effect)
shadow	True or False: specifies whether to draw an attractive shadow under the pie
startangle	Rotate the “start” of the pie chart by this number of degrees counterclockwise from the horizontal axis
autopct	A format string to label the segments by their percentage fractional value, or a function for generating such a string from the data
pctdistance	The radial position of the autopct text, relative to the pie radius. The default is 0.6 (i.e., within the pie, which can be awkward for narrow segments)
labeldistance	The radial position of the label text, relative to the pie radius; the default is 1.1 (just outside the pie)
radius	The radius of the pie (the default is 1); this is useful when creating overlapping pie charts with different radii

Text and Annotation

- Creating a good visualization involves guiding the reader so that the figure tells a story
- Matplotlib provides several ways to add different kinds of annotation to your plots
  - The most important annotations are text, arrows, lines and shapes
- The method **plt.text**(*x*, *y*, *s*) is a method to add a text string *s* at position (*x*, *y*) (in *data coordinates*) to the axes

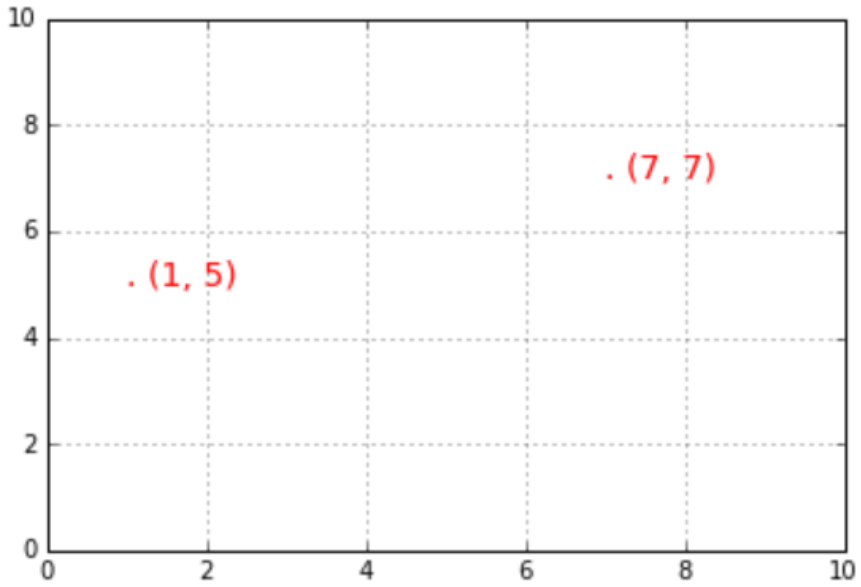
```
plt.axis([0, 10, 0, 10])
plt.grid()

plt.text(1, 5, ". (1, 5)");
```



Text Properties

- Individual keyword arguments can be used to specify text properties:

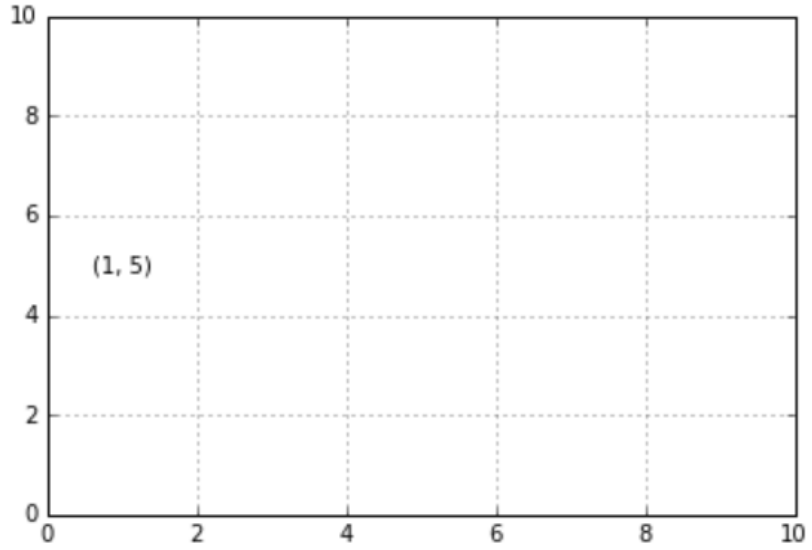
```
plt.axis([0, 10, 0, 10])
plt.grid()

style = dict(size=14, color='red')

plt.text(1, 5, ". (1, 5)", **style)
plt.text(7, 7, ". (7, 7)", **style);
```



Text Alignment

- By default, the text is aligned above and to the left of the specified coordinates
  - the "." at the beginning of each string will approximately mark the given coordinate location
- This can be changed using the ha (horizontalalignment) and va (verticalalignment) arguments, which determine how the label is aligned relative to its position
  - Valid values are ’center’, ’right’, ’left’, ’top’, ’bottom’ and ’baseline’ as appropriate

```
plt.axis([0, 10, 0, 10])
plt.grid()

plt.text(1, 5, "(1, 5)", ha='center', va='center');
```



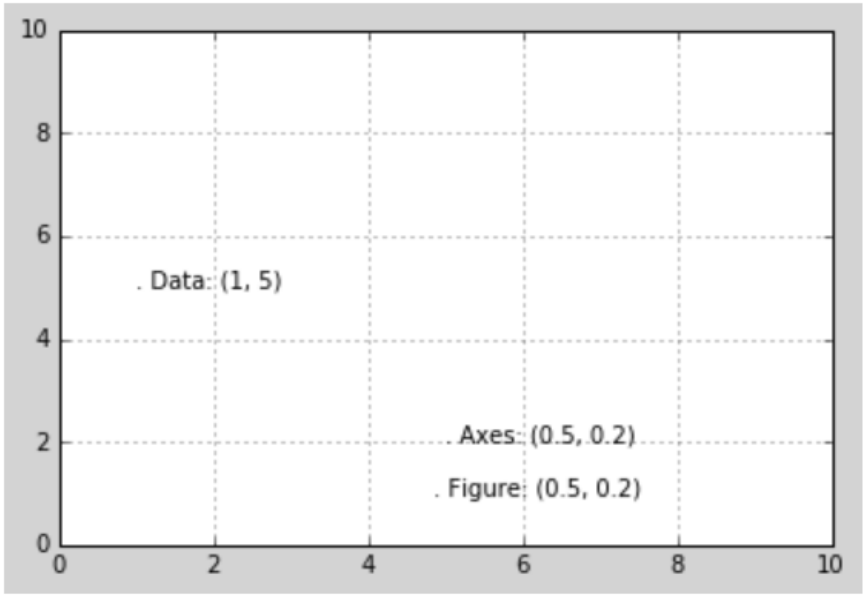
Transforms and Text Position

- Sometimes it's preferable to anchor the text to a position on the axes or figure, independent of the data
- This is done by modifying the **transform** argument
- There are three pre-defined transforms that can be useful in this situation:
  - **ax.transData**: Transform associated with data coordinates (the default)
  - **ax.transAxes**: Transform associated with the axes (in units of axes dimensions)
    - (0,0) is lower left, (1,1) is upper right
  - **fig.transFigure**: Transform associated with the figure (in units of figure dimensions)
    - (0,0) is lower left, (1,1) is upper right

Transforms and Text Position

- Let's look at an example of drawing text at various locations using these transforms:

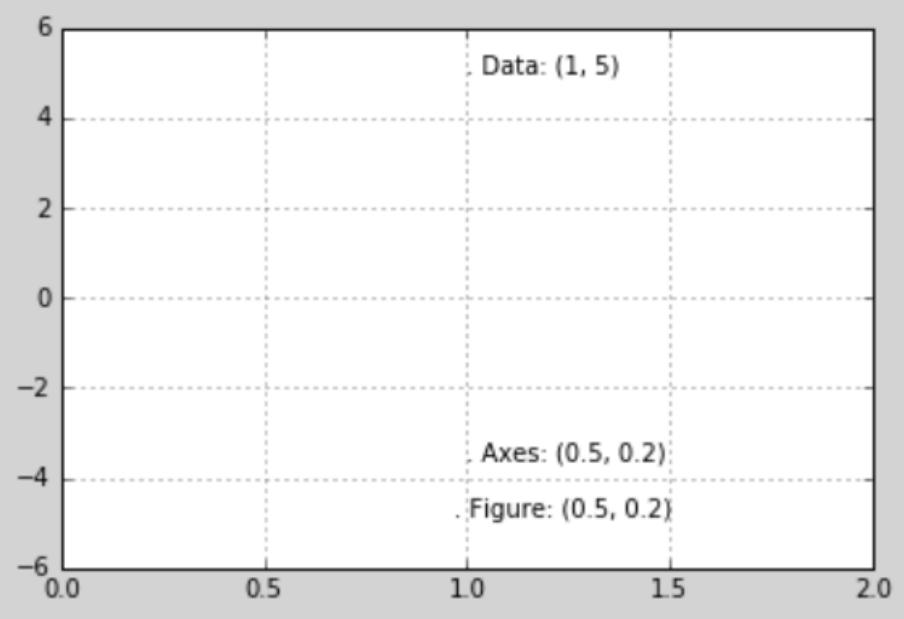
```
fig = plt.figure(facecolor='lightgray')
ax = plt.axes()
ax.axis([0, 10, 0, 10])
ax.grid()

ax.text(1, 5, ". Data: (1, 5)")
ax.text(0.5, 0.2, ". Axes: (0.5, 0.2)", transform=ax.transAxes)
ax.text(0.5, 0.2, ". Figure: (0.5, 0.2)", transform=fig.transFigure);
```



Transforms and Text Position

- Notice now that if we change the axes limits, it is only the transData coordinates that will be affected, while the others remain stationary:

```
ax.set_xlim(0, 2)
ax.set_ylim(-6, 6)
fig
```



Arrows and Annotation

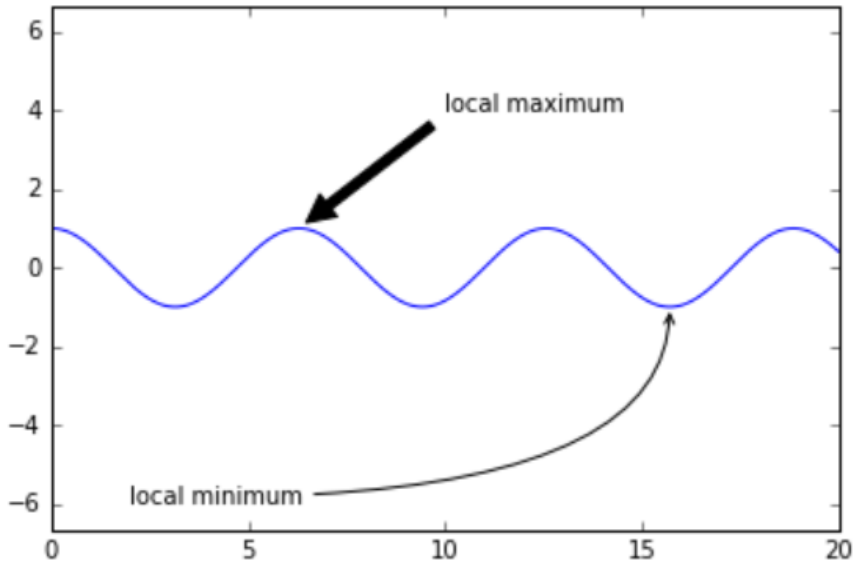
Experis KickStart ManpowerGroup

- Along with tick marks and text, another useful annotation mark is the simple arrow
- The **plt.annotate()** method is similar to plt.text(), but draws an arrow from the text to a specified point in the plot. The important arguments to plt.annotate() are:

Argument	Description
s	the string to output as a text label
xy	a tuple, (x,y) giving the coordinates of the position to annotate (i.e., where the arrow points to)
xytext	a tuple, (x,y) giving the coordinates of the text label (i.e., where the arrow points from)
xycoords	an optional string determining the type of coordinates referred to by the argument xy: – 'data': data coordinates, the default – 'axes fraction': fractional coordinates of the axes – 'figure fraction': fractional coordinates of the figure size
textcoords	as for xycoords, an optional string determining the type of coordinates referred to by xytext. An additional value is permitted for this string: 'offset points' specifies that the tuple xytext is an offset in points from the xy position.
arrowprops	if present, determines the properties and style of the arrow drawn between xytext and xy

Arrows and Annotation

```
x = np.linspace(0, 20, 1000)
plt.plot(x, np.cos(x))
plt.axis('equal')

plt.annotate('local maximum', xy=(2 * np.pi, 1), xytext=(10, 4),
             arrowprops=dict(facecolor='black', shrink=0.05))
plt.annotate('local minimum', xy=(5 * np.pi, -1), xytext=(2, -6),
             arrowprops=dict(arrowstyle='->',
                             connectionstyle='angle3,angleA=0,angleB=-90'));
```



Contour Plots

Experis KickStart ManpowerGroup

- **pyplot.contour()** makes a contour plot of a provided two-dimensional array
- In its simplest invocation, **contour(Z)**, no further arguments are required
  - The (x, y) values, which represents positions on the plot, are taken as the indexes of the 2D array Z and the contour intervals are selected automatically from the Z values
- To explicitly include (x, y) coordinates, pass them as **contour(X, Y, Z)**
  - The arrays X and Y must have the same shape as Z
- The contour levels can be controlled by a further argument:
  - either a scalar, N, giving the total number of contour levels
  - or a sequence, V, explicitly listing the values of Z at which to draw contours
- The contours are colored according to Matplotlib's default **colormap**
  - The Z values are normalized linearly onto the interval [0,1], which is then mapped onto a list of colors that are used to style the contours at the corresponding values

Contour Plots

- The module matplotlib.cm provides several colormap schemes
- As an alternative, contour supports the **colors** argument which takes either a single Matplotlib color specifier or a sequence of such specifiers
  - For single-color contour plots, contours corresponding to negative values are plotted in dashed lines
- The widths of the contour lines can be styled individually or all together with the argument **linewidths**

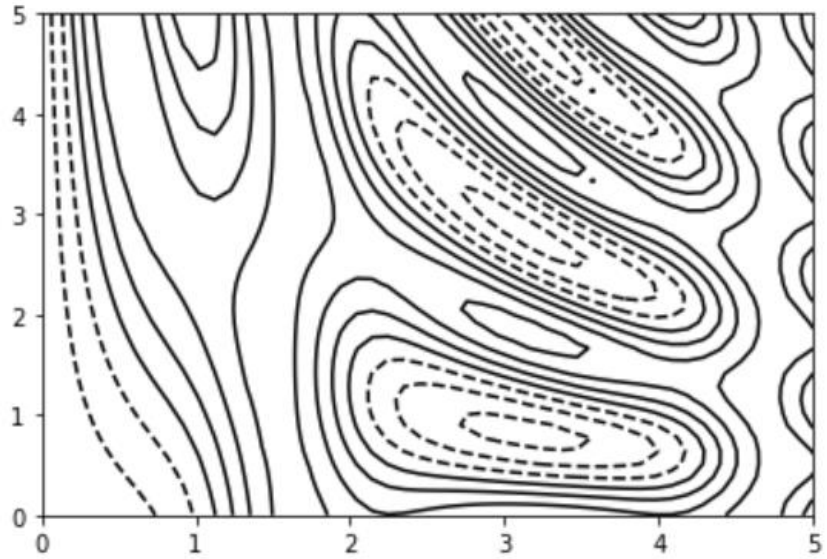
Contour Plots

- We'll demonstrate a contour plot using the following function z = f(x, y):

```
def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

- The most straightforward way to prepare a grid of x and y values for the contour plot is to use **np.meshgrid()**, which builds 2D grids from 2 one-dimensional arrays:

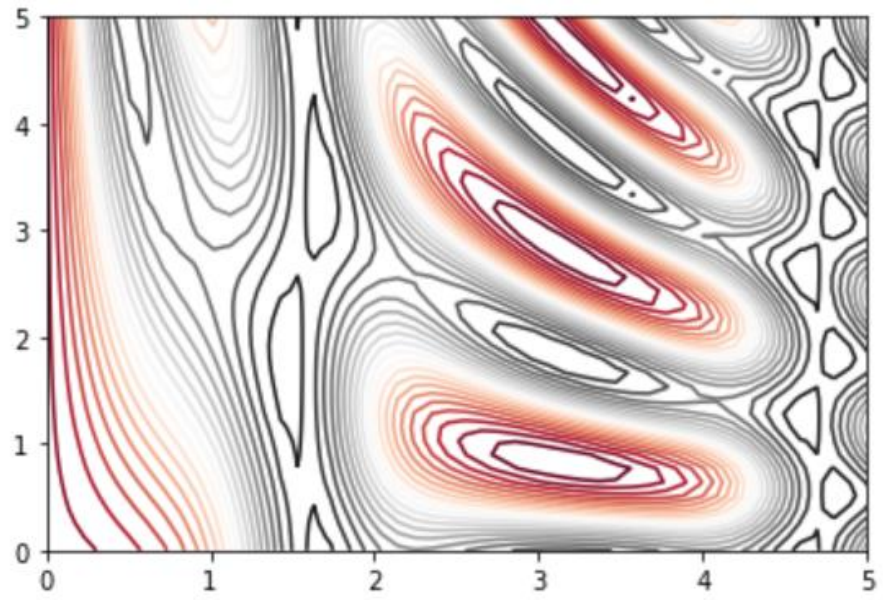
```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)

plt.contour(X, Y, Z, colors='black');
```



Contour Plots

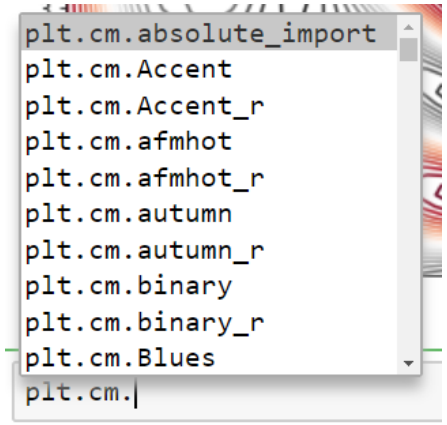
- The lines can be color-coded by specifying a colormap with the **cmap** argument
- Here, we'll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range:

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```



Color Maps

- Matplotlib has a wide range of colormaps available, which you can easily browse in IPython by doing a tab completion on the plt.cm module:

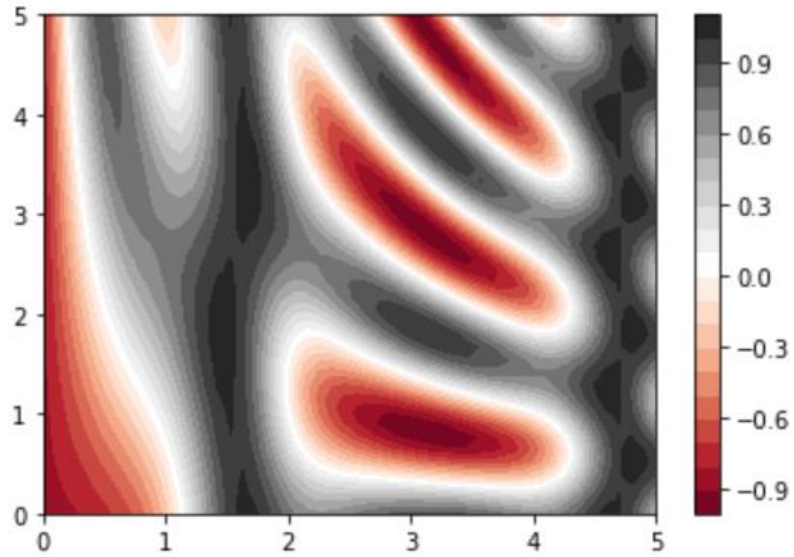


- Some of the more practical ones are: cm.hot, cm.bone, cm.winter, cm.jet, cm.Greys and cm.hsv
- If you want to use a colormap with its colors reversed, tack a _r on the end of its name (e.g, cm.hot_r)

Filled Contour Plots

- We can switch to a filled contour plot using the **plt.contourf()** function
- This function uses largely the same syntax as plt.contour()
- Additionally, we'll add a **plt.colorbar()** command, which automatically creates an additional axis with labeled color information for the plot:

```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```



Stylesheets

- Matplotlib has a convenient **style** module, which includes a number of default stylesheets, as well as the ability to create and package your own styles
- These stylesheets are formatted similarly to the *.matplotlibrc* files, but must be named with a *.mplstyle* extension
- The available styles are listed in plt.style.available:

```
plt.style.available[:10]
```

```
['bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn-bright',
 'seaborn-colorblind',
 'seaborn-dark-palette']
```

Stylesheets

Experis KickStart ManpowerGroup

- The basic way to switch to a stylesheet is to call:

```
plt.style.use('stylename')
```

- But this will change the style for the rest of the session!
- Alternatively, you can use the style context manager, which sets a style temporarily:

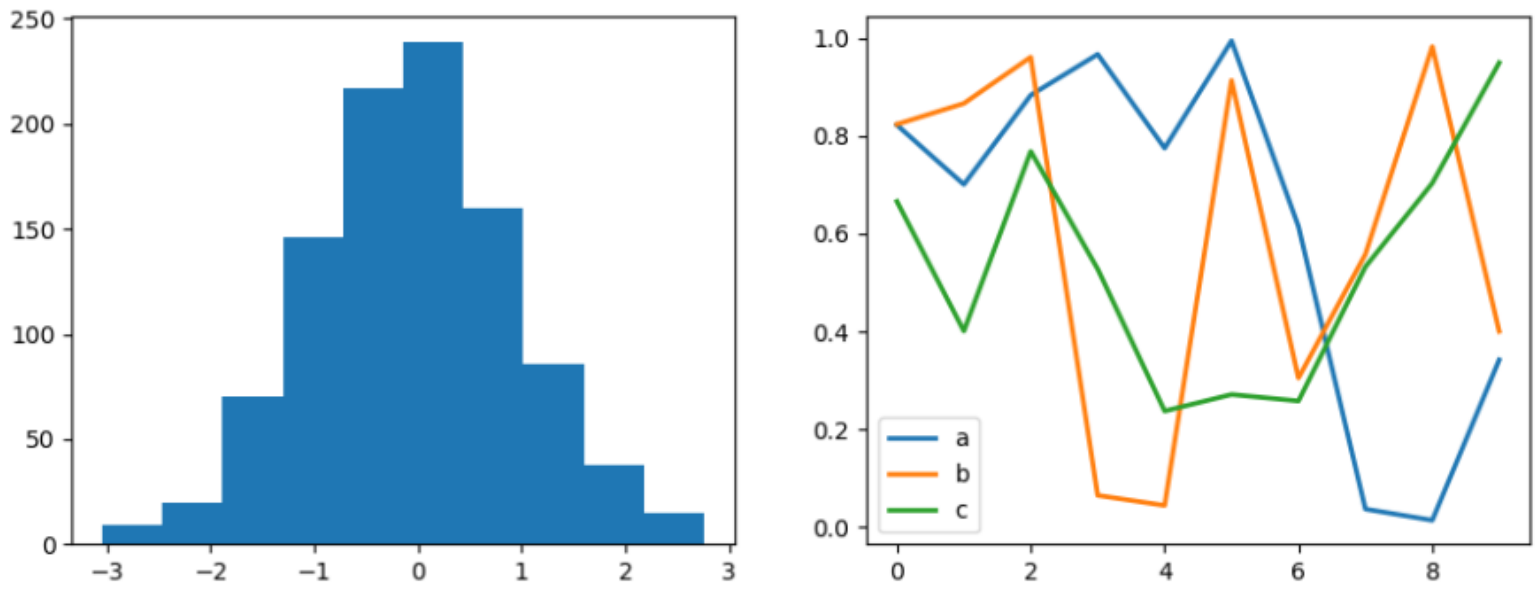
```
with plt.style.context('stylename'):
    make_a_plot()
```

- Let's create a function that will make two basic types of plot, so we can explore how these plots look like using various built-in styles:

```
def hist_and_lines():
    np.random.seed(0)
    fig, ax = plt.subplots(1, 2, figsize=(11, 4))
    ax[0].hist(np.random.randn(1000))

    for i in range(3):
        ax[1].plot(np.random.rand(10))
    ax[1].legend(['a', 'b', 'c'], loc='lower left')
```

Default Style

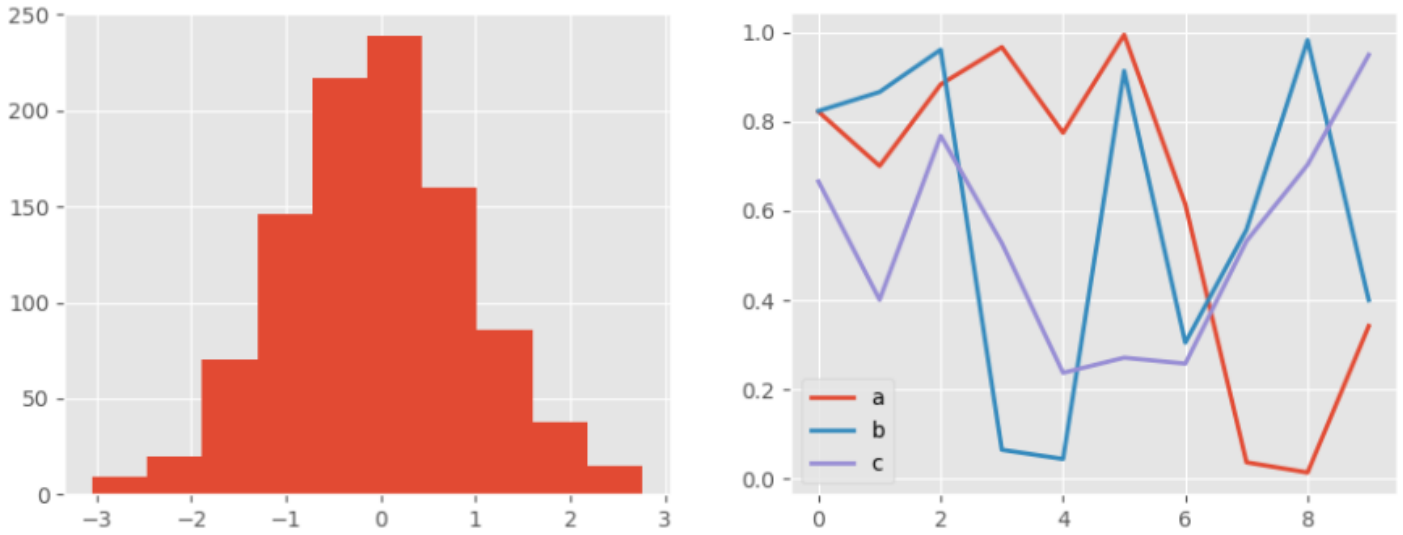
```
hist_and_lines()
```



ggplot

- The ggplot package in the R language is a very popular visualization tool
- Matplotlib's ggplot style mimics the default styles from that package:

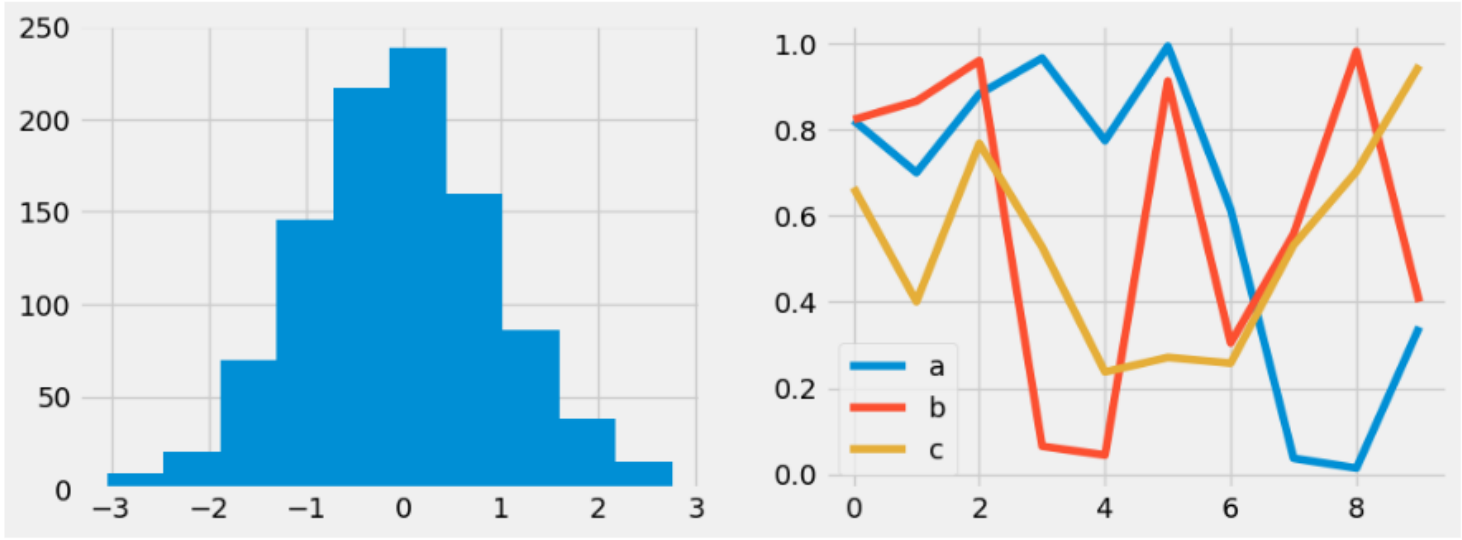
```
with plt.style.context('ggplot'):
    hist_and_lines()
```



FiveThirtyEight Style

- The fivethirtyeight style mimics the graphics found on the popular FiveThirtyEight website
- It is typified by bold colors, thick lines, and transparent axes:

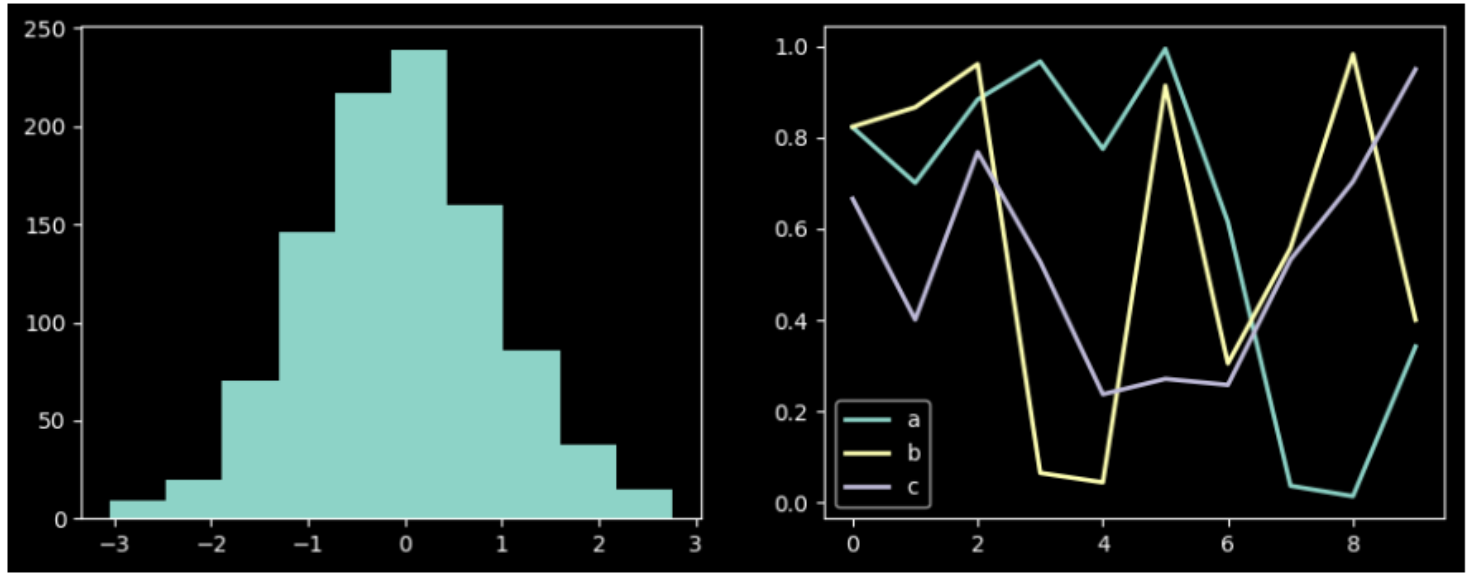
```
with plt.style.context('fivethirtyeight'):
    hist_and_lines()
```



Dark Background

- For figures used within presentations, it is often useful to have a dark rather than light background
- The dark_background style provides this:

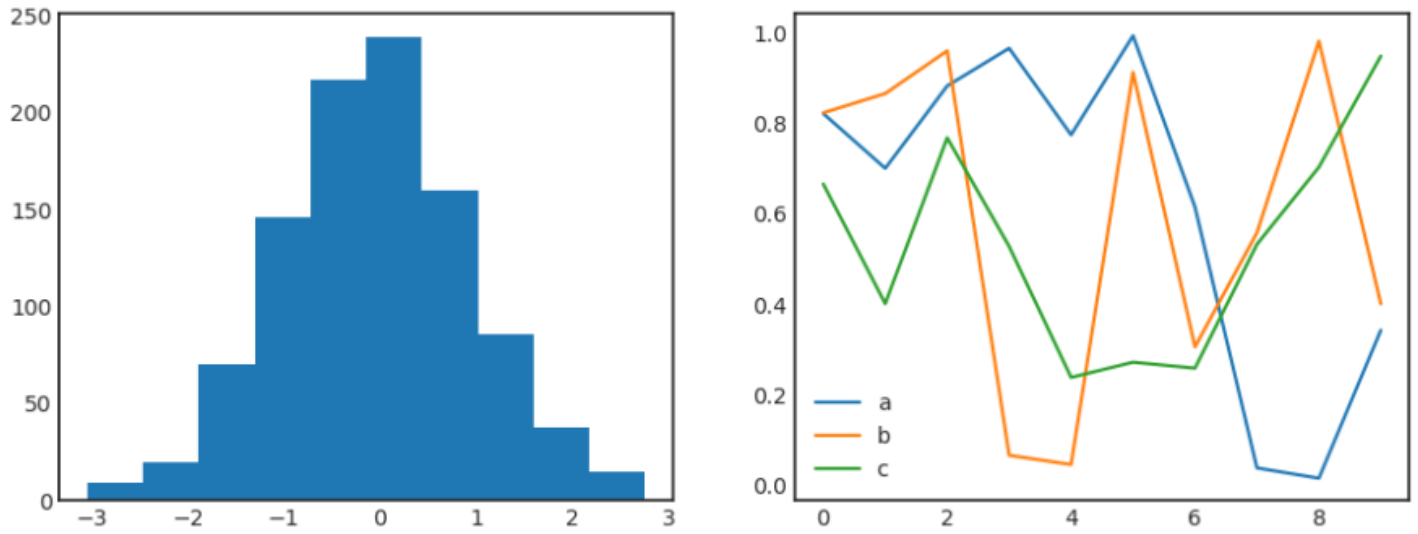
```
with plt.style.context('dark_background'):
    hist_and_lines()
```



Seaborn Style

- Matplotlib also has stylesheets inspired by the Seaborn library
- For example, using the seaborn-white style:

```
with plt.style.context('seaborn-white'):
    hist_and_lines()
```



Visualization with Seaborn

Experis KickStart ManpowerGroup

- Matplotlib has proven to be an incredibly useful and popular visualization tool, but it has its own shortcomings:
  - Matplotlib's defaults are not exactly the best choices. It was based off of MATLAB circa 1999, and this often shows.
  - Matplotlib's API is relatively low level. Doing sophisticated statistical visualization is possible, but often requires a *lot* of boilerplate code.
  - Matplotlib predated Pandas by more than a decade, and thus is not designed for use with Pandas DataFrames. In order to visualize data from a Pandas DataFrame, you must extract each Series and often concatenate them together into the right format.
- An answer to these problems is 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

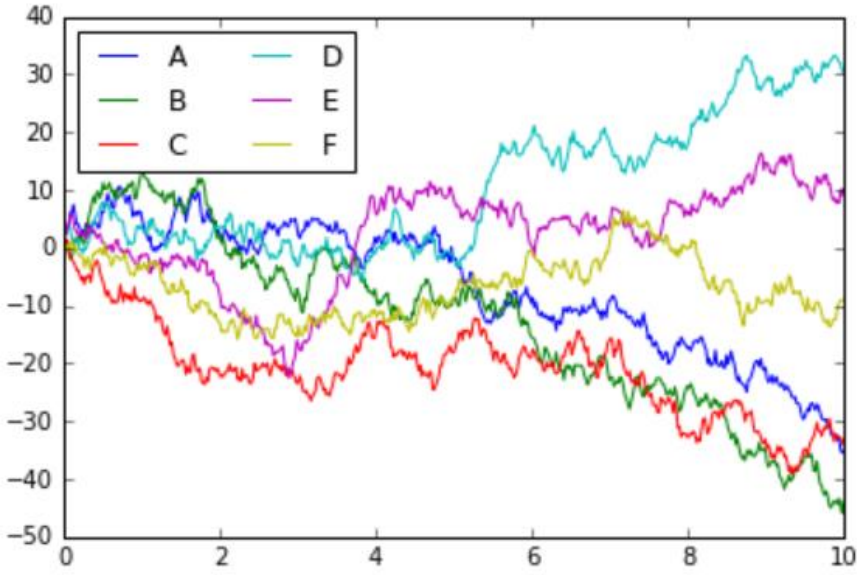
Seaborn Versus Matplotlib

- Here is an example of a simple random-walk plot in Matplotlib, using its classic plot formatting and colors:

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

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

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



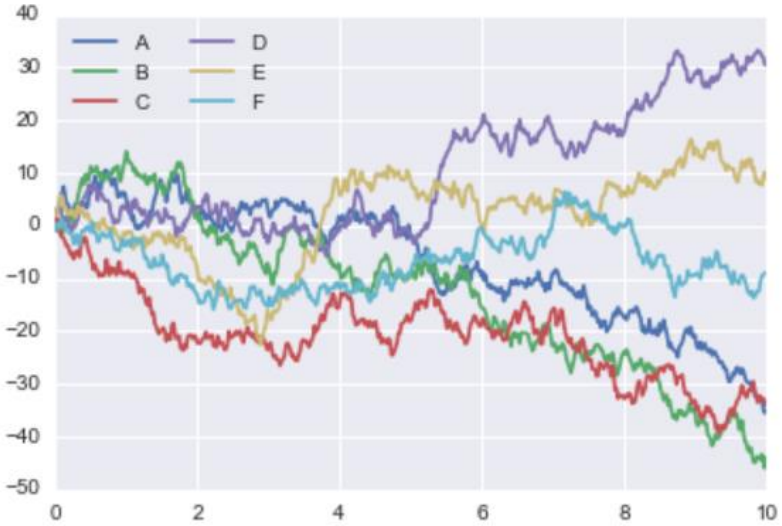
Seaborn Versus Matplotlib

- Now let's take a look at how it works with Seaborn
- By convention, Seaborn is imported as sns
- We can set the style by calling Seaborn's set() method:

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

- Now let's rerun the same three lines as before:

```
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left')
plt.show()
```



- Ah, much better!