Visualization with Matplotlib

April 17, 2015

1 Overview of `matplotlib`

So far in our course, we've covered basic `Python` to more advanced features of `Python`'s array processing and data analysis libraries. While we have gotten into the meat of handling numbers themselves, it would be nice to have a library of tools to visualize these underlying data in a powerful but aesthetic way. The solution, which has become a massive open-source project in its own right, is `matplotlib`. From the `matplotlib` homepage:

> `matplotlib` is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB or Mathematica), web application servers, and six graphical user interface toolkits.
>
> `matplotlib` tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code. For a sampling, see the screenshots, thumbnail gallery, and examples directory.
>
> For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Let's import all of the libraries we will use in this session.

```
In [1]: from __future__ import division
        import numpy as np
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        plt.rc('figure', figsize=(5, 5))
        from pandas import Series, DataFrame
        import pandas as pd
        np.set_printoptions(precision=4)
```

When you use the `IPython` notebook, you can print plots to the output of individual cells by including the magic command:

```
In [2]: %matplotlib inline
```

There are two ways to think about creating and displaying plots using `matplotlib`. The first, and simpler, approach is the imperative, "scripting" paradigm. Modeled after the plotting functionality of MATLAB, this gives you an easy way to generate a large quantity of plots.

The second paradigm is the object-oriented approach, which requires a larger amount of initial code, but with a much higher degree of flexibility and robust functionality.

2 The MATLAB approach

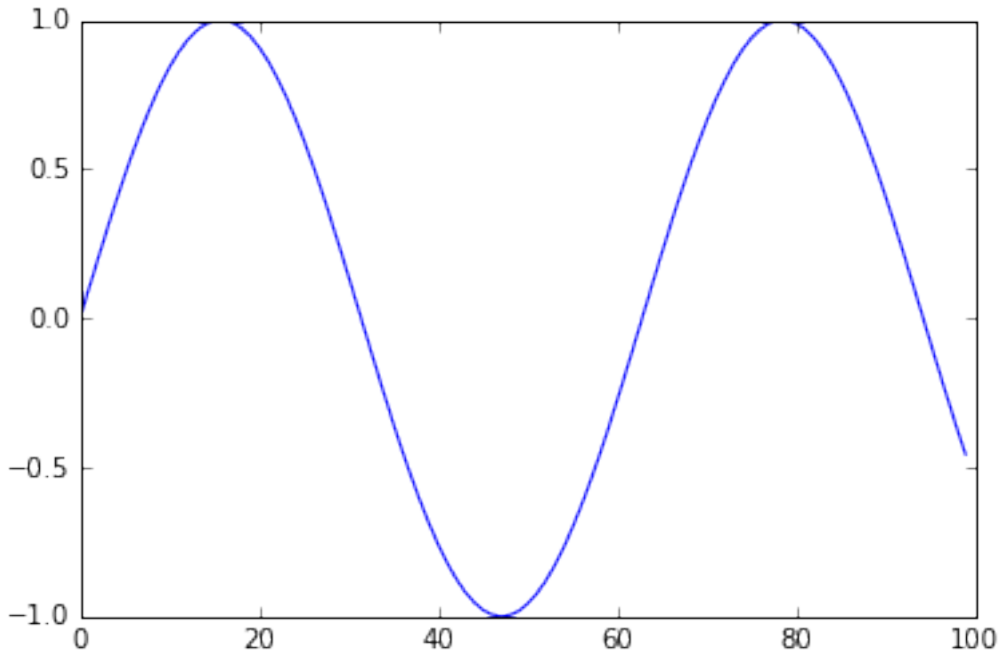
The main module that we use to generate plots is the `pyplot` submodule of `matplotlib`. According to established convention, we import this module as follows:

```
In [3]: import matplotlib.pyplot as plt
```

From now on, we will use `plt` to denote methods and fields in the `pyplot` module. Here is a simple demonstration of the MATLAB approach to plotting.

```
In [4]: x = np.arange(0,10,0.1) # generates an ndarray from 0 to 9.9
        plt.plot(np.sin(x))
```

```
Out[4]: [<matplotlib.lines.Line2D at 0x7fe665a9a4d0>]
```

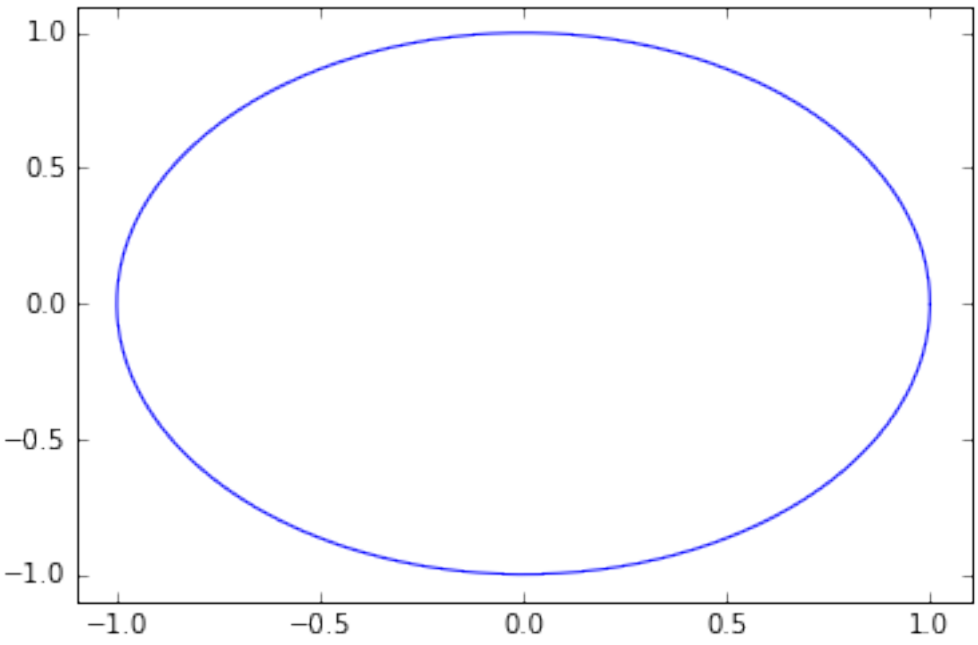


The `plot` function takes an array-like type and produces a line plot of the array. If you give `plot` a single array, it will implicitly assume that you mean to plot coordinate pairs $(i, arr[i])$, where i is an integer.

Instead, you can pass two arrays x and y (of the same size), which produces a plot of coordinates $(x[i], y[i])$.

```
In [5]: x = np.arange(0,2.0*np.pi, 0.01)
        plt.plot(np.cos(x),np.sin(x))
        plt.xlim([-1.1,1.1])
        plt.ylim([-1.1,1.1])
```

```
Out[5]: (-1.1, 1.1)
```



It is very simple to customize the style of the plots.

```
In [6]: plt.plot?
```

Line color and marker arguments By passing in an optional character argument, you can specify the color of the line being plotted.

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

Alternatively, you can specify a custom (e.g. hexadecimal) color by passing a `color=#123456` argument. For customizing the line marker shapes, you can specify from a number of built-in arguments.

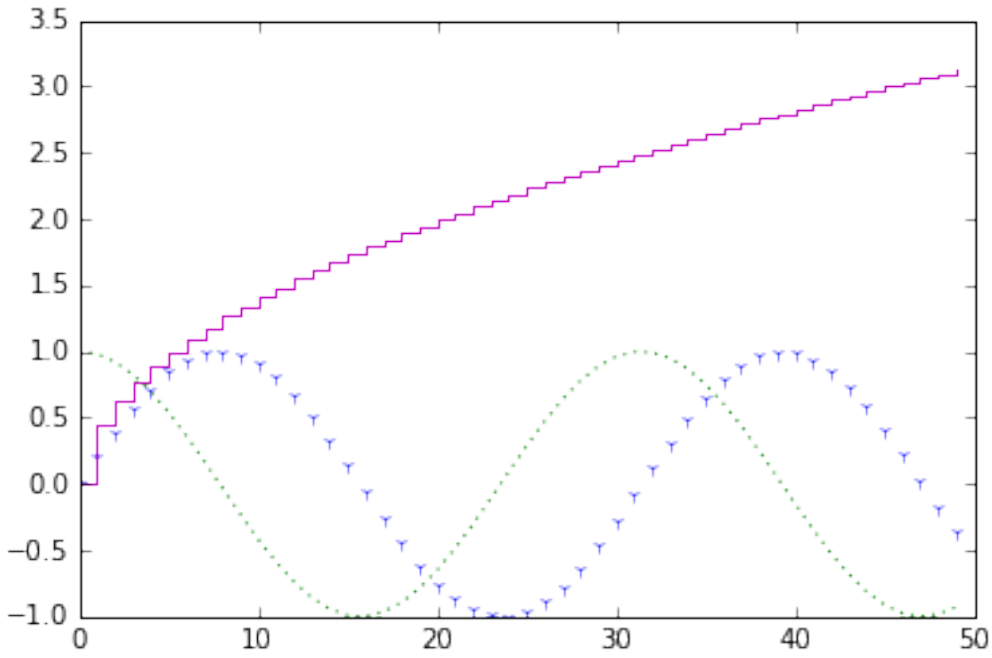
Character	Description	Character	Description
`'-'`	solid line style	`'3'`	tri_left marker
`'--'`	dashed line style	`'4'`	tri_right marker
`'-.'`	dash-dot line style	`'s'`	square marker

Character	Description	Character	Description
':'	dotted line style	'p'	pentagon marker
'.'	point marker	'*'	star marker
','	pixel marker	'h'	hexagon1 marker
'o'	circle marker	'H'	hexagon2 marker
'v'	triangle_down marker	'+'	plus marker
'^'	triangle_up marker	'x'	x marker
'<'	triangle_left marker	'D'	diamond marker
'>'	triangle_right marker	'd'	thin_diamond marker
'1'	tri_down marker	'_'	hline marker
'2'	tri_up marker		

There are even more keyword arguments, but we won't go into the details here. Here is a simple example:

```
In [7]: x = np.arange(0,10,0.2)
        plt.plot(np.sin(x), '1')
        plt.plot(np.cos(x), ':')
        plt.plot(np.sqrt(x), 'm', drawstyle='steps-post')
```

```
Out[7]: [<matplotlib.lines.Line2D at 0x7fe665920e10>]
```

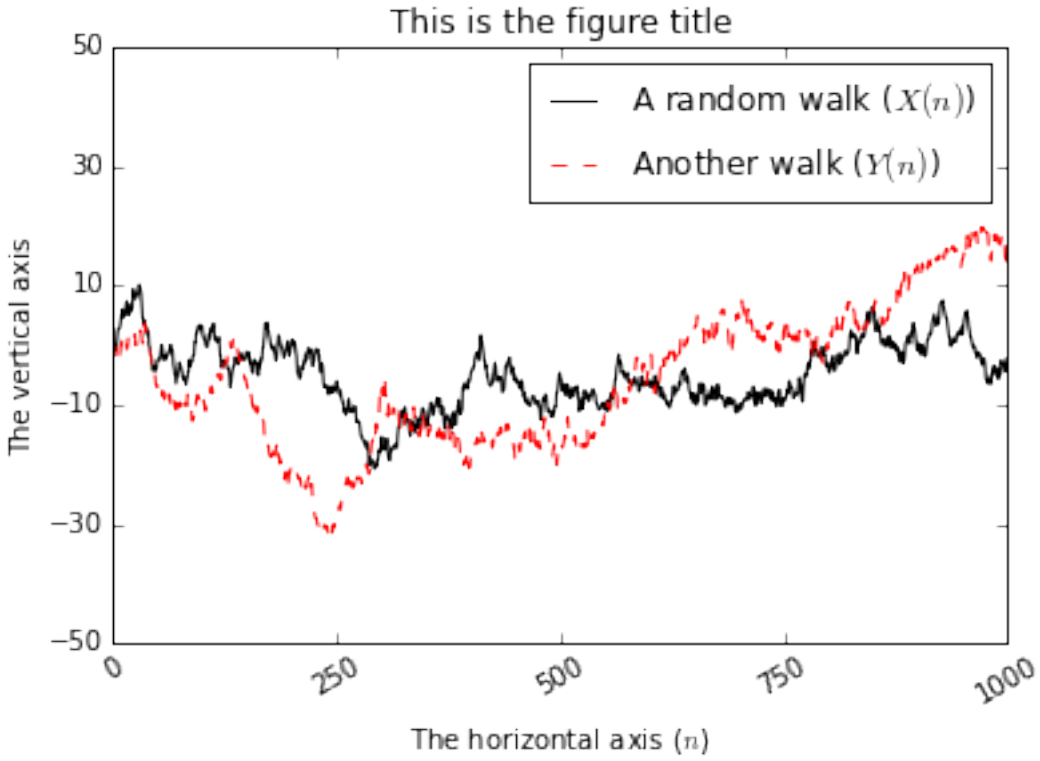


Notice that calling `plot` several times in one cell allows you to plot several graphs on one figure.

Now, let's customize the title, labels, legend, and ticks of the plot. In the MATLAB paradigm, we can use simple figure methods like `title`, `xlabel`, and `ylabel`, as well as call the `legend` method. To specify a legend string, include the optional argument `label` in the plot method.

```
In [8]: x = np.random.randn(1000)
        y = np.random.randn(1000)

        plt.plot(x.cumsum(), 'k', label='A random walk ($X(n)$)')
        plt.plot(y.cumsum(), 'r--', label='Another walk ($Y(n)$)')

        # Title and labels
        plt.title('This is the figure title')
        plt.xlabel('The horizontal axis ($n$)')
        plt.ylabel('The vertical axis')

        # Tick values
        plt.xticks(range(0,1001,250), rotation=30)
        plt.yticks(range(-50,51, 20))

        plt.legend(loc='best') # especially useful for random data
```

```
Out[8]: <matplotlib.legend.Legend at 0x7fe6658deb90>
```



If you notice carefully, `matplotlib` can render LaTeXin title, axis, and legend strings. Simply include the LaTeXdollar sign and `matplotlib` will do the rest for you. Now, there are tricky grey-areas with this functionality. For example, if you want to typeset the Greek letter τ on your plots, `matplotlib` will not properly interpret the string. (Why is this? See if you can figure out why.) To force `matplotlib` to interpret strings literally, you can instead write `r'$\tau$'`, which will tell `matplotlib` to ignore the formatting ambiguity.

There are tons of ways to customize your plots further, but we'll leave this to your exploration of the `matplotlib` documentation.

3 The object-oriented approach

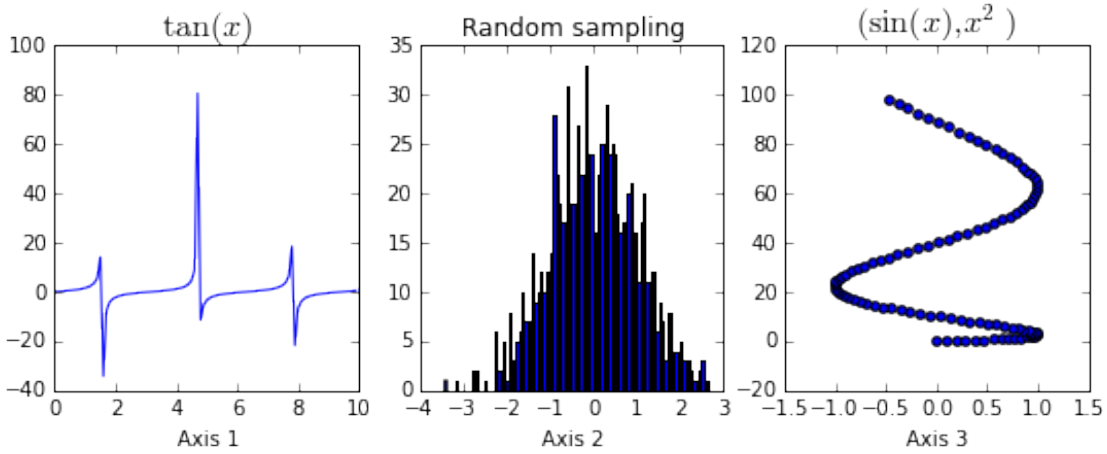
Whereas in the MATLAB approach, all plotting activity was centered around the `matplotlib` *figure*, the object-oriented approach shifts this attention to the *axis*.

When you begin plotting, you first initialize a figure and then add axes to the figure. Each axis now functions as its own plotting environment, which allows you to specify all of the previous functions nearly identically as before.

Why go to all of this work to specify the axis objects? The immediate advantage is that you can now easily construct **several** axes on one figure, which is an ability I have personally found incredibly useful.

Here is a simple way to get started:

```
In [9]: fig = plt.figure(figsize=(9,3)) # instantiate a new figure object

        # Add three axes aligned horizontally
        ax1 = fig.add_subplot(1,3,1)
        ax2 = fig.add_subplot(1,3,2)
        ax3 = fig.add_subplot(1,3,3)

        x = np.arange(0.0, 10.0, 0.1)

        # Simple plot
        ax1.plot(x, np.tan(x))

        # Histogram plot
        ax2.hist(np.random.randn(1000), bins=100)

        # Scatter plot, parametric
        ax3.scatter(np.sin(x), x*x)

        ax1.set_xlabel("Axis 1")
        ax2.set_xlabel("Axis 2")
        ax3.set_xlabel("Axis 3")

        ax1.set_title(r"$\tan(x)$", fontsize=16)
        ax2.set_title("Random sampling")
        ax3.set_title(r"$(\sin(x),x^2)$", fontsize=16)

Out[9]: <matplotlib.text.Text at 0x7fe665722d50>
```

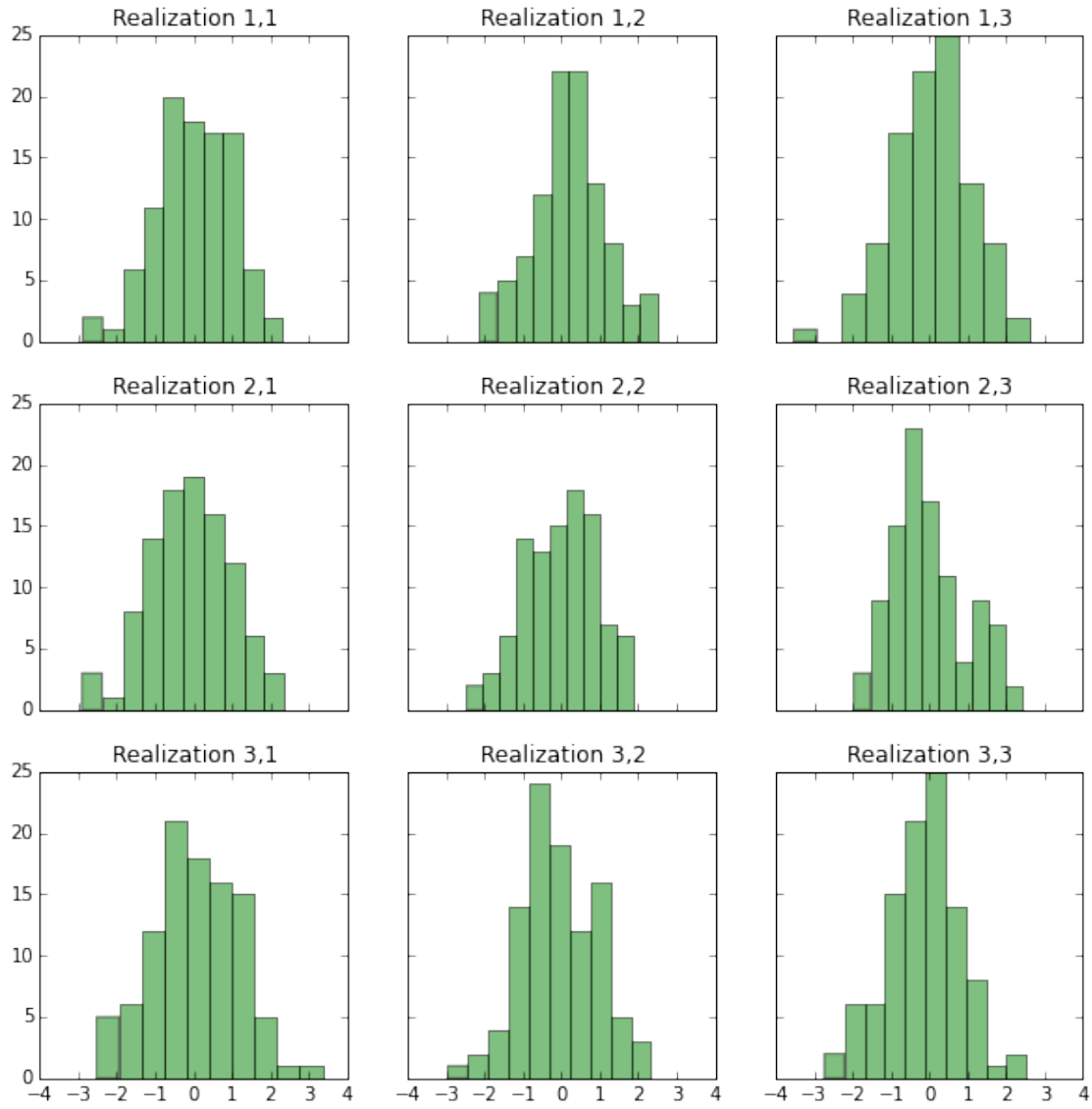


Here, we construct subplots by using the `add_subplot` method. The first two arguments of `add_subplot` indicate the number of rows and columns, respectively. Notice that for axes objects, we use `set_xlabel` and `set_title` instead of `xlabel` and `title`, but otherwise the functions work as one might expect compared to the MATLAB approach. This is generally the case for axes methods.

Beyond `plot`, `matplotlib` provides a host of other plotting methods, depending on exactly what your visualization needs. On display here is the `hist` and `scatter` methods. `hist` takes an array and plots the distribution of values of the array in a bar chart. `scatter` is similar to `plot`, but it requires exactly two arrays to generate coordinate pairs.

One way to deal with a large number of axes is to think about them as iterable objects. This can *dramatically* reduce the amount of code requisite to do sophisticated plots. For example:

```
In [10]: fig, axes = plt.subplots(3,3, figsize=(10,10), sharex=True, sharey=True)

         for i in range(3):
             for j in range(3):
                 x = np.random.randn(100)
                 axes[i, j].hist(x, color='g', alpha=0.5)
                 axes[i, j].set_title("Realization %i,%i" % (i+1,j+1))

         plt.subplots_adjust(wspace=0.2,hspace=0.2)
```



Of note, you can specify whether two (or in the above case, *every*) subplots share an x or y-axis. This can be a nice technique to reduce the clutter around a plot. Another function that is useful for figure formatting is `subplots_adjust`, which allows you to specify the spacing between plots and the margins from the borders of the aggregate figure.

4 Plotting functions

Here we will use the MATLAB approach just for brevity of code. We have already seen `plot` fairly extensively, so now we will explore other `matplotlib` plotting functions that you might want to explore.

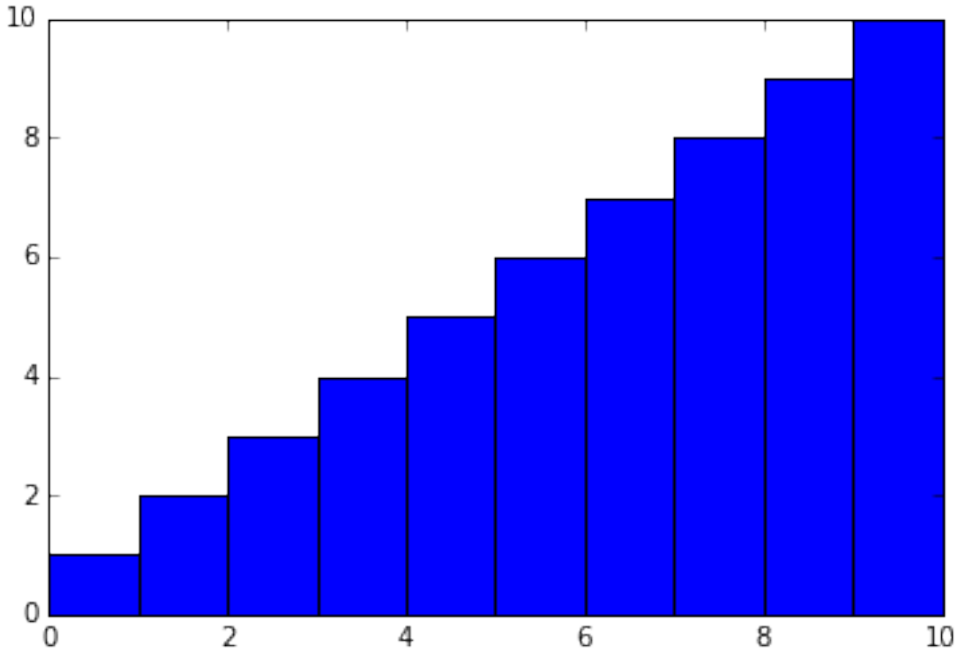
4.1 `bar` and `barh`

The `bar` and `barh` methods allow you to generate bar plots, with the distinction that `bar` orients the rectangles of the plot along the vertical axis while `barh` orients along the horizontal axis. Outside from orientation, both work identically (we will from now on assume `bar`).

`bar` takes an array denoting the x-coordinates of the left sides of the bars and an array denoting the heights of the bars. Optionally, you can add a scalar value (or array, for each bar) to denote the width of every bar. As with every plotting function, you can then specify color, transparency (alpha), and the legend label of the bars. For bar plots, you can add additional options, `xerr` and `yerr`, to specify the error bars in the x and y directions for the plot.

As a first example, we simply create a bar chart denoting increasing values:

```
In [11]: vals = np.arange(0,10,1)

         plt.bar(vals, vals + 1, 1)

Out[11]: <Container object of 10 artists>
```

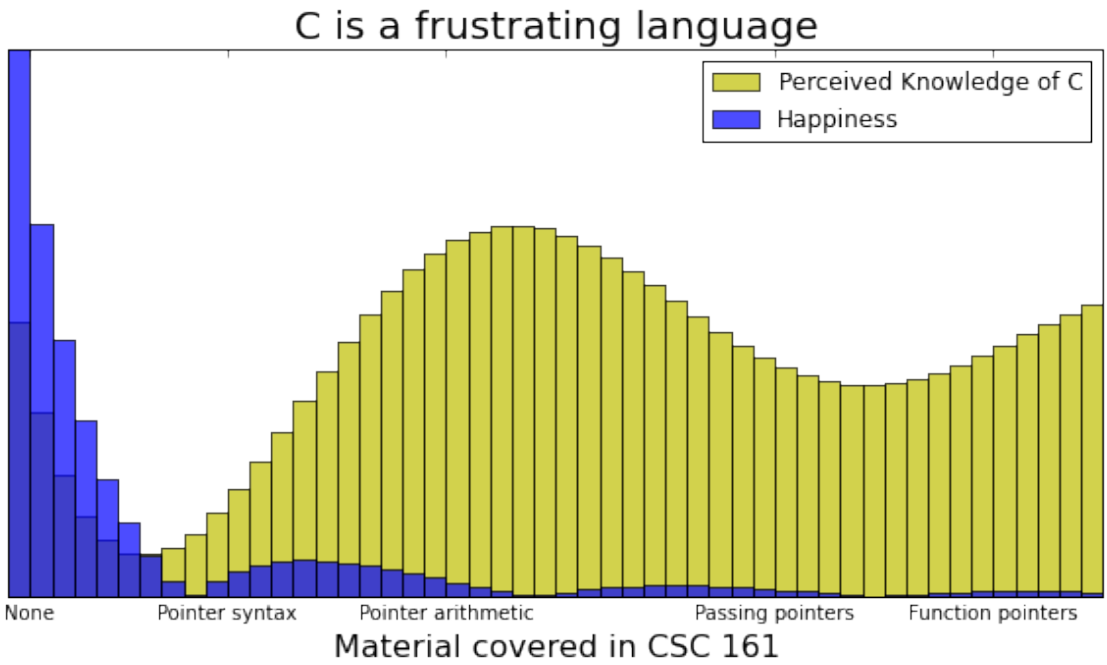


Of course, we can make this much more sophisticated. Here's a fun example that demonstrates some of the main features of `bar`.

```
In [12]: width = 0.2
         rows = np.arange(0,10, width)

         plt.figure(figsize=(10,5))

         data1 = 1.0 - 2.0/(rows +1.0) * np.sin(rows)
         data2 = 2./(rows + 1.)**2 * np.abs(np.cos(rows))

         plt.bar(rows, data1, width, color="y", alpha=0.7,
                 label="Perceived Knowledge of C")
         plt.bar(rows, data2, width, color="b", alpha=0.7, label="Happiness")
         plt.legend(loc="best")

         plt.xticks([0.2,2.0,4.0,7.0,9.0],
                    ("None", "Pointer syntax", "Pointer arithmetic",
```

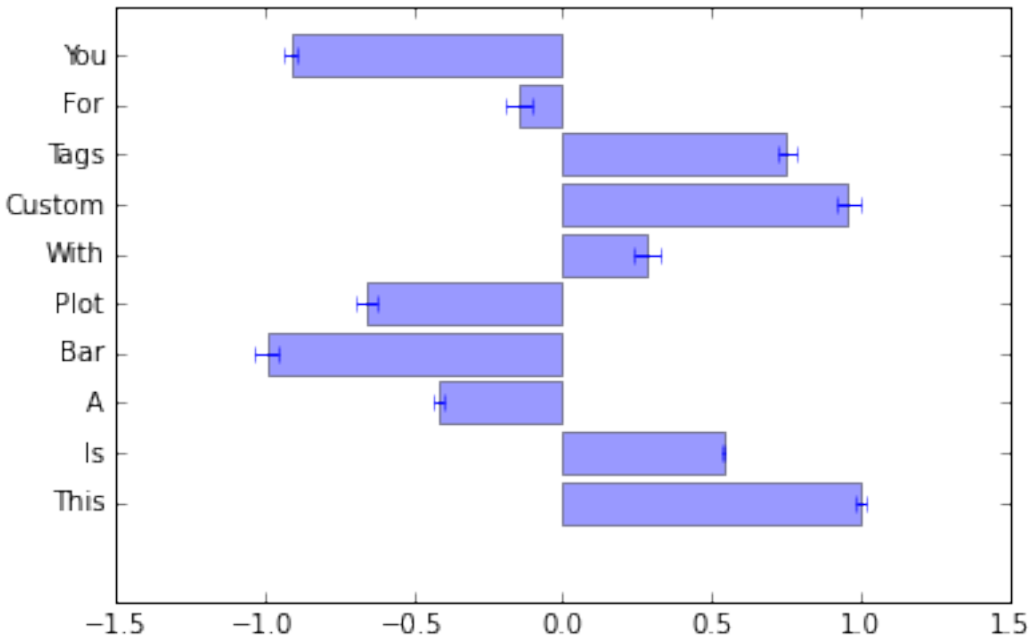
```
                    "Passing pointers",
                    "Function pointers"), rotation=0)

        plt.xlabel("Material covered in CSC 161", fontsize=16)
        plt.yticks([])
        plt.title("C is a frustrating language", fontsize=20)
```

```
Out[12]: <matplotlib.text.Text at 0x7fe664d51390>
```




```
In [13]: x = np.arange(0,10,1)

        plt.barh(x, np.cos(x), 0.85, align='center',
                 xerr= 0.05 * np.random.rand(np.size(x)), alpha=0.4)

        plt.yticks(x[:], ("This", "Is", "A", "Bar", "Plot", "With", "Custom",
                          "Tags", "For", "You", "To", "See"))
```

```
Out[13]: ([<matplotlib.axis.YTick at 0x7fe66482cb50>,
           <matplotlib.axis.YTick at 0x7fe664774550>,
           <matplotlib.axis.YTick at 0x7fe6646fca50>,
           <matplotlib.axis.YTick at 0x7fe6646467d0>,
           <matplotlib.axis.YTick at 0x7fe664646f10>,
           <matplotlib.axis.YTick at 0x7fe664650690>,
           <matplotlib.axis.YTick at 0x7fe664650dd0>,
           <matplotlib.axis.YTick at 0x7fe664659550>,
           <matplotlib.axis.YTick at 0x7fe664659c90>,
           <matplotlib.axis.YTick at 0x7fe664665410>],
          <a list of 10 Text yticklabel objects>)
```



4.2 Plotting functions in pandas

It's important to know how to build plots manually from data stored in `NumPy`. However, we can also use `Pandas` to produce high-quality `matplotlib` plots from existing `Series` and `DataFrame` objects with considerable ease and flexibility.

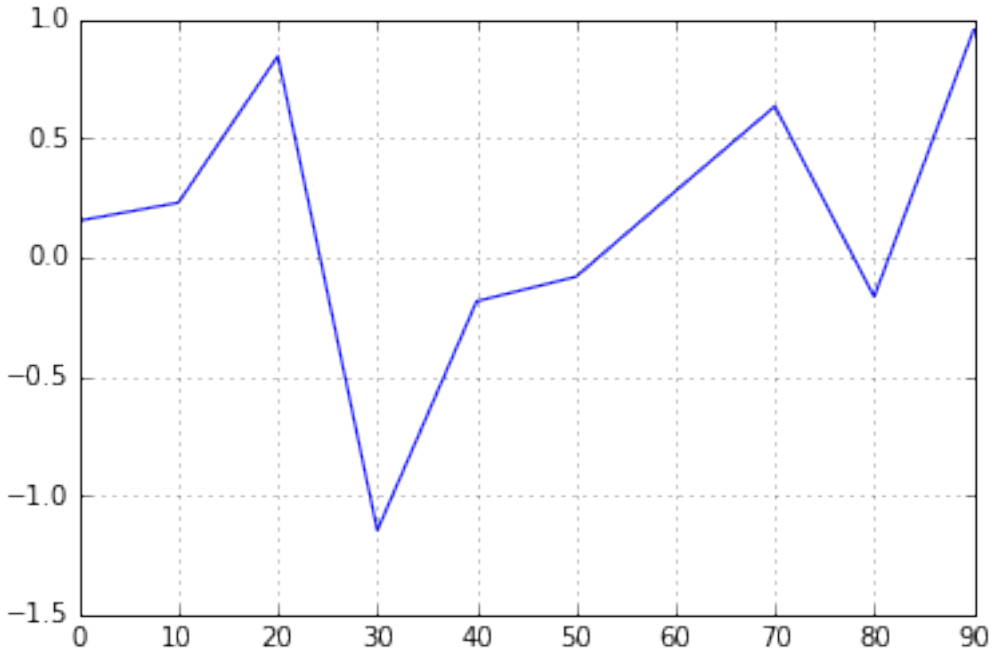
This section will walk through a variety of options you have at your disposal when building visualizations with `Pandas`, but is by no means exhaustive. For more information, check out the Pandas Documentation.

4.2.1 Line plots

Given a `Series` object, one natural approach to plotting is with line plots. This is the default behavior of the method `Series.plot`, displayed below.

```
In [14]: s = Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))
         s.plot()
```

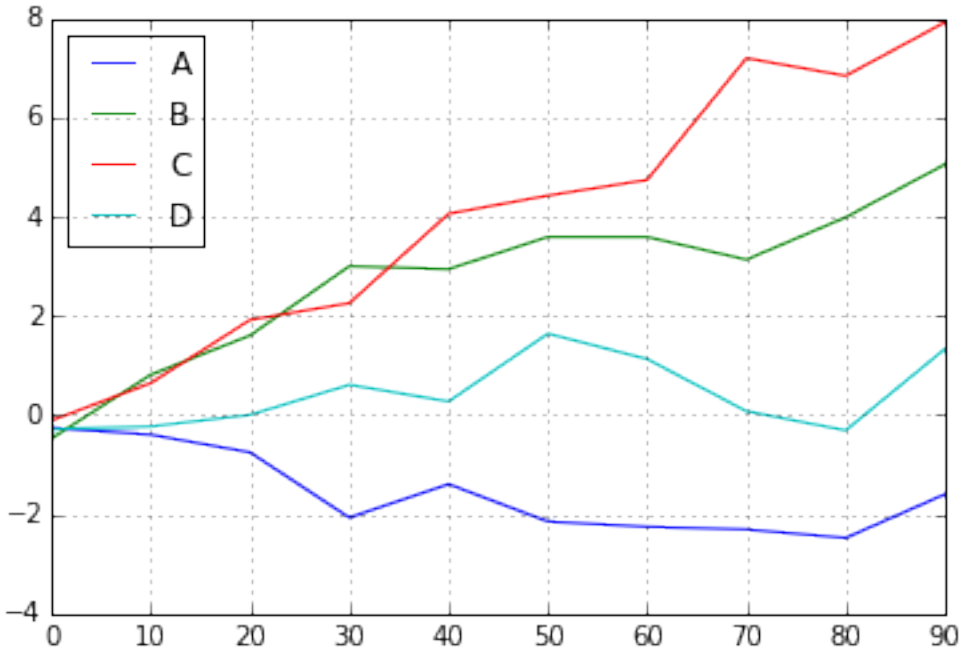
```
Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe6659664d0>
```



This extends naturally to the `DataFrame` object. Since `DataFrame` objects already label the columns of their internal data, it is also easy to produce legends.

```
In [15]: df = DataFrame(np.random.randn(10, 4).cumsum(0),
                        columns=['A', 'B', 'C', 'D'],
                        index=np.arange(0, 100, 10))
         df.plot()
```

```
Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe664594910>
```



4.2.2 Bar plots

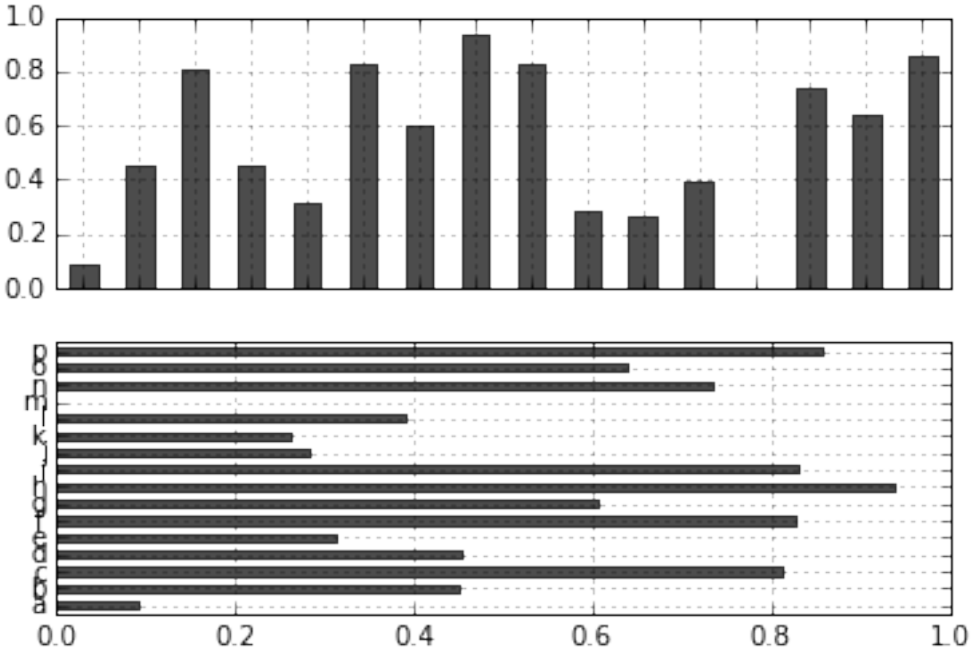
Of course, there are many more types of visualization than line plots. In general, one can specify the type of plot a `Series` or `DataFrame` generates by changing the optional parameter `kind`.

Here is an example showcasing the `bar` and `barh` plots we saw earlier. Additionally, we can specify the specific axis we want as the base of the plot. `Pandas` takes care of the formatting as well.

Plotting with `Pandas` plays nicely with both the MATLAB-style of generation, as with the prior examples, or with the Object-oriented paradigm, as below.

```
In [16]: fig, axes = plt.subplots(2, 1)
         data = Series(np.random.rand(16), index=list('abcdefghijklmnop'))
         data.plot(kind='bar', ax=axes[0], color='k', alpha=0.7)
         data.plot(kind='barh', ax=axes[1], color='k', alpha=0.7)
```

```
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe6643cb0d0>
```



Again, `DataFrame` objects have similar functionality. Consider the following `DataFrame`.

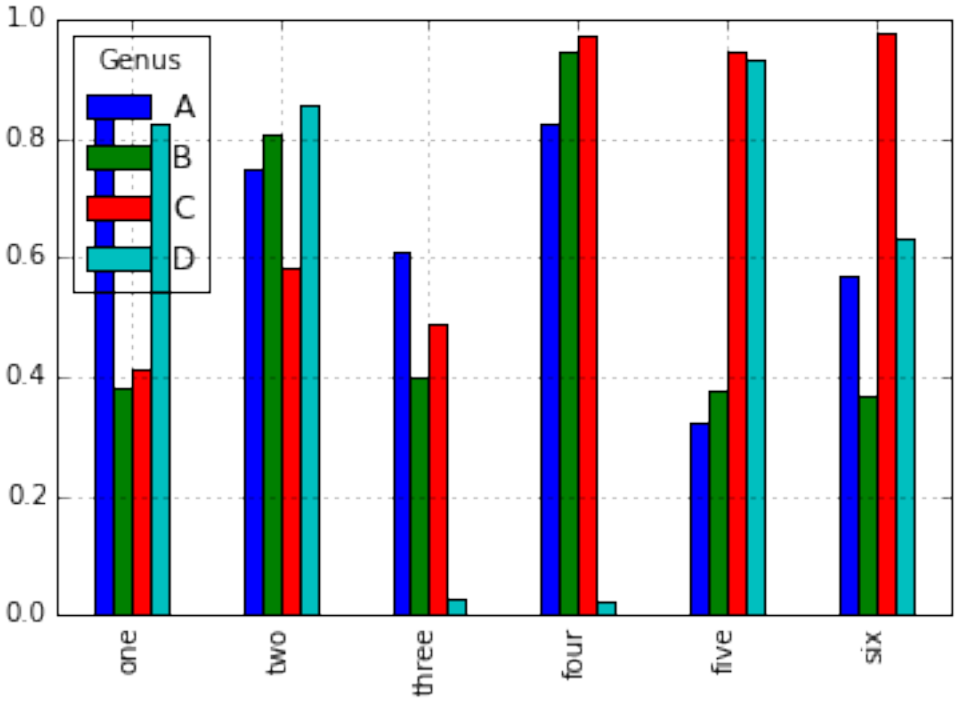
```
In [17]: df = DataFrame(np.random.rand(6, 4),
                        index=['one', 'two', 'three', 'four', 'five', 'six'],
                        columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
         df
```

```
Out[17]: Genus         A         B         C         D
         one    0.836035  0.383135  0.413375  0.823983
         two    0.748730  0.808821  0.582242  0.856516
         three  0.611993  0.401634  0.489481  0.027948
         four   0.823606  0.948357  0.974807  0.023504
         five   0.324714  0.376265  0.946262  0.934007
         six    0.569295  0.366941  0.979173  0.634197
```

We can plot the data in a bar graph just as we would for a `Series` object. Notice that the legend by default is not fixed to any particular location on the plot. This is the `"best"` parameter choice for legend location. You can hide the legend by specifying `legend=False`.

```
In [18]: df.plot(kind='bar')
```

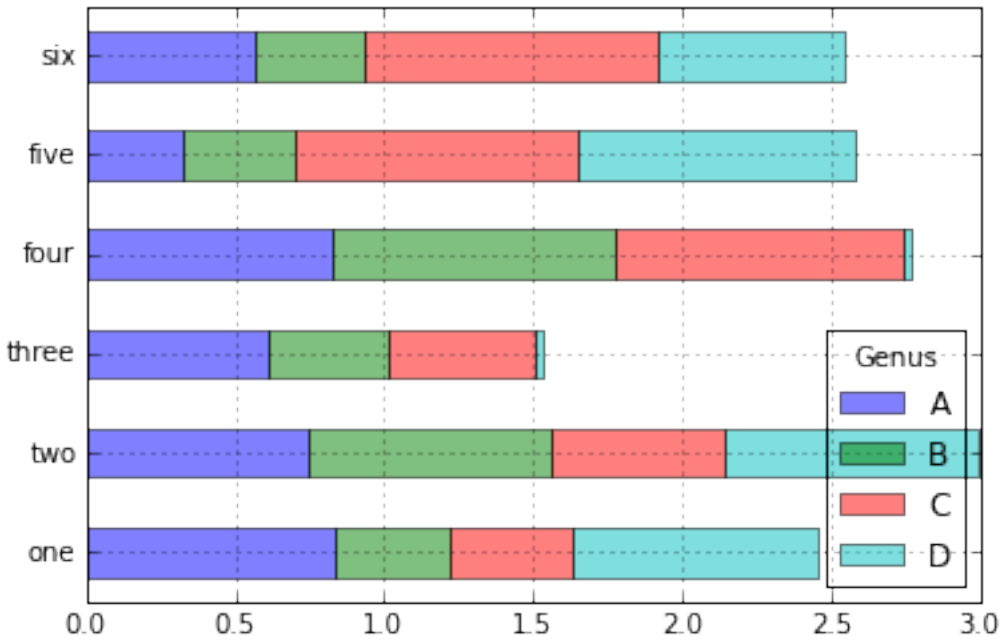
```
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe664491310>
```



Bar plots can also be stacked by using the `stacked` parameter. Notice also that when using `bar` or `barh`, `Pandas` takes care of aligning the data properly to its index label, using the column label for the legend.

```
In [19]: df.plot(kind='barh', stacked=True, alpha=0.5)
```

```
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe664094890>
```



Now, Wes Mckinney has a collection of restaurant tip data in `csv` format that you can download here. We can load up the file and put it directly into a `DataFrame` object using `read_csv` (more on this next week!).

```
In [20]: tips = pd.read_csv('mckinney-files/tips.csv')

         tips.head() # head specifies to display a reasonable amount of output.
```

```
Out[20]:    total_bill   tip     sex smoker  day    time  size
         0       16.99  1.01  Female     No  Sun  Dinner     2
         1       10.34  1.66    Male     No  Sun  Dinner     3
         2       21.01  3.50    Male     No  Sun  Dinner     3
         3       23.68  3.31    Male     No  Sun  Dinner     2
         4       24.59  3.61  Female     No  Sun  Dinner     4
```

We want to cross-tabulate between the day of the week and the size of the party. In other words, we want to count how many parties of one were seated on Friday; how many parties of two; etc, for each day of the week we have data (Friday, Saturday, Sunday, and Thursday). To do this we will use `crosstab`.

First, let's look at the columns formed by `tips`.

```
In [21]: tips.columns
```

```
Out[21]: Index([u'total_bill', u'tip', u'sex', u'smoker', u'day', u'time', u'size'], dtype='object')
```

We can cross-tabulate between the day of the week (Thursday, Friday, Saturday, or Sunday) and the number of guests per party (1-6), using `crosstab`.

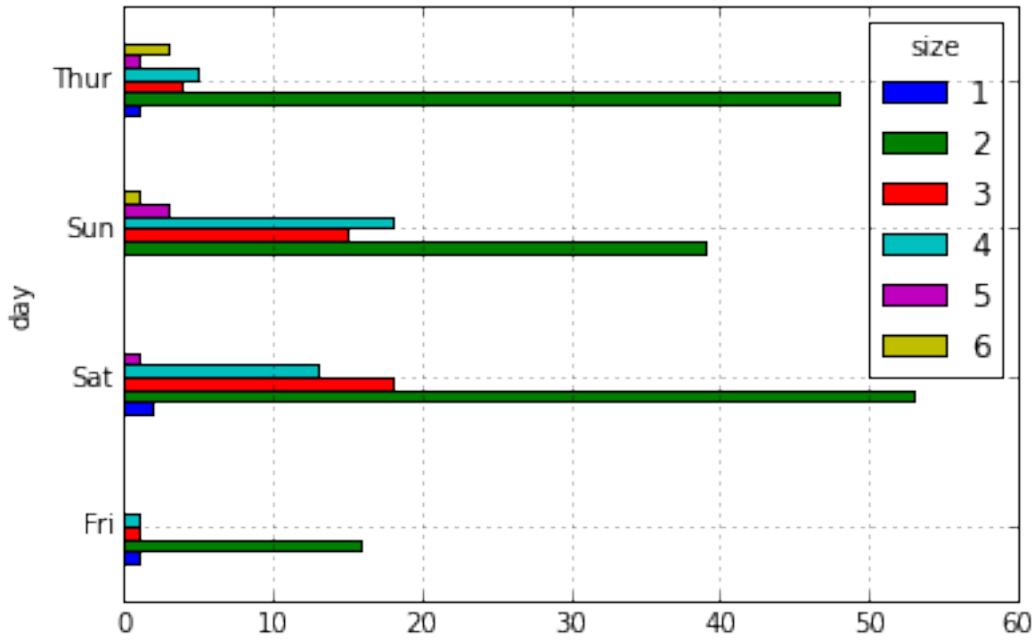
```
In [22]: party_counts = pd.crosstab(tips["day"], tips["size"])

In [23]: party_counts
```

```
Out[23]: size  1   2   3   4  5  6
         day
         Fri   1  16   1   1  0  0
         Sat   2  53  18  13  1  0
         Sun   0  39  15  18  3  1
         Thur  1  48   4   5  1  3
```

Now, we can proceed to the analysis. One type of plot would simply show the breakdown of guests given a day of the week.

```
In [24]: party_counts.plot(kind='barh')
```

```
Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe6651ade90>
```



Not so enlightening, because there is wide variation in the data. Let's restrict our analysis to parties with a size between 2 and 5 (inclusive).

```
In [25]: # Not many 1- and 6-person parties
         party_counts = party_counts.ix[:, 2:5]
```

We can "normalize" the daily data by dividing the values of a particular entry by the sum of the values along that row. Notice that we use `astype(float)` to make sure that no integer division problems are encountered, and we specify `axis=0` to say that we are normalizing along the day, not the party.

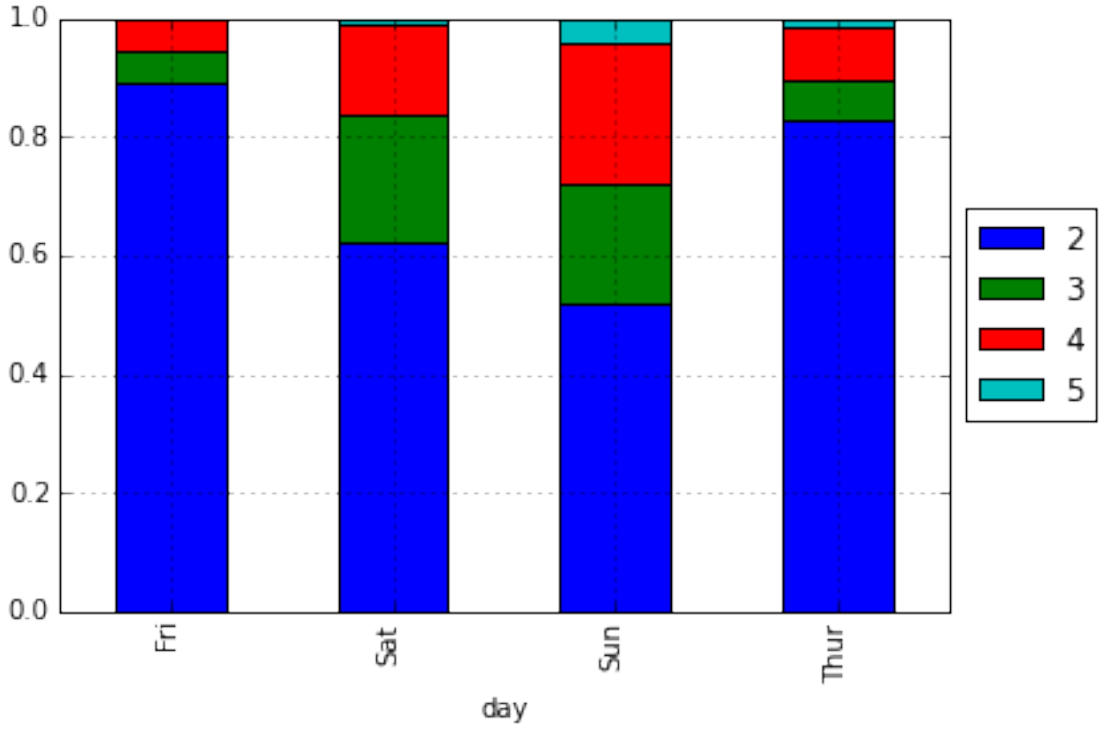
```
In [26]: # Normalize to sum to 1
         party_pcts = party_counts.div(party_counts.sum(1).astype(float), axis=0)
         party_pcts
```

```
Out[26]: size         2         3         4         5
         day
         Fri   0.888889  0.055556  0.055556  0.000000
         Sat   0.623529  0.211765  0.152941  0.011765
         Sun   0.520000  0.200000  0.240000  0.040000
         Thur  0.827586  0.068966  0.086207  0.017241
```

Given this new percentage data, it might make more sense to stack the bars so we can see how the distribution changes from day to day.

```
In [27]: party_pcts.plot(kind='bar', stacked=True)

         plt.legend(loc='center left', bbox_to_anchor=(1.0, 0.5))
```

```
Out[27]: <matplotlib.legend.Legend at 0x7fe664ef7990>
```



Parties seem to get much larger on weekends, while couples dominate during weekdays. Not bad for a short analysis!

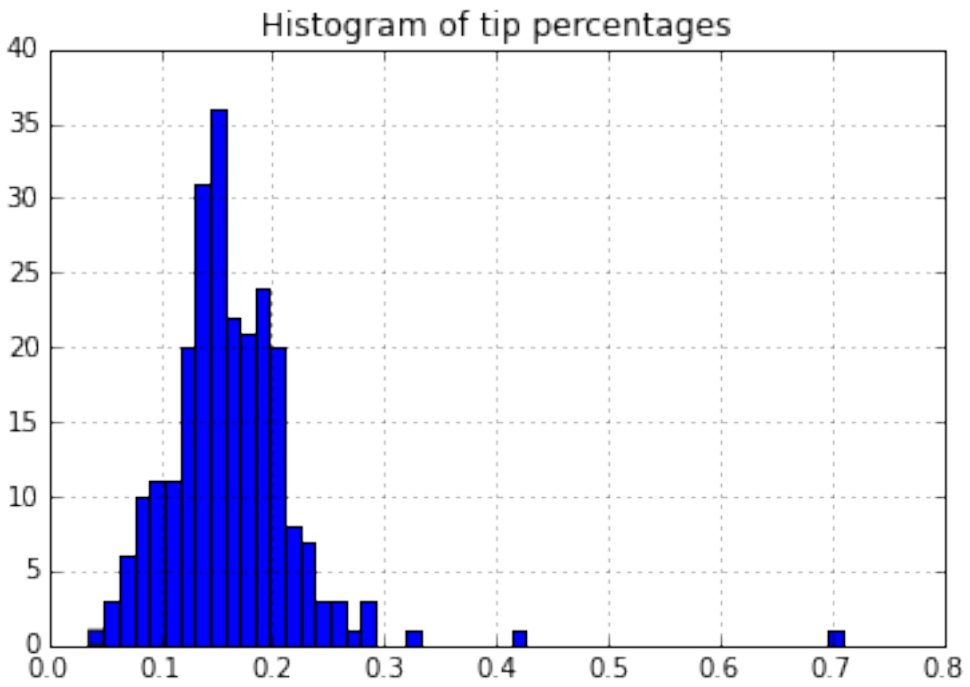
4.2.3 Histograms and density plots

Of course, one might also ask what the distribution of tip percentages a server can expect to see at a given night. `Pandas` helps us answer this with the easy integration of histograms and density plots.

First, let's calculate tip percentages. Luckily, the tip data and the total bill data are already given, so adding a new column is simple.

```
In [28]: plt.figure()

         tips['tip_pct'] = tips['tip'] / tips['total_bill']
         tips['tip_pct'].hist(bins=50)

         plt.title("Histogram of tip percentages")
```

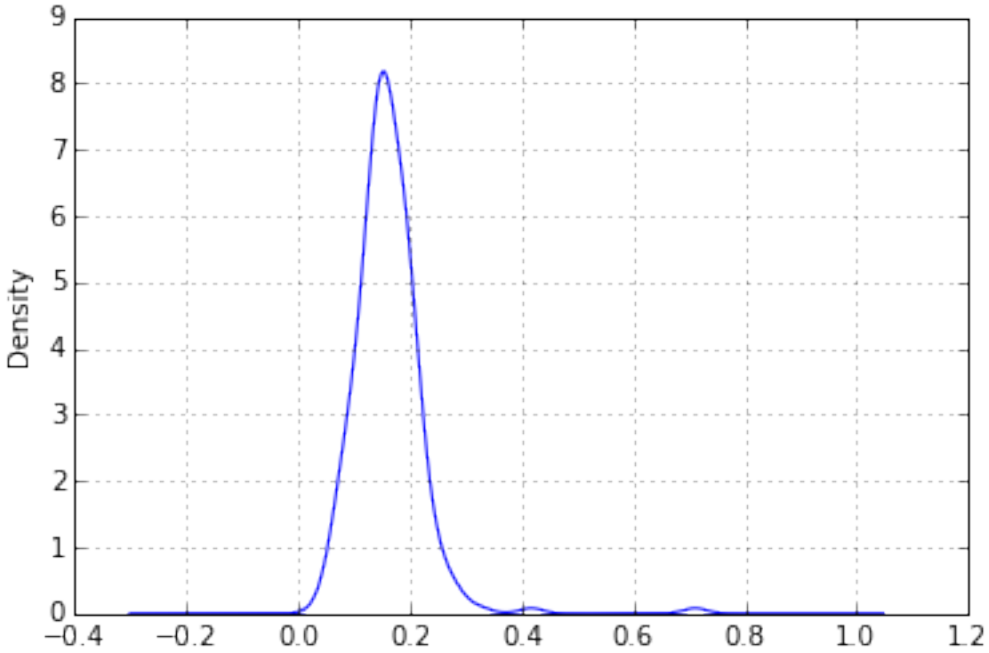
```
Out[28]: <matplotlib.text.Text at 0x7fe664f8ccd0>
```



Looks like the average is right about 15%. Not too shocking. Perhaps instead of a histogram of bins, you want to show a smooth distribution of density. To do so, we can simply choose the `kde` style of plot.

```
In [29]: tips['tip_pct'].plot(kind='kde')
```

```
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe664ae2f90>
```



Quiz: Are the above two plots `Series` plots or `DataFrame` plots?

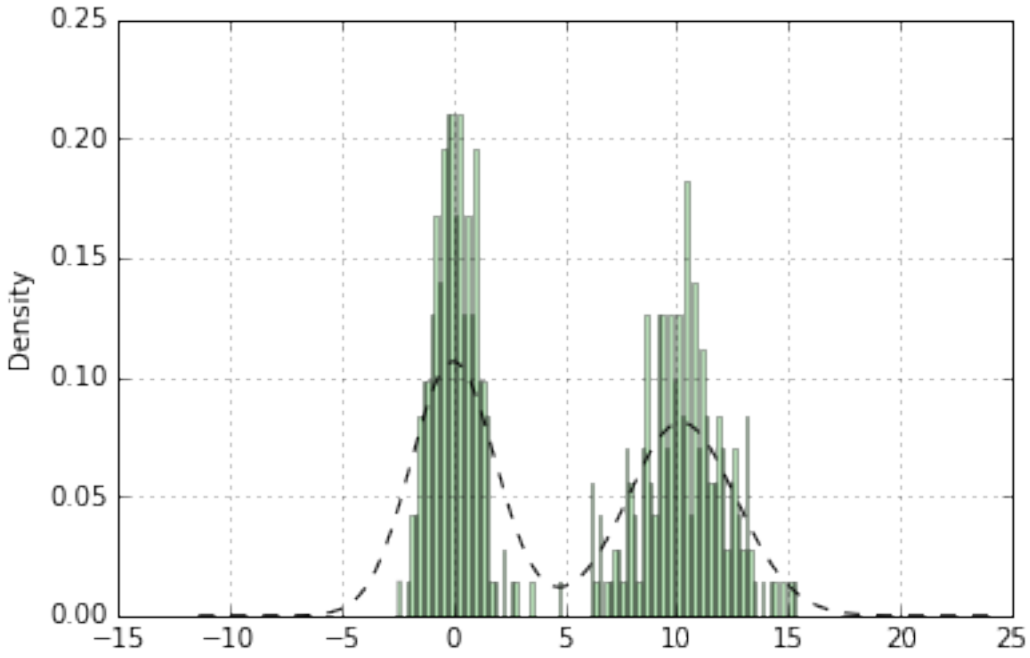
You can actually plot histograms and density plots together. Consider the following random data samples.

```
In [30]: comp1 = np.random.normal(0, 1, size=200)  # N(0, 1)
         comp2 = np.random.normal(10, 2, size=200)  # N(10, 4)
```

By having one cell plot both a histogram and a kernel density estimate plot, we can overlay the two of them together to form a solid understanding of the distribution of data in the set.

```
In [31]: values = Series(np.concatenate([comp1, comp2]))
         values.hist(bins=100, alpha=0.3, color='g', normed=True)
         values.plot(kind='kde', style='k--')
```

```
Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe65c3960d0>
```



This allows us to provide a considerable amount of information compactly into one figure. We can also do it without losing the essence of the visualization.

4.2.4 Scatter plots

When infering the relationship between two series of data, the scatter plot can provide significant assistance to visualize correlation. Consider the following economic data, which you can download as a `csv` file from here.

```
In [32]: macro = pd.read_csv('mckinney-files/macrodata.csv')
         macro.set_index(['year','quarter']).tail()
```

```
Out[32]:                realgdp  realcons   realinv  realgovt  realdpi      cpi  \
         year quarter
         2008 3        13324.600    9267.7  1990.693   991.551   9838.3  216.889
              4        13141.920    9195.3  1857.661  1007.273   9920.4  212.174
```

```
2009 1        12925.410    9209.2  1558.494   996.287   9926.4  212.671
     2        12901.504    9189.0  1456.678  1023.528  10077.5  214.469
     3        12990.341    9256.0  1486.398  1044.088  10040.6  216.385

                   m1  tbilrate  unemp      pop  infl  realint
year quarter
2008 3        1474.7      1.17    6.0  305.270 -3.16     4.33
     4        1576.5      0.12    6.9  305.952 -8.79     8.91
2009 1        1592.8      0.22    8.1  306.547  0.94    -0.71
     2        1653.6      0.18    9.2  307.226  3.37    -3.19
     3        1673.9      0.12    9.6  308.013  3.56    -3.44
```

This is a macroeconomic dataset containing the following metrics: * real gross domestic product * real aggregate consumption * real investment * real government investment * real disposable income * consumer prices * M1 money stock * Treasury bill 1-month yields * unemployment rate * population * inflation * real interest rates

It's a fair bet that this is more information than we want to process at the moment, so we can define a new `DataFrame` considering only the essence of the data we need in this example.

Wes McKinney then takes the data and applies transformations to make the visualization easier.

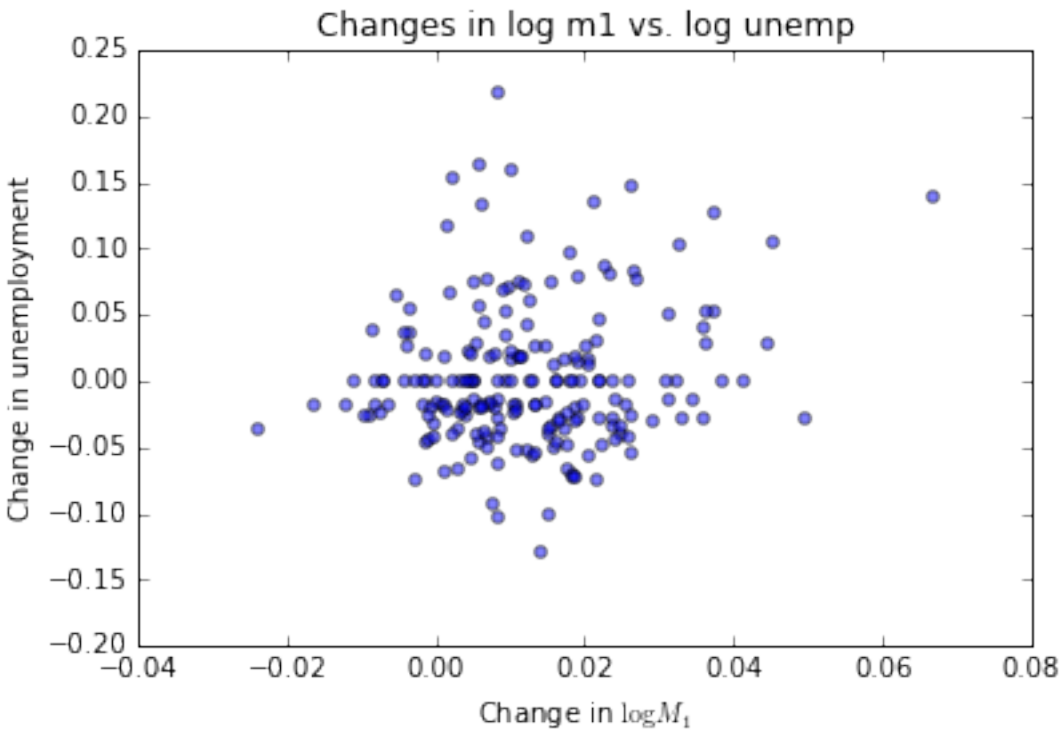
```
In [33]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]
         trans_data = np.log(data).diff().dropna()
         trans_data[-5:]
```

```
Out[33]:           cpi        m1  tbilrate     unemp
         198 -0.007904  0.045361 -0.396881  0.105361
         199 -0.021979  0.066753 -2.277267  0.139762
         200  0.002340  0.010286  0.606136  0.160343
         201  0.008419  0.037461 -0.200671  0.127339
         202  0.008894  0.012202 -0.405465  0.042560
```

How does the change in the size of M1 correspond to changes in the unemployment rate? Let's find out!

```
In [34]: plt.scatter(trans_data['m1'], trans_data['unemp'], alpha=0.5)
         plt.xlabel("Change in $\log M_1$")
         plt.ylabel("Change in unemployment")
         plt.title('Changes in log %s vs. log %s' % ('m1', 'unemp'))
```

```
Out[34]: <matplotlib.text.Text at 0x7fe65bfb2750>
```



It looks like increases in the money supply may have a positive effect on the unemployment rate. Although, it is difficult to say exactly how (we might need a model to infer anything more). Certainly, unemployment seems to be decreasing when the money supply shrinks, according to the data.

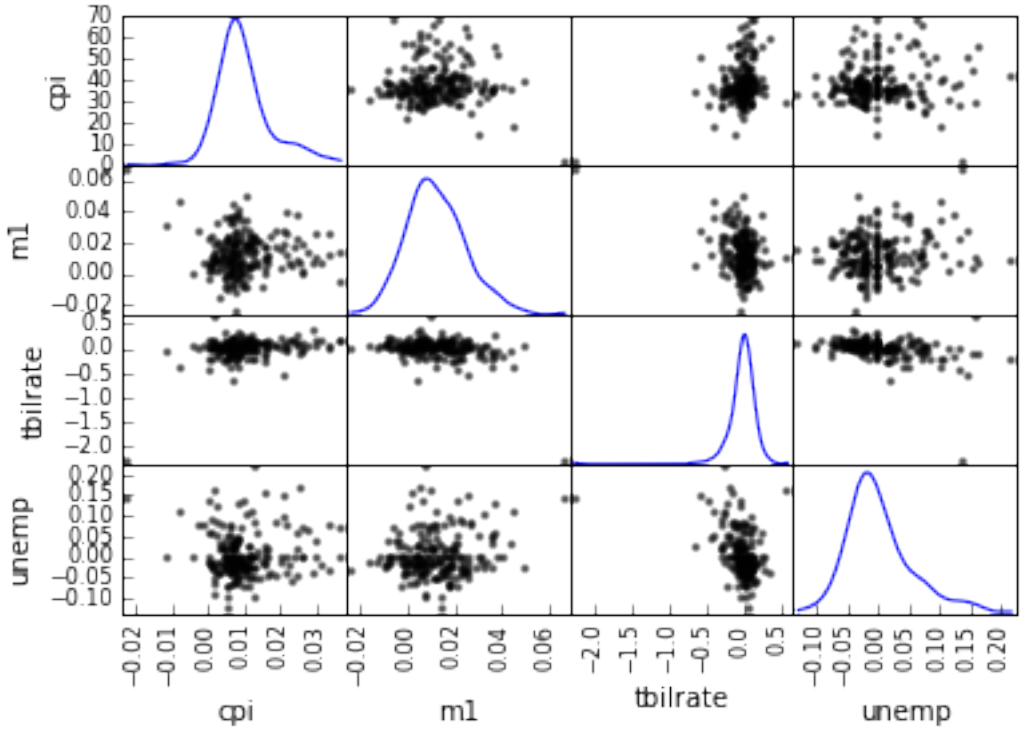
Suppose you have a new dataset and you have no idea how the various series are related. One quick approach to get a feel for the relationships, which you can later expand upon in a more thorough analysis, is the scatter matrix. Given n series of data, `scatter_matrix` produces an $n \times n$ matrix of scatter plots corresponding to pairs of data.

The question is what to do on the main diagonal; a scatter plot of a data series with itself is quite uninteresting. Instead, the default behavior is to produce a histogram of the data series, but you can specify this to be a `kde` plot using the `diag` optional parameter.

```
In [35]: pd.scatter_matrix(trans_data, diagonal='kde', color='k')
```

```
Out[35]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fe65c01ad50>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65bf280d0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65be3cbd0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65bdac350>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x7fe65bd2f150>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65bd5a310>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65bc94d90>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65bb99fd0>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x7fe65bb7bf90>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65bb0c210>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe665a00f50>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65ba2ced0>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x7fe65b9b1b10>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x7fe65b921190>,
```

```
<matplotlib.axes._subplots.AxesSubplot object at 0x7fe65b898e90>,
<matplotlib.axes._subplots.AxesSubplot object at 0x7fe65b873a10>]], dtype=object)
```



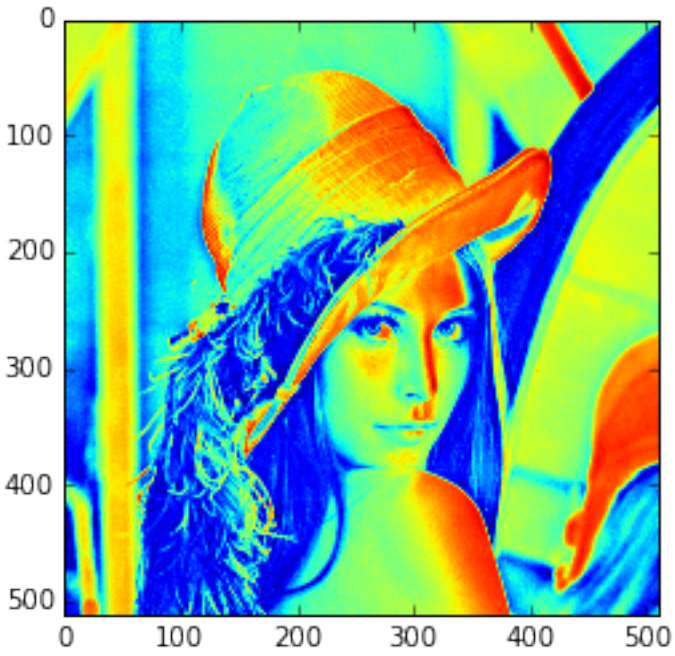
5 Image Processing

In the module we will start an application of `scipy` and `numpy` in order to manipulate images. For further resources we are using ideas and from `http://scipy-lectures.github.io/advanced/image_processing/`.

5.1 Displaying Files

First we need to import the `scipy` and `numpy` into our file. After doing this we want to write an array into a file.

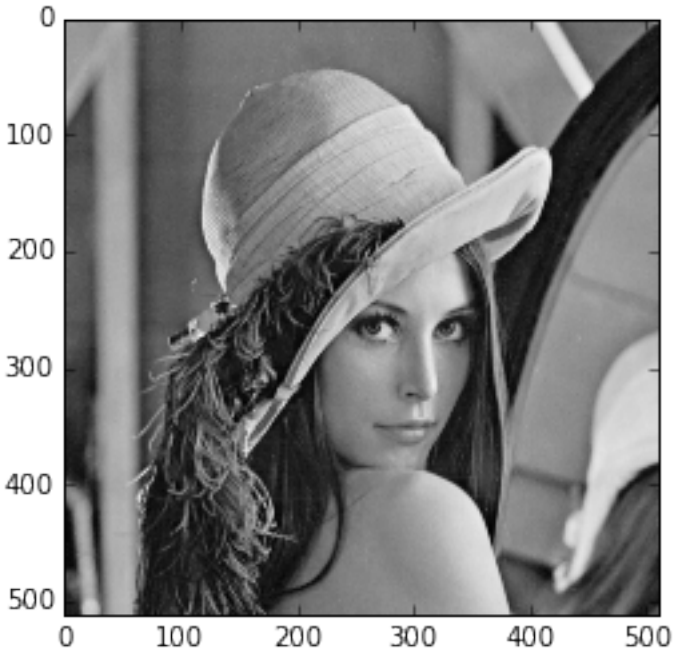
```
In [36]: %matplotlib inline
         from scipy import misc

In [37]: l = misc.lena()
         misc.imsave('lena.png', l) # uses the Image module (PIL)

In [38]: import matplotlib.pyplot as plt
         plt.imshow(l)

Out[38]: <matplotlib.image.AxesImage at 0x7fe65b2bc890>
```



We can also change the color of our image to reflect the original greyscale.

```
In [39]: plt.imshow(l, cmap=plt.cm.gray)
```

```
Out[39]: <matplotlib.image.AxesImage at 0x7fe65b1aef90>
```



We can increase the contrast by changing the mininimum and maximum values.

```
In [40]: plt.imshow(l, cmap=plt.cm.gray, vmin=100, vmax=200)
         plt.axis('off') # Remove axes and ticks

Out[40]: (-0.5, 511.5, 511.5, -0.5)
```

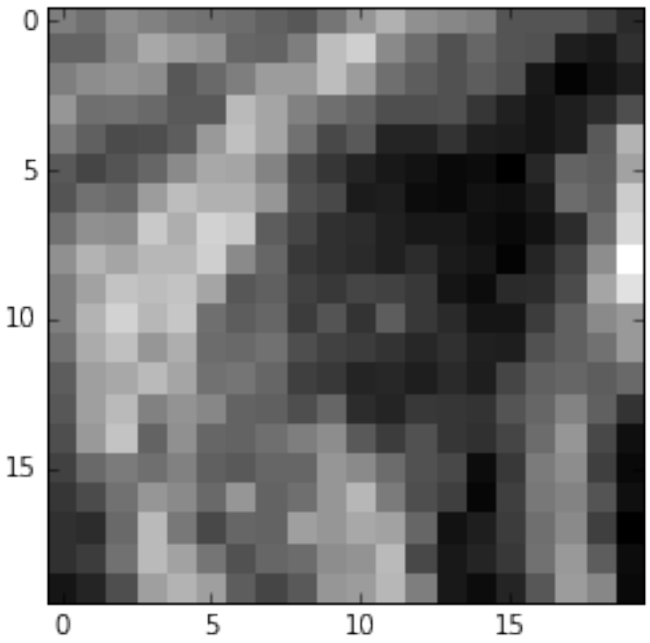
An interesting image processing technique is drawing contour lines. We can do this using `plt.contour`.

```
In [41]: plt.imshow(l, cmap=plt.cm.gray,vmin=100, vmax=200)
         plt.contour(l, [60, 150])
         plt.axis('off')

Out[41]: (-0.5, 511.5, 511.5, -0.5)
```

We can inspect individual elements for intensity variation using `interpolation='nearest'`.

```
In [42]: plt.imshow(l[200:220, 200:220], cmap=plt.cm.gray)
         plt.imshow(l[200:220, 200:220], cmap=plt.cm.gray,
                    interpolation='nearest')
```

```
Out[42]: <matplotlib.image.AxesImage at 0x7fe6599603d0>
```



5.2 Basic Image Manipulations

Images are arrays. Consequently, we can use array manipulations that we used from `numpy`.

```
In [43]: import scipy
         import numpy as np

In [44]: lena = scipy.misc.lena()
         lena[10:13, 20:23]
         lena[100:120] = 255

         lx, ly = lena.shape
         X, Y = np.ogrid[0:lx, 0:ly]
         mask = (X - lx/2)**2 + (Y - ly/2)**2 > lx*ly/4
         lena[mask] = 0
         lena[range(400), range(400)] = 255

         plt.figure(figsize=(3, 3))
         plt.axes([0, 0, 1, 1])
         plt.imshow(lena, cmap=plt.cm.gray)
         plt.axis('off')

Out[44]: (-0.5, 511.5, 511.5, -0.5)
```

5.3 Geometric Transformations

We can easily rotate and flip the image using the `numpy` library.

```
In [45]: from scipy import ndimage

         lena = scipy.misc.lena()
```

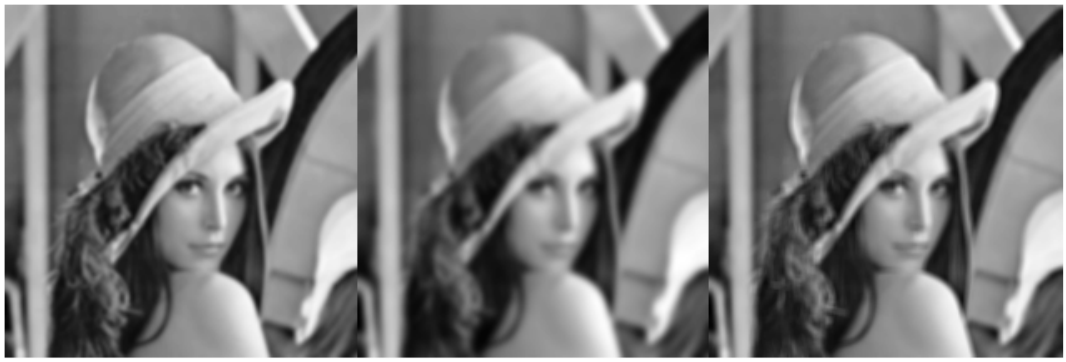
```
lx, ly = lena.shape

# Cropping

crop_lena = lena[lx/4:-lx/4, ly/4:-ly/4]
# up <-> down flip
flip_ud_lena = np.flipud(lena)
# rotation
rotate_lena = ndimage.rotate(lena, 45)
rotate_lena_noreshape = ndimage.rotate(lena, 45, reshape=False)

plt.figure(figsize=(12.5, 2.5))


plt.subplot(151)
plt.imshow(lena, cmap=plt.cm.gray)
plt.axis('off')
plt.subplot(152)
plt.imshow(crop_lena, cmap=plt.cm.gray)
plt.axis('off')
plt.subplot(153)
plt.imshow(flip_ud_lena, cmap=plt.cm.gray)
plt.axis('off')
plt.subplot(154)
plt.imshow(rotate_lena, cmap=plt.cm.gray)
plt.axis('off')
plt.subplot(155)
plt.imshow(rotate_lena_noreshape, cmap=plt.cm.gray)
plt.axis('off')

plt.subplots_adjust(wspace=0.02, hspace=0.3, top=1, bottom=0.1, left=0,
                    right=1)
```

5.4 Image Filtering

We can filter images by replacing the value of the pixels by a function of adjacent pixels. In the example below we use two different filters. The Gaussian filter sets the value of a pixel to the weighted average of the value of neighboring pixels, where nearby pixels have greater weights. The uniform filter is simply the average value of the pixels a set distance away.

In [46]:
```
lena = scipy.misc.lena()
blurred_lena = ndimage.gaussian_filter(lena, sigma=3)
very_blurred = ndimage.gaussian_filter(lena, sigma=5)
```

```
local_mean = ndimage.uniform_filter(lena, size=11)


plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.imshow(blurred_lena, cmap=plt.cm.gray)
plt.axis('off')
plt.subplot(132)
plt.imshow(very_blurred, cmap=plt.cm.gray)
plt.axis('off')
plt.subplot(133)
plt.imshow(local_mean, cmap=plt.cm.gray)
plt.axis('off')

plt.subplots_adjust(wspace=0, hspace=0., top=0.99, bottom=0.01,
                    left=0.01, right=0.99)
```



5.5 Image Sharpening

We can also sharpen a blurred image. The following shows the original image followed by a blurred image and a resharpened image.

In [47]:
```
l = scipy.misc.lena()
blurred_l = ndimage.gaussian_filter(l, 3)

filter_blurred_l = ndimage.gaussian_filter(blurred_l, 1)

alpha = 30
sharpened = blurred_l + alpha * (blurred_l - filter_blurred_l)

plt.figure(figsize=(12, 4))

plt.subplot(131)
plt.imshow(l, cmap=plt.cm.gray)
plt.axis('off')
plt.subplot(132)
plt.imshow(blurred_l, cmap=plt.cm.gray)
plt.axis('off')
```

```
plt.subplot(133)
plt.imshow(sharpened, cmap=plt.cm.gray)
plt.axis('off')
```

```
Out[47]: (-0.5, 511.5, 511.5, -0.5)
```

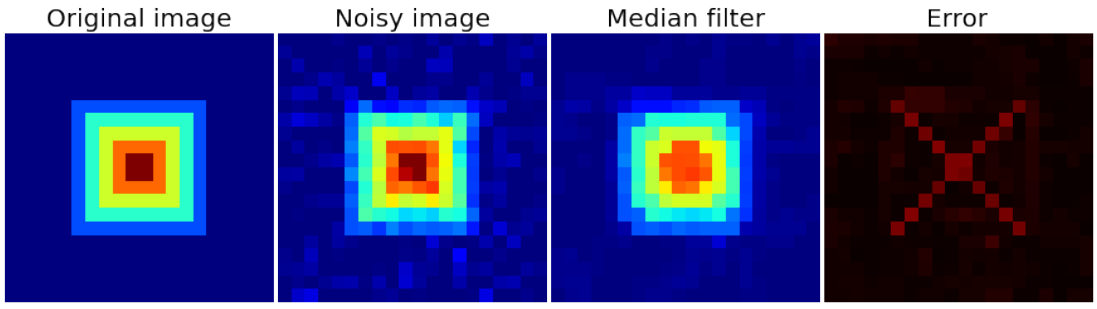
5.6 Denoising

Applying the filters we learned to help us blur and sharpen images allow us to denoise an image. However, these filters are not without problems. The Gaussian filter smoothes out the noise, but it also smoothes out the edges of the picture. A median picture smoothes the noise, but it preserves the edges better than the Gaussian filter.

```
In [48]: l = scipy.misc.lena()
         l = l[230:290, 220:320]

         noisy = l + 0.4*l.std()*np.random.random(l.shape)

         gauss_denoised = ndimage.gaussian_filter(noisy, 2)
         med_denoised = ndimage.median_filter(noisy, 3)


         plt.figure(figsize=(12,2.8))

         plt.subplot(131)
         plt.imshow(noisy, cmap=plt.cm.gray, vmin=40, vmax=220)
         plt.axis('off')
         plt.title('noisy', fontsize=20)
         plt.subplot(132)
         plt.imshow(gauss_denoised, cmap=plt.cm.gray, vmin=40, vmax=220)
         plt.axis('off')
         plt.title('Gaussian filter', fontsize=20)
         plt.subplot(133)
         plt.imshow(med_denoised, cmap=plt.cm.gray, vmin=40, vmax=220)
         plt.axis('off')
         plt.title('Median filter', fontsize=20)

         plt.subplots_adjust(wspace=0.02, hspace=0.02, top=0.9, bottom=0, left=0,
                             right=1)
```

The median filter is better when working with straight edges (low-curviture images).

In [49]:
```
im = np.zeros((20, 20))
im[5:-5, 5:-5] = 1
im = ndimage.distance_transform_bf(im)
im_noise = im + 0.2*np.random.randn(*im.shape)

im_med = ndimage.median_filter(im_noise, 3)

plt.figure(figsize=(12, 5))

plt.subplot(141)
plt.imshow(im, interpolation='nearest')
plt.axis('off')
plt.title('Original image', fontsize=20)
plt.subplot(142)
plt.imshow(im_noise, interpolation='nearest', vmin=0, vmax=5)
plt.axis('off')
plt.title('Noisy image', fontsize=20)
plt.subplot(143)
plt.imshow(im_med, interpolation='nearest', vmin=0, vmax=5)
plt.axis('off')
plt.title('Median filter', fontsize=20)
plt.subplot(144)
plt.imshow(np.abs(im - im_med), cmap=plt.cm.hot, vmin=0, vmax=5, interpolation='nearest')
plt.axis('off')
plt.title('Error', fontsize=20)


plt.subplots_adjust(wspace=0.02, hspace=0.02, top=0.9, bottom=0, left=0, right=1)
```



5.6.1 Try It!

1. Try adding noise to the image `Lena`. Once you have a noisy image try using a median and Gaussian filter to smooth the image.
2. Create an error chart to measure the error between the two techniques.
3. Try using a new filter like `ndimage.maximum_filter`, and `ndimage.percentile_filter` on the image of concentric squares.
4. Try using a non-rank filter like `scipy.signal.wiener`.