

Agile Tools for Real-World Data

Python for Data Analysis



O'REILLY®

Wes McKinney

Python for Data Analysis

Looking for complete instructions on manipulating, processing, cleaning, and crunching structured data in Python? This hands-on book is packed with practical case studies that show you how to effectively solve a broad set of data analysis problems, using several Python libraries—including NumPy, pandas, matplotlib, and IPython.

Written by Wes McKinney, the main author of the pandas library, *Python for Data Analysis* also serves as a practical, modern introduction to scientific computing in Python for data-intensive applications. It's ideal for analysts new to Python and for Python programmers new to scientific computing.

- Use the IPython interactive shell as your primary development environment
- Learn basic and advanced NumPy (Numerical Python) features
- Get started with data analysis tools in the pandas library
- Use high-performance tools to load, clean, transform, merge, and reshape data
- Create scatter plots and static or interactive visualizations with matplotlib
- Apply the pandas groupby facility to slice, dice, and summarize datasets
- Work with time series data in many different forms
- Learn how to solve problems in web analytics, social sciences, finance, and economics, through detailed examples

Wes McKinney is the main author of pandas, the popular open source Python library for data analysis. Prior to starting Lambda Foundry, an enterprise data analysis company, he worked as a quantitative analyst at AQR Capital Management. Wes is an active speaker and participant in the Python and open source communities.

US \$39.99

CAN \$41.99

ISBN: 978-1-449-31979-3



5 3 9 9 9
9 781449 319793



“The scientific and data analysis communities have been waiting for this book for years: loaded with concrete practical advice, yet full of insight about how all the pieces fit together. It should become a canonical reference for technical computing in Python for years to come.”

—Fernando Pérez

research scientist at UC Berkeley,
one of the originators of IPython

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

Python for Data Analysis

Wes McKinney

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Plotting and Visualization

Making plots and static or interactive visualizations is one of the most important tasks in data analysis. It may be a part of the exploratory process; for example, helping identify outliers, needed data transformations, or coming up with ideas for models. For others, building an interactive visualization for the web using a toolkit like d3.js (<http://d3js.org/>) may be the end goal. Python has many visualization tools (see the end of this chapter), but I'll be mainly focused on matplotlib (<http://matplotlib.sourceforge.net>).

matplotlib is a (primarily 2D) desktop plotting package designed for creating publication-quality plots. The project was started by John Hunter in 2002 to enable a MATLAB-like plotting interface in Python. He, Fernando Pérez (of IPython), and others have collaborated for many years since then to make IPython combined with matplotlib a very functional and productive environment for scientific computing. When used in tandem with a GUI toolkit (for example, within IPython), matplotlib has interactive features like zooming and panning. It supports many different GUI backends on all operating systems and additionally can export graphics to all of the common vector and raster graphics formats: PDF, SVG, JPG, PNG, BMP, GIF, etc. I have used it to produce almost all of the graphics outside of diagrams in this book.

matplotlib has a number of add-on toolkits, such as `mplot3d` for 3D plots and `basemap` for mapping and projections. I will give an example using `basemap` to plot data on a map and to read *shapefiles* at the end of the chapter.

To follow along with the code examples in the chapter, make sure you have started IPython in Pylab mode (`ipython --pylab`) or enabled GUI event loop integration with the `%gui` magic.

A Brief matplotlib API Primer

There are several ways to interact with matplotlib. The most common is through *pylab mode* in IPython by running `ipython --pylab`. This launches IPython configured to be able to support the matplotlib GUI backend of your choice (Tk, wxPython, PyQt, Mac

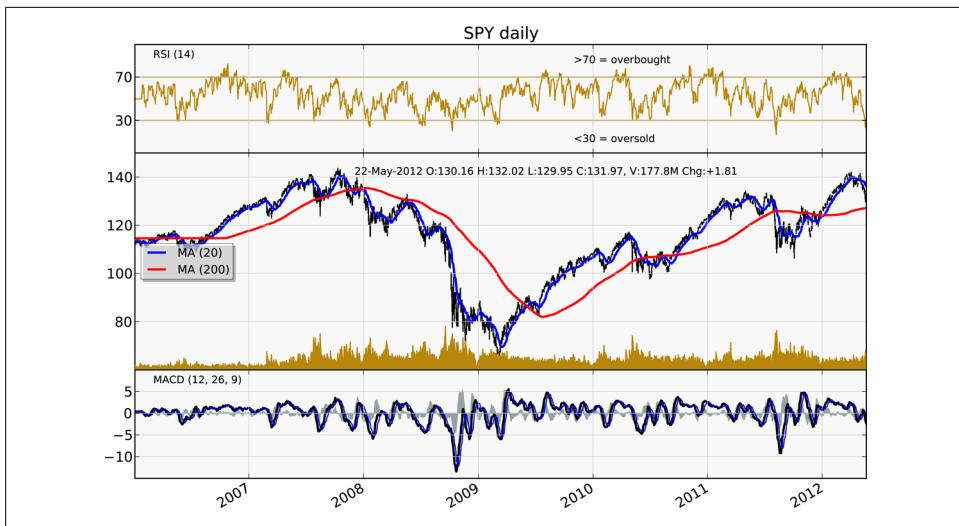


Figure 8-1. A more complex matplotlib financial plot

OS X native, GTK). For most users, the default backend will be sufficient. Pylab mode also imports a large set of modules and functions into IPython to provide a more MATLAB-like interface. You can test that everything is working by making a simple plot:

```
plot(np.arange(10))
```

If everything is set up right, a new window should pop up with a line plot. You can close it by using the mouse or entering `close()`. Matplotlib API functions like `plot` and `close` are all in the `matplotlib.pyplot` module, which is typically imported by convention as:

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

While the pandas plotting functions described later deal with many of the mundane details of making plots, should you wish to customize them beyond the function options provided you will need to learn a bit about the matplotlib API.



There is not enough room in the book to give a comprehensive treatment to the breadth and depth of functionality in matplotlib. It should be enough to teach you the ropes to get up and running. The matplotlib gallery and documentation are the best resource for becoming a plotting guru and using advanced features.

Figures and Subplots

Plots in matplotlib reside within a `Figure` object. You can create a new figure with `plt.figure`:

```
In [13]: fig = plt.figure()
```

If you are in pylab mode in IPython, a new empty window should pop up. `plt.figure` has a number of options, notably `figsize` will guarantee the figure has a certain size and aspect ratio if saved to disk. Figures in matplotlib also support a numbering scheme (for example, `plt.figure(2)`) that mimics MATLAB. You can get a reference to the active figure using `plt.gcf()`.

You can't make a plot with a blank figure. You have to create one or more `subplots` using `add_subplot`:

```
In [14]: ax1 = fig.add_subplot(2, 2, 1)
```

This means that the figure should be 2×2 , and we're selecting the first of 4 subplots (numbered from 1). If you create the next two subplots, you'll end up with a figure that looks like [Figure 8-2](#).

```
In [15]: ax2 = fig.add_subplot(2, 2, 2)
```

```
In [16]: ax3 = fig.add_subplot(2, 2, 3)
```

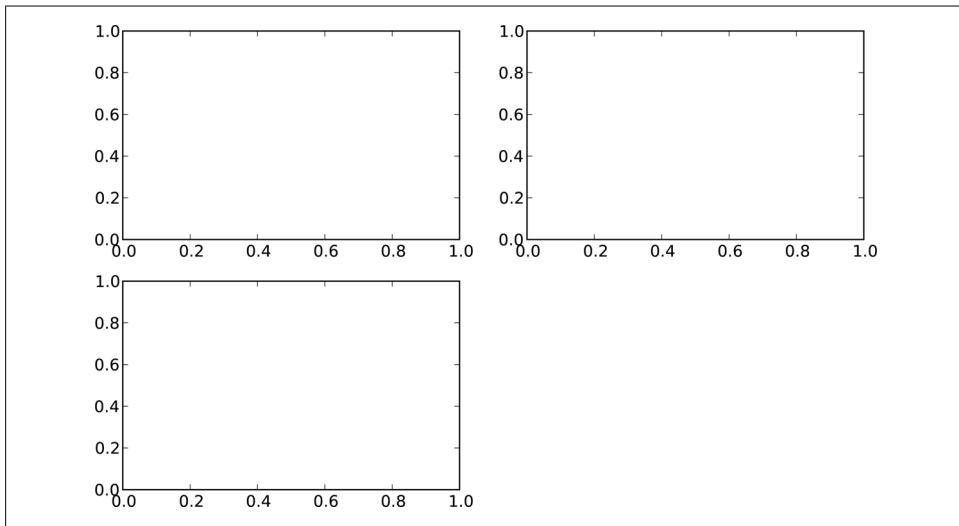


Figure 8-2. An empty matplotlib Figure with 3 subplots

When you issue a plotting command like `plt.plot([1.5, 3.5, -2, 1.6])`, matplotlib draws on the last figure and subplot used (creating one if necessary), thus hiding the figure and subplot creation. Thus, if we run the following command, you'll get something like [Figure 8-3](#):

```
In [17]: from numpy.random import randn
```

```
In [18]: plt.plot(randn(50).cumsum(), 'k--')
```

The '`k--'`' is a *style* option instructing matplotlib to plot a black dashed line. The objects returned by `fig.add_subplot` above are `AxesSubplot` objects, on which you can directly plot on the other empty subplots by calling each one's instance methods, see [Figure 8-4](#):

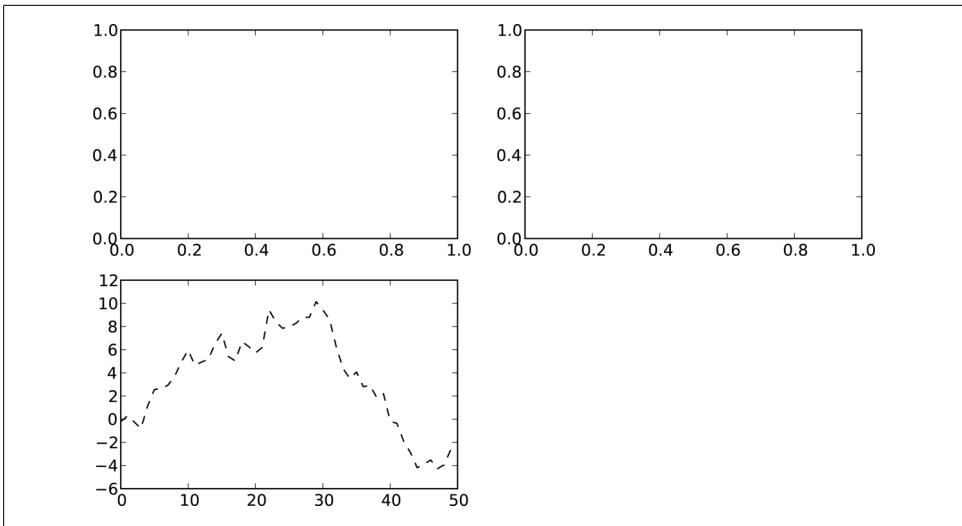


Figure 8-3. Figure after single plot

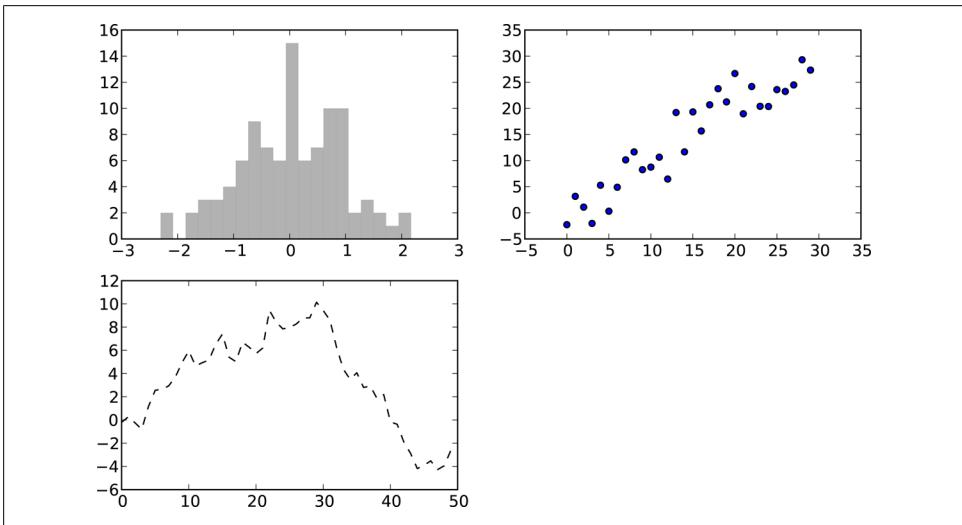


Figure 8-4. Figure after additional plots

```
In [19]: _ = ax1.hist(randn(100), bins=20, color='k', alpha=0.3)
```

```
In [20]: ax2.scatter(np.arange(30), np.arange(30) + 3 * randn(30))
```

You can find a comprehensive catalogue of plot types in the matplotlib documentation.

Since creating a figure with multiple subplots according to a particular layout is such a common task, there is a convenience method, `plt.subplots`, that creates a new figure and returns a NumPy array containing the created subplot objects:

```
In [22]: fig, axes = plt.subplots(2, 3)

In [23]: axes
Out[23]:
array([[Axes(0.125,0.536364;0.227941x0.363636),
       Axes(0.398529,0.536364;0.227941x0.363636),
       Axes(0.672059,0.536364;0.227941x0.363636)],
      [Axes(0.125,0.1;0.227941x0.363636),
       Axes(0.398529,0.1;0.227941x0.363636),
       Axes(0.672059,0.1;0.227941x0.363636)]], dtype=object)
```

This is very useful as the `axes` array can be easily indexed like a two-dimensional array; for example, `axes[0, 1]`. You can also indicate that subplots should have the same X or Y axis using `sharex` and `sharey`, respectively. This is especially useful when comparing data on the same scale; otherwise, matplotlib auto-scales plot limits independently. See [Table 8-1](#) for more on this method.

Table 8-1. pyplot.subplots options

Argument	Description
<code>nrows</code>	Number of rows of subplots
<code>ncols</code>	Number of columns of subplots
<code>sharex</code>	All subplots should use the same X-axis ticks (adjusting the <code>xlim</code> will affect all subplots)
<code>sharey</code>	All subplots should use the same Y-axis ticks (adjusting the <code>ylim</code> will affect all subplots)
<code>subplot_kw</code>	Dict of keywords for creating the
<code>*fig_kw</code>	Additional keywords to subplots are used when creating the figure, such as <code>plt.subplots(2, 2, figsize=(8, 6))</code>

Adjusting the spacing around subplots

By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots. This spacing is all specified relative to the height and width of the plot, so that if you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself. The spacing can be most easily changed using the `subplots_adjust` Figure method, also available as a top-level function:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

`wspace` and `hspace` controls the percent of the figure width and figure height, respectively, to use as spacing between subplots. Here is a small example where I shrink the spacing all the way to zero (see [Figure 8-5](#)):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

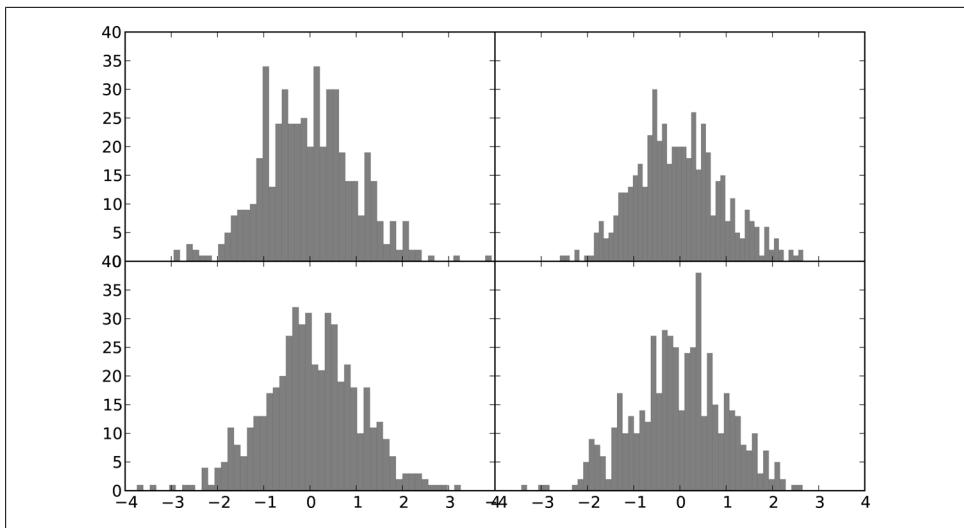


Figure 8-5. Figure with no inter-subplot spacing

You may notice that the axis labels overlap. matplotlib doesn't check whether the labels overlap, so in a case like this you would need to fix the labels yourself by specifying explicit tick locations and tick labels. More on this in the coming sections.

Colors, Markers, and Line Styles

Matplotlib's main `plot` function accepts arrays of X and Y coordinates and optionally a string abbreviation indicating color and line style. For example, to plot `x` versus `y` with green dashes, you would execute:

```
ax.plot(x, y, 'g--')
```

This way of specifying both color and linestyle in a string is provided as a convenience; in practice if you were creating plots programmatically you might prefer not to have to munge strings together to create plots with the desired style. The same plot could also have been expressed more explicitly as:

```
ax.plot(x, y, linestyle='--', color='g')
```

There are a number of color abbreviations provided for commonly-used colors, but any color on the spectrum can be used by specifying its RGB value (for example, '#CECECE'). You can see the full set of linestyles by looking at the docstring for `plot`.

Line plots can additionally have *markers* to highlight the actual data points. Since matplotlib creates a continuous line plot, interpolating between points, it can occasionally be unclear where the points lie. The marker can be part of the style string, which must have color followed by marker type and line style (see [Figure 8-6](#)):

```
In [28]: plt.plot(randn(30).cumsum(), 'ko--')
```

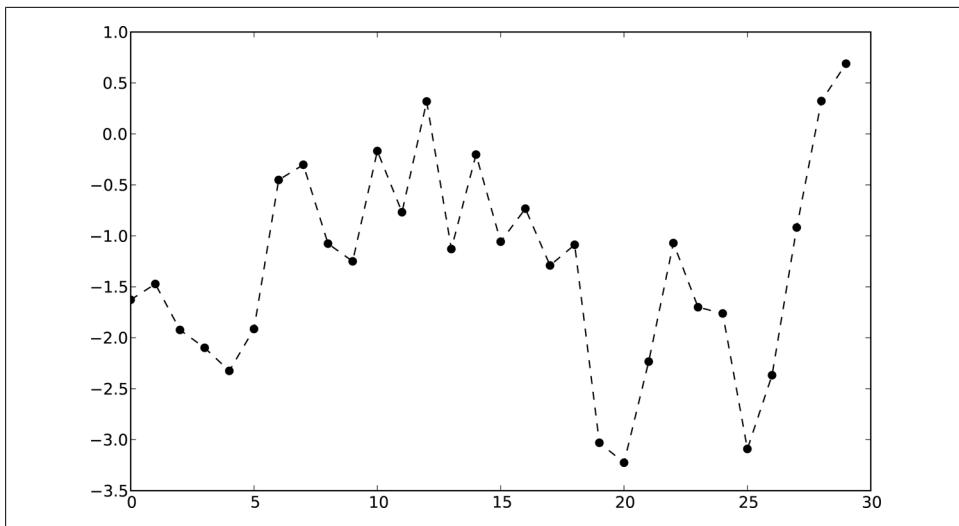


Figure 8-6. Line plot with markers example

This could also have been written more explicitly as:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the `drawstyle` option:

```
In [30]: data = randn(30).cumsum()
```

```
In [31]: plt.plot(data, 'k--', label='Default')
Out[31]: [<matplotlib.lines.Line2D at 0x461cdd0>]
```

```
In [32]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
Out[32]: [<matplotlib.lines.Line2D at 0x461f350>]
```

```
In [33]: plt.legend(loc='best')
```

Ticks, Labels, and Legends

For most kinds of plot decorations, there are two main ways to do things: using the procedural `pyplot` interface (which will be very familiar to MATLAB users) and the more object-oriented native `matplotlib` API.

The `pyplot` interface, designed for interactive use, consists of methods like `xlim`, `xticks`, and `xticklabels`. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways:

- Called with no arguments returns the current parameter value. For example `plt.xlim()` returns the current X axis plotting range

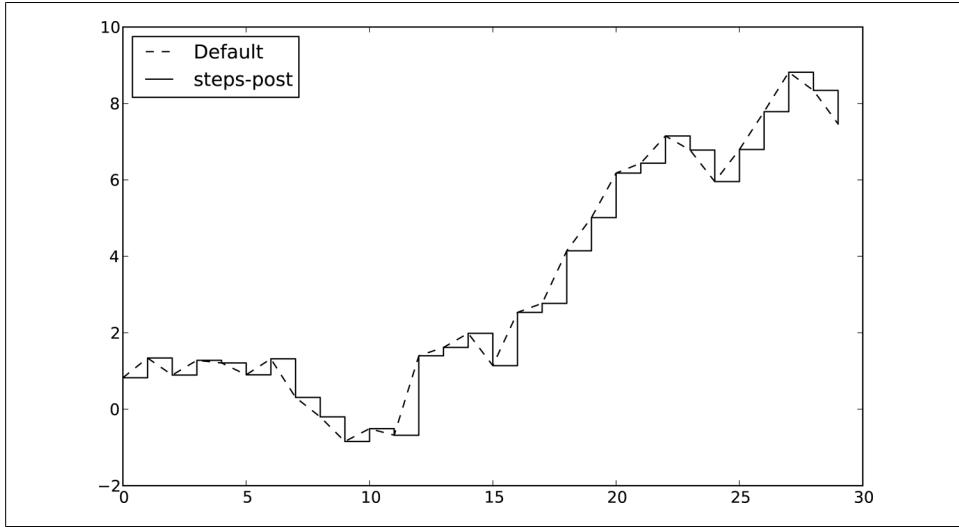


Figure 8-7. Line plot with different drawstyle options

- Called with parameters sets the parameter value. So `plt.xlim([0, 10])`, sets the X axis range to 0 to 10

All such methods act on the active or most recently-created `AxesSubplot`. Each of them corresponds to two methods on the subplot object itself; in the case of `xlim` these are `ax.get_xlim` and `ax.set_xlim`. I prefer to use the subplot instance methods myself in the interest of being explicit (and especially when working with multiple subplots), but you can certainly use whichever you find more convenient.

Setting the title, axis labels, ticks, and ticklabels

To illustrate customizing the axes, I'll create a simple figure and plot of a random walk (see [Figure 8-8](#)):

```
In [34]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
```

```
In [35]: ax.plot(randn(1000).cumsum())
```

To change the X axis ticks, it's easiest to use `set_xticks` and `set_xticklabels`. The former instructs matplotlib where to place the ticks along the data range; by default these locations will also be the labels. But we can set any other values as the labels using `set_xticklabels`:

```
In [36]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])
```

```
In [37]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
...:                                     rotation=30, fontsize='small')
```

Lastly, `set_xlabel` gives a name to the X axis and `set_title` the subplot title:

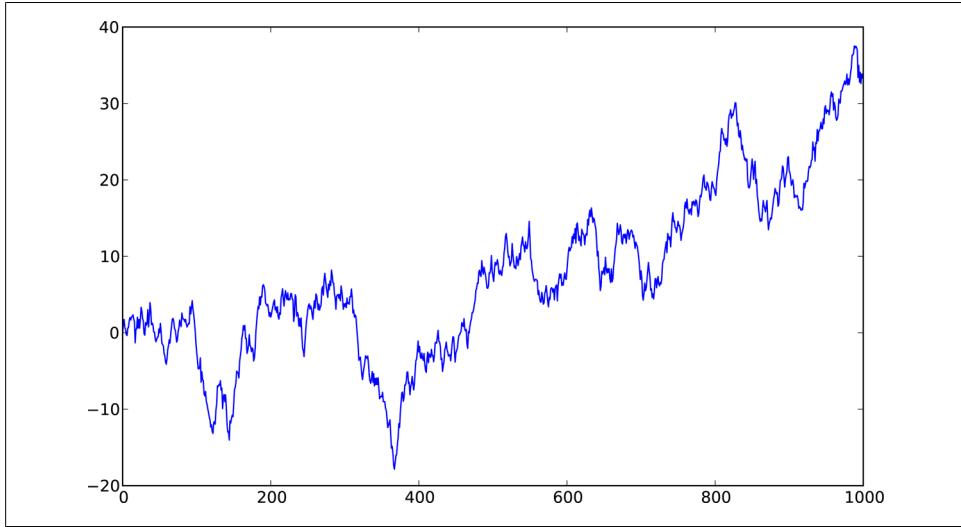


Figure 8-8. Simple plot for illustrating xticks

```
In [38]: ax.set_title('My first matplotlib plot')
Out[38]: <matplotlib.text.Text at 0x7f9190912850>
```

```
In [39]: ax.set_xlabel('Stages')
```

See [Figure 8-9](#) for the resulting figure. Modifying the Y axis consists of the same process, substituting `y` for `x` in the above.

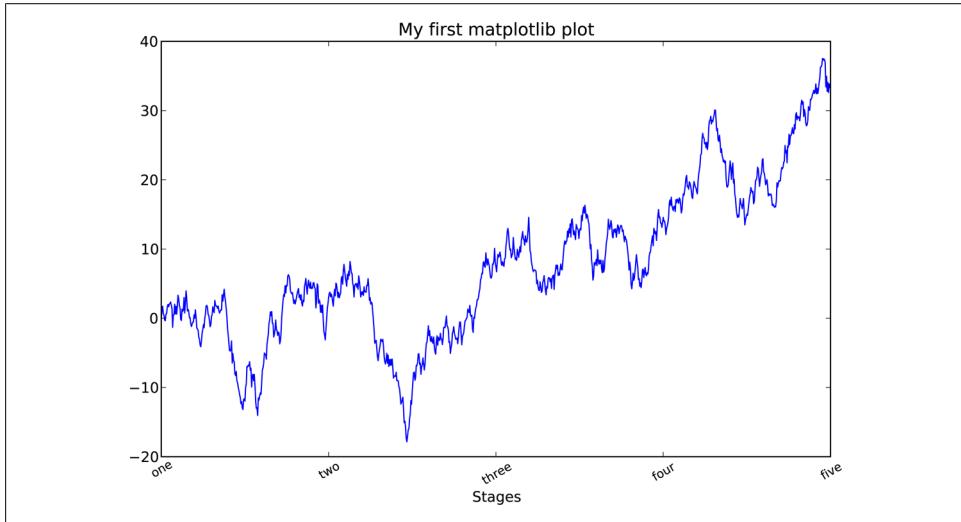


Figure 8-9. Simple plot for illustrating xticks

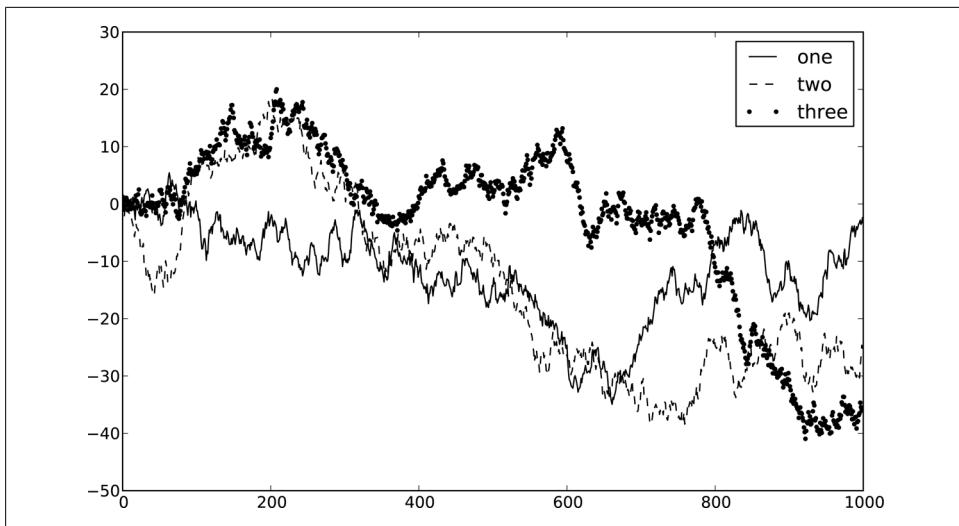


Figure 8-10. Simple plot with 3 lines and legend

Adding legends

Legends are another critical element for identifying plot elements. There are a couple of ways to add one. The easiest is to pass the `label` argument when adding each piece of the plot:

```
In [40]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
In [41]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[41]: [<matplotlib.lines.Line2D at 0x4720a90>]
In [42]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[42]: [<matplotlib.lines.Line2D at 0x4720f90>]
In [43]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[43]: [<matplotlib.lines.Line2D at 0x4723550>]
```

Once you've done this, you can either call `ax.legend()` or `plt.legend()` to automatically create a legend:

```
In [44]: ax.legend(loc='best')
```

See [Figure 8-10](#). The `loc` tells matplotlib where to place the plot. If you aren't picky '`best`' is a good option, as it will choose a location that is most out of the way. To exclude one or more elements from the legend, pass no label or `label='nolegend'`.

Annotations and Drawing on a Subplot

In addition to the standard plot types, you may wish to draw your own plot annotations, which could consist of text, arrows, or other shapes.

Annotations and text can be added using the `text`, `arrow`, and `annotate` functions. `text` draws text at given coordinates (`x`, `y`) on the plot with optional custom styling:

```
ax.text(x, y, 'Hello world!',  
        family='monospace', fontsize=10)
```

Annotations can draw both text and arrows arranged appropriately. As an example, let's plot the closing S&P 500 index price since 2007 (obtained from Yahoo! Finance) and annotate it with some of the important dates from the 2008-2009 financial crisis. See [Figure 8-11](#) for the result:

```
from datetime import datetime  
  
fig = plt.figure()  
ax = fig.add_subplot(1, 1, 1)  
  
data = pd.read_csv('ch08/spx.csv', index_col=0, parse_dates=True)  
spx = data['SPX']  
  
spx.plot(ax=ax, style='k-')  
  
crisis_data = [  
    (datetime(2007, 10, 11), 'Peak of bull market'),  
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),  
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')  
]  
  
for date, label in crisis_data:  
    ax.annotate(label, xy=(date, spx.asof(date) + 50),  
               xytext=(date, spx.asof(date) + 200),  
               arrowprops=dict(facecolor='black'),  
               horizontalalignment='left', verticalalignment='top')  
  
# Zoom in on 2007-2010  
ax.set_xlim(['1/1/2007', '1/1/2011'])  
ax.set_ylim([600, 1800])  
  
ax.set_title('Important dates in 2008-2009 financial crisis')
```

See the online matplotlib gallery for many more annotation examples to learn from.

Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as *patches*. Some of these, like `Rectangle` and `Circle` are found in `matplotlib.pyplot`, but the full set is located in `matplotlib.patches`.

To add a shape to a plot, you create the patch object `shp` and add it to a subplot by calling `ax.add_patch(shp)` (see [Figure 8-12](#)):

```
fig = plt.figure()  
ax = fig.add_subplot(1, 1, 1)  
  
rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)  
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)  
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],  
                  color='g', alpha=0.5)
```

```
ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

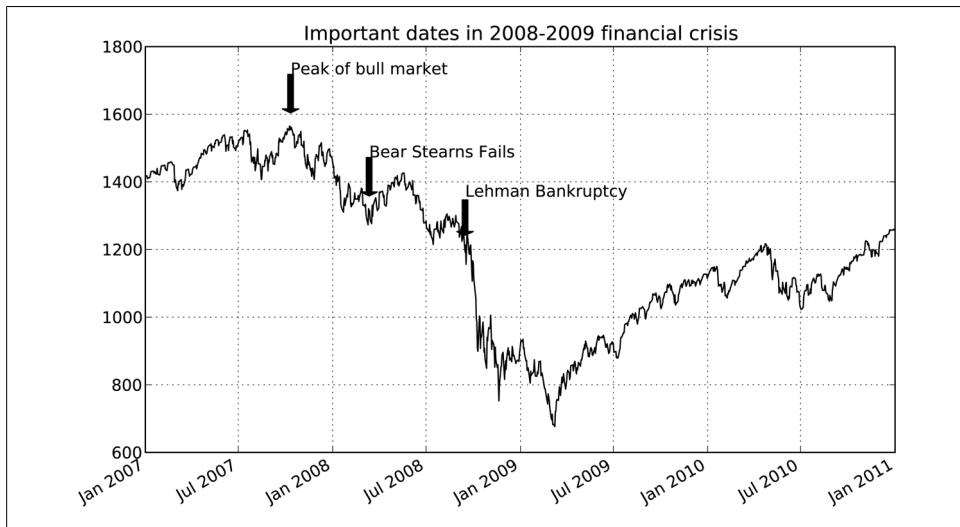


Figure 8-11. Important dates in 2008-2009 financial crisis

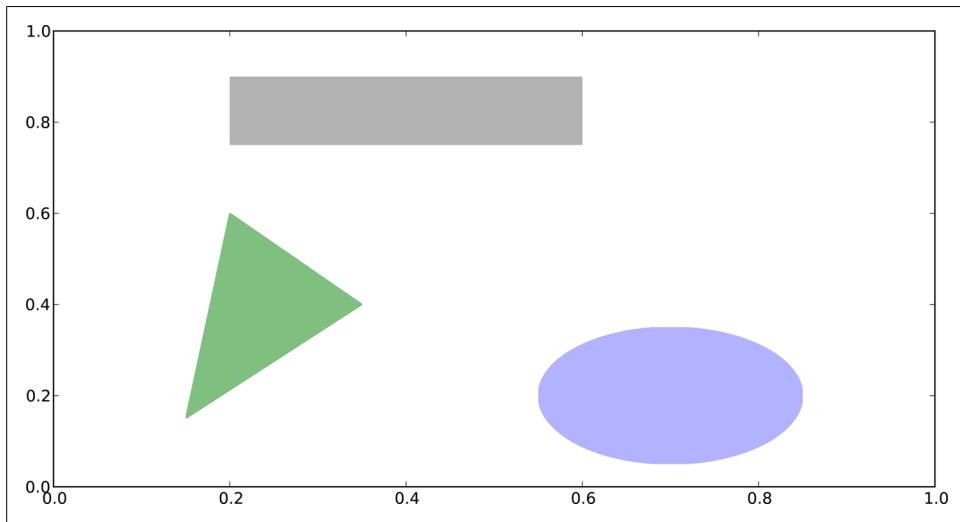


Figure 8-12. Figure composed from 3 different patches

If you look at the implementation of many familiar plot types, you will see that they are assembled from patches.

Saving Plots to File

The active figure can be saved to file using `plt.savefig`. This method is equivalent to the figure object's `savefig` instance method. For example, to save an SVG version of a figure, you need only type:

```
plt.savefig('figpath.svg')
```

The file type is inferred from the file extension. So if you used `.pdf` instead you would get a PDF. There are a couple of important options that I use frequently for publishing graphics: `dpi`, which controls the dots-per-inch resolution, and `bbox_inches`, which can trim the whitespace around the actual figure. To get the same plot as a PNG above with minimal whitespace around the plot and at 400 DPI, you would do:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

`savefig` doesn't have to write to disk; it can also write to any file-like object, such as a `StringIO`:

```
from io import StringIO
buffer = StringIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

For example, this is useful for serving dynamically-generated images over the web.

Table 8-2. Figure.savefig options

Argument	Description
<code>fname</code>	String containing a filepath or a Python file-like object. The figure format is inferred from the file extension, e.g. <code>.pdf</code> for PDF or <code>.png</code> for PNG.
<code>dpi</code>	The figure resolution in dots per inch; defaults to 100 out of the box but can be configured
<code>facecolor</code> , <code>edge color</code>	The color of the figure background outside of the subplots. ' <code>w</code> ' (white), by default
<code>format</code>	The explicit file format to use (<code>'png'</code> , <code>'pdf'</code> , <code>'svg'</code> , <code>'ps'</code> , <code>'eps'</code> , ...)
<code>bbox_inches</code>	The portion of the figure to save. If <code>'tight'</code> is passed, will attempt to trim the empty space around the figure

matplotlib Configuration

matplotlib comes configured with color schemes and defaults that are geared primarily toward preparing figures for publication. Fortunately, nearly all of the default behavior can be customized via an extensive set of global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on. There are two main ways to interact with the matplotlib configuration system. The first is programmatically from Python using the `rc` method. For example, to set the global default figure size to be 10 x 10, you could enter:

```
plt.rc('figure', figsize=(10, 10))
```

The first argument to `rc` is the component you wish to customize, such as '`figure`', '`axes`', '`xtick`', '`ytick`', '`grid`', '`legend`' or many others. After that can follow a sequence of keyword arguments indicating the new parameters. An easy way to write down the options in your program is as a dict:

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size'   : 'small'}
plt.rc('font', **font_options)
```

For more extensive customization and to see a list of all the options, matplotlib comes with a configuration file `matplotlibrc` in the `matplotlib/mpl-data` directory. If you customize this file and place it in your home directory titled `.matplotlibrc`, it will be loaded each time you use matplotlib.

Plotting Functions in pandas

As you've seen, matplotlib is actually a fairly low-level tool. You assemble a plot from its base components: the data display (the type of plot: line, bar, box, scatter, contour, etc.), legend, title, tick labels, and other annotations. Part of the reason for this is that in many cases the data needed to make a complete plot is spread across many objects. In pandas we have row labels, column labels, and possibly grouping information. This means that many kinds of fully-formed plots that would ordinarily require a lot of matplotlib code can be expressed in one or two concise statements. Therefore, pandas has an increasing number of high-level plotting methods for creating standard visualizations that take advantage of how data is organized in DataFrame objects.



As of this writing, the plotting functionality in pandas is undergoing quite a bit of work. As part of the 2012 Google Summer of Code program, a student is working full time to add features and to make the interface more consistent and usable. Thus, it's possible that this code may fall out-of-date faster than the other things in this book. The online pandas documentation will be the best resource in that event.

Line Plots

Series and DataFrame each have a `plot` method for making many different plot types. By default, they make line plots (see [Figure 8-13](#)):

```
In [55]: s = Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))
In [56]: s.plot()
```

The Series object's index is passed to matplotlib for plotting on the X axis, though this can be disabled by passing `use_index=False`. The X axis ticks and limits can be adjusted using the `xticks` and `xlim` options, and Y axis respectively using `yticks` and `ylim`. See

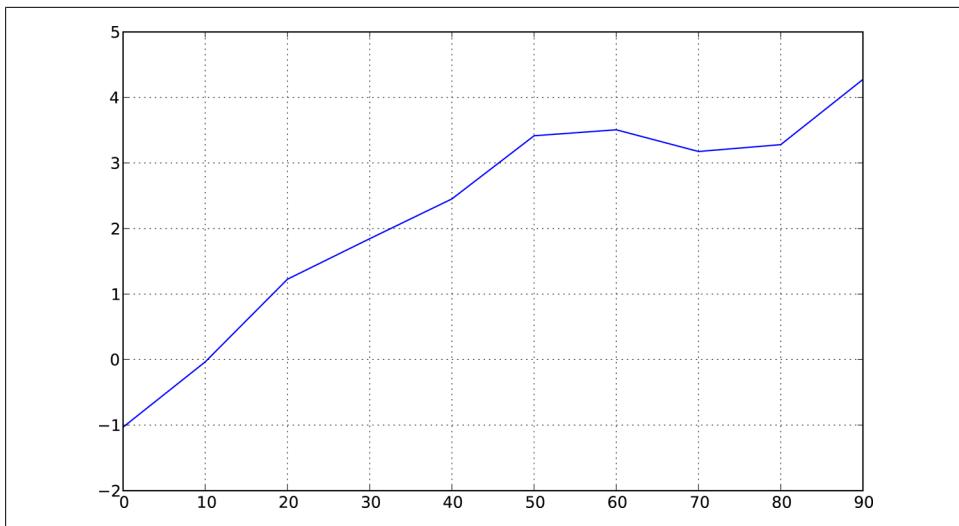


Figure 8-13. Simple Series plot example

[Table 8-3](#) for a full listing of `plot` options. I'll comment on a few more of them throughout this section and leave the rest to you to explore.

Most of pandas's plotting methods accept an optional `ax` parameter, which can be a matplotlib subplot object. This gives you more flexible placement of subplots in a grid layout. There will be more on this in the later section on the matplotlib API.

DataFrame's `plot` method plots each of its columns as a different line on the same subplot, creating a legend automatically (see [Figure 8-14](#)):

```
In [57]: df = DataFrame(np.random.randn(10, 4).cumsum(0),
....:                   columns=['A', 'B', 'C', 'D'],
....:                   index=np.arange(0, 100, 10))

In [58]: df.plot()
```



Additional keyword arguments to `plot` are passed through to the respective matplotlib plotting function, so you can further customize these plots by learning more about the matplotlib API.

Table 8-3. Series.plot method arguments

Argument	Description
<code>label</code>	Label for plot legend
<code>ax</code>	matplotlib subplot object to plot on. If nothing passed, uses active matplotlib subplot
<code>style</code>	Style string, like ' <code>ko--</code> ', to be passed to matplotlib.
<code>alpha</code>	The plot fill opacity (from 0 to 1)

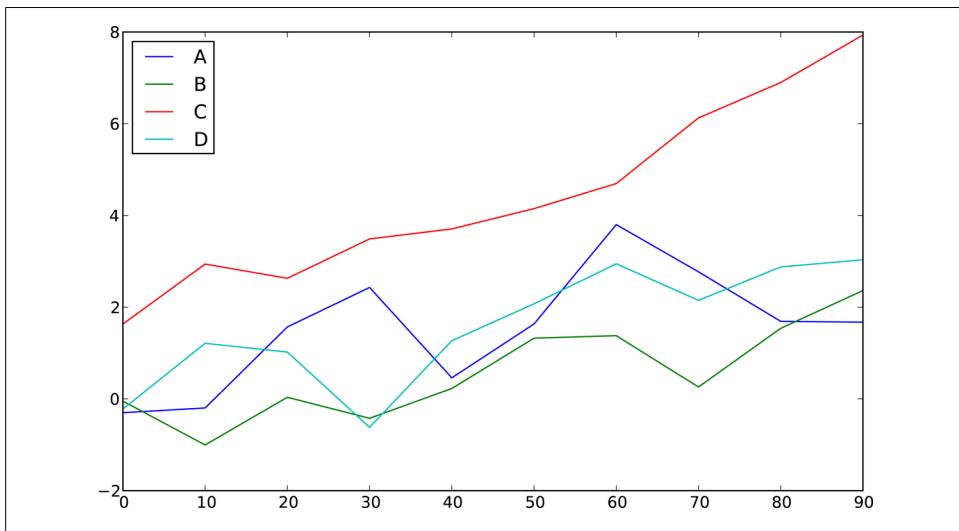


Figure 8-14. Simple DataFrame plot example

Argument	Description
kind	Can be 'line', 'bar', 'barh', 'kde'
logy	Use logarithmic scaling on the Y axis
use_index	Use the object index for tick labels
rot	Rotation of tick labels (0 through 360)
xticks	Values to use for X axis ticks
yticks	Values to use for Y axis ticks
xlim	X axis limits (e.g. [0, 10])
ylim	Y axis limits
grid	Display axis grid (on by default)

DataFrame has a number of options allowing some flexibility with how the columns are handled; for example, whether to plot them all on the same subplot or to create separate subplots. See [Table 8-4](#) for more on these.

Table 8-4. DataFrame-specific plot arguments

Argument	Description
subplots	Plot each DataFrame column in a separate subplot
sharex	If subplots=True, share the same X axis, linking ticks and limits
sharey	If subplots=True, share the same Y axis
figsize	Size of figure to create as tuple

Argument	Description
title	Plot title as string
legend	Add a subplot legend (True by default)
sort_columns	Plot columns in alphabetical order; by default uses existing column order



For time series plotting, see [Chapter 10](#).

Bar Plots

Making bar plots instead of line plots is as simple as passing `kind='bar'` (for vertical bars) or `kind='barh'` (for horizontal bars). In this case, the Series or DataFrame index will be used as the X (`bar`) or Y (`barh`) ticks (see [Figure 8-15](#)):

```
In [59]: fig, axes = plt.subplots(2, 1)
```

```
In [60]: data = Series(np.random.rand(16), index=list('abcdefghijklmnp'))
```

```
In [61]: data.plot(kind='bar', ax=axes[0], color='k', alpha=0.7)
```

```
Out[61]: <matplotlib.axes.AxesSubplot at 0x4ee7750>
```

```
In [62]: data.plot(kind='barh', ax=axes[1], color='k', alpha=0.7)
```



For more on the `plt.subplots` function and matplotlib axes and figures, see the later section in this chapter.

With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value. See [Figure 8-16](#):

```
In [63]: df = DataFrame(np.random.rand(6, 4),
....:                   index=['one', 'two', 'three', 'four', 'five', 'six'],
....:                   columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
```

```
In [64]: df
```

```
Out[64]:
```

Genus	A	B	C	D
one	0.301686	0.156333	0.371943	0.270731
two	0.750589	0.525587	0.689429	0.358974
three	0.381504	0.667707	0.473772	0.632528
four	0.942408	0.180186	0.708284	0.641783
five	0.840278	0.909589	0.010041	0.653207
six	0.062854	0.589813	0.811318	0.060217

```
In [65]: df.plot(kind='bar')
```

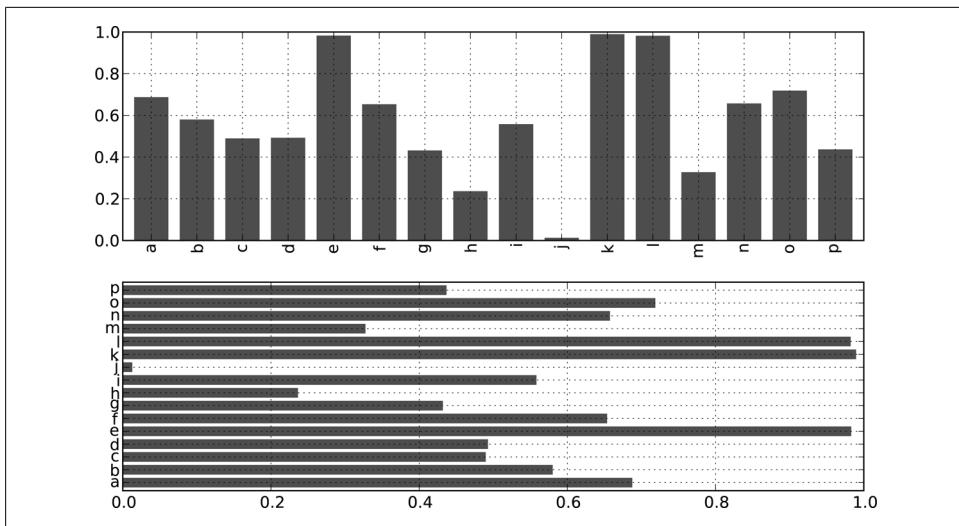


Figure 8-15. Horizontal and vertical bar plot example

Note that the name “Genus” on the DataFrame’s columns is used to title the legend. Stacked bar plots are created from a DataFrame by passing `stacked=True`, resulting in the value in each row being stacked together (see [Figure 8-17](#)):

```
In [67]: df.plot(kind='barh', stacked=True, alpha=0.5)
```



A useful recipe for bar plots (as seen in an earlier chapter) is to visualize a Series’s value frequency using `value_counts`:
`s.value_counts().plot(kind='bar')`

Returning to the tipping data set used earlier in the book, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day. I load the data using `read_csv` and make a cross-tabulation by day and party size:

```
In [68]: tips = pd.read_csv('ch08/tips.csv')
```

```
In [69]: party_counts = pd.crosstab(tips.day, tips.size)
```

```
In [70]: party_counts
```

```
Out[70]:
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
```

```
# Not many 1- and 6-person parties
In [71]: party_counts = party_counts[:, 2:5]
```

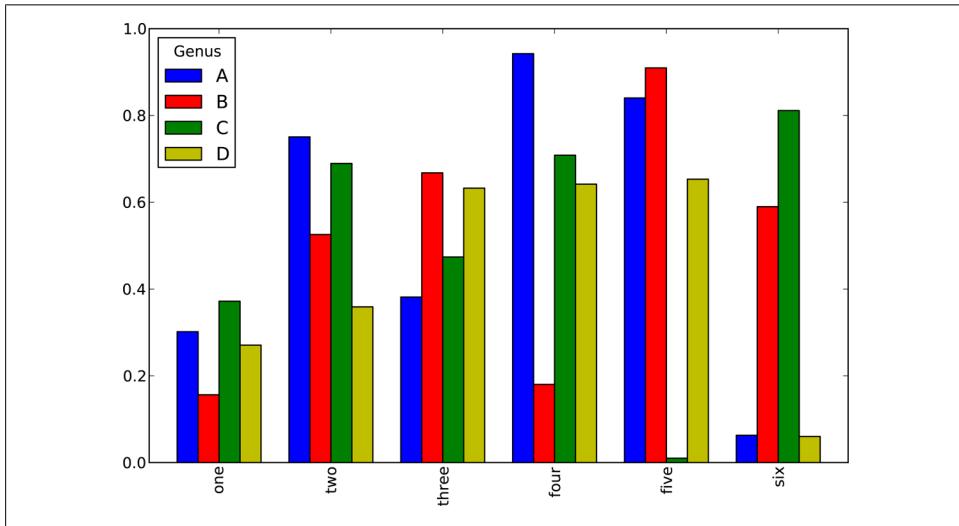


Figure 8-16. DataFrame bar plot example

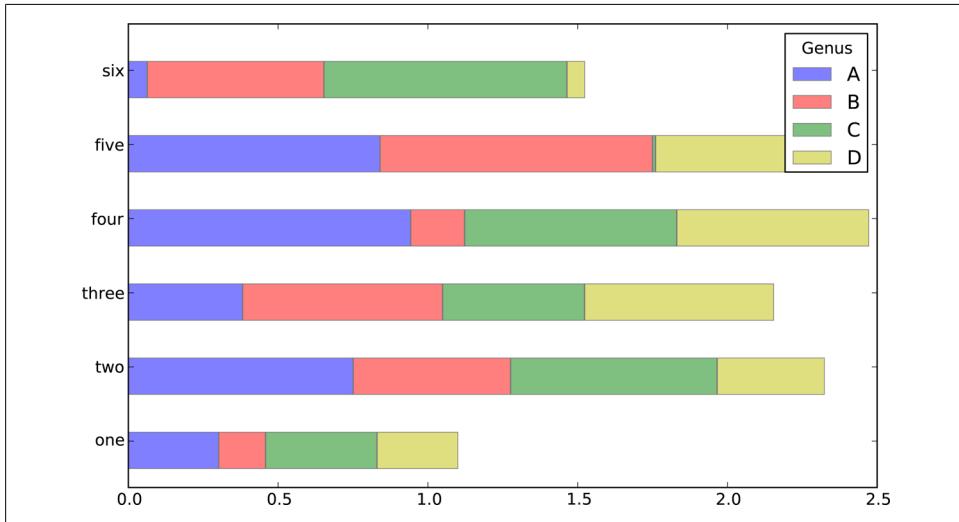


Figure 8-17. DataFrame stacked bar plot example

Then, normalize so that each row sums to 1 (I have to cast to float to avoid integer division issues on Python 2.7) and make the plot (see [Figure 8-18](#)):

```
# Normalize to sum to 1
In [72]: party_pcts = party_counts.div(party_counts.sum(1).astype(float), axis=0)
```

```
In [73]: party_pcts  
Out[73]:  
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
```

```
In [74]: party_pcts.plot(kind='bar', stacked=True)
```

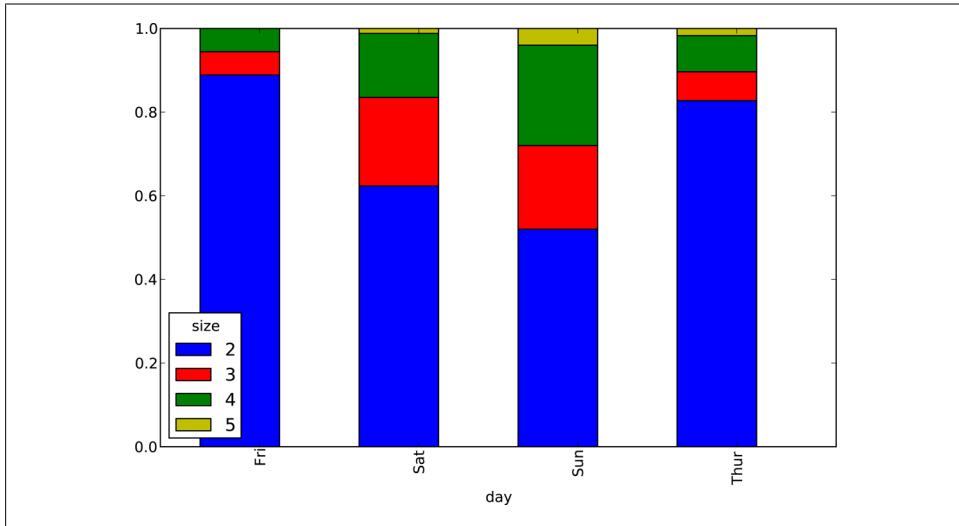


Figure 8-18. Fraction of parties by size on each day

So you can see that party sizes appear to increase on the weekend in this data set.

Histograms and Density Plots

A histogram, with which you may be well-acquainted, is a kind of bar plot that gives a discretized display of value frequency. The data points are split into discrete, evenly spaced bins, and the number of data points in each bin is plotted. Using the tipping data from before, we can make a histogram of tip percentages of the total bill using the `hist` method on the Series (see Figure 8-19):

```
In [76]: tips['tip_pct'] = tips['tip'] / tips['total_bill']
```

```
In [77]: tips['tip_pct'].hist(bins=50)
```

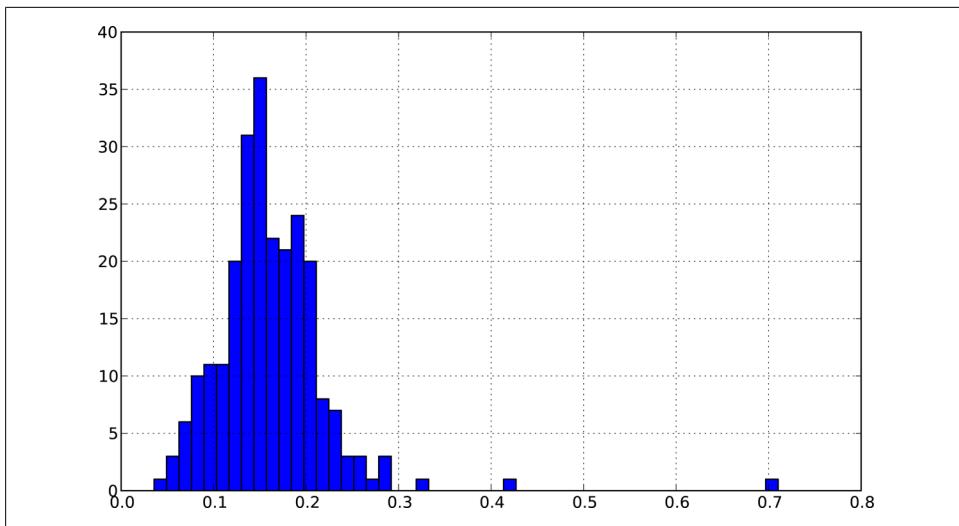


Figure 8-19. Histogram of tip percentages

A related plot type is a *density plot*, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data. A usual procedure is to approximate this distribution as a mixture of kernels, that is, simpler distributions like the normal (Gaussian) distribution. Thus, density plots are also known as KDE (kernel density estimate) plots. Using `plot` with `kind='kde'` makes a density plot using the standard mixture-of-normals KDE (see Figure 8-20):

```
In [79]: tips['tip_pct'].plot(kind='kde')
```

These two plot types are often plotted together; the histogram in normalized form (to give a binned density) with a kernel density estimate plotted on top. As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions (see Figure 8-21):

```
In [81]: comp1 = np.random.normal(0, 1, size=200) # N(0, 1)
```

```
In [82]: comp2 = np.random.normal(10, 2, size=200) # N(10, 4)
```

```
In [83]: values = Series(np.concatenate([comp1, comp2]))
```

```
In [84]: values.hist(bins=100, alpha=0.3, color='k', normed=True)
Out[84]: <matplotlib.axes.AxesSubplot at 0x5cd2350>
```

```
In [85]: values.plot(kind='kde', style='k--')
```

Scatter Plots

Scatter plots are a useful way of examining the relationship between two one-dimensional data series. matplotlib has a `scatter` plotting method that is the workhorse of

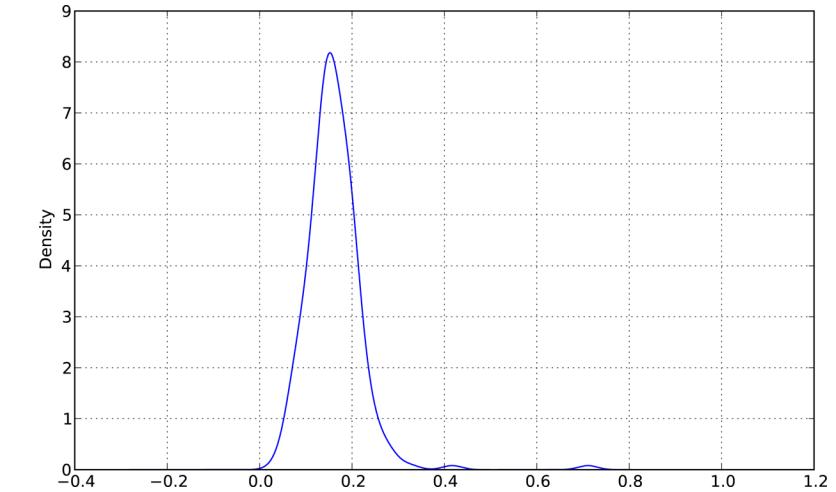


Figure 8-20. Density plot of tip percentages

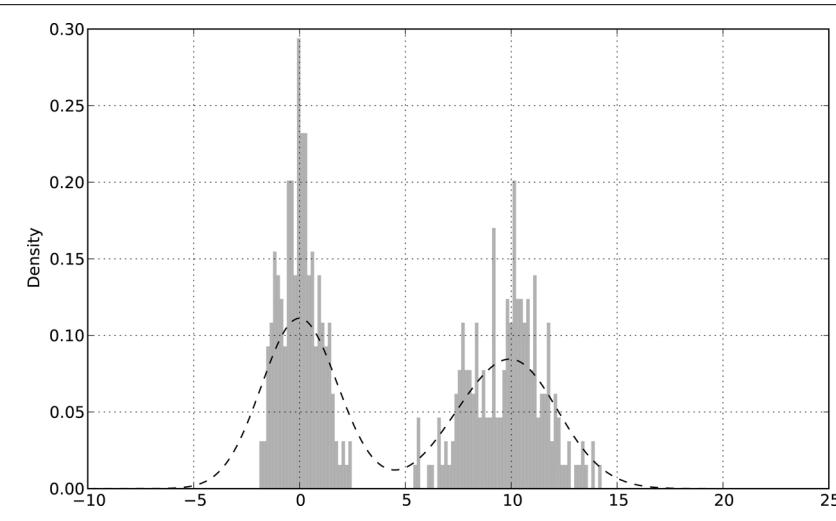


Figure 8-21. Normalized histogram of normal mixture with density estimate

making these kinds of plots. To give an example, I load the `macrodata` dataset from the `statsmodels` project, select a few variables, then compute log differences:

```
In [86]: macro = pd.read_csv('ch08/macrodta.csv')  
In [87]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]  
In [88]: trans_data = np.log(data).diff().dropna()
```

```
In [89]: trans_data[-5:]  
Out[89]:  
    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
```

It's easy to plot a simple scatter plot using `plt.scatter` (see [Figure 8-22](#)):

```
In [91]: plt.scatter(trans_data['m1'], trans_data['unemp'])  
Out[91]: <matplotlib.collections.PathCollection at 0x43c31d0>
```

```
In [92]: plt.title('Changes in log %s vs. log %s' % ('m1', 'unemp'))
```

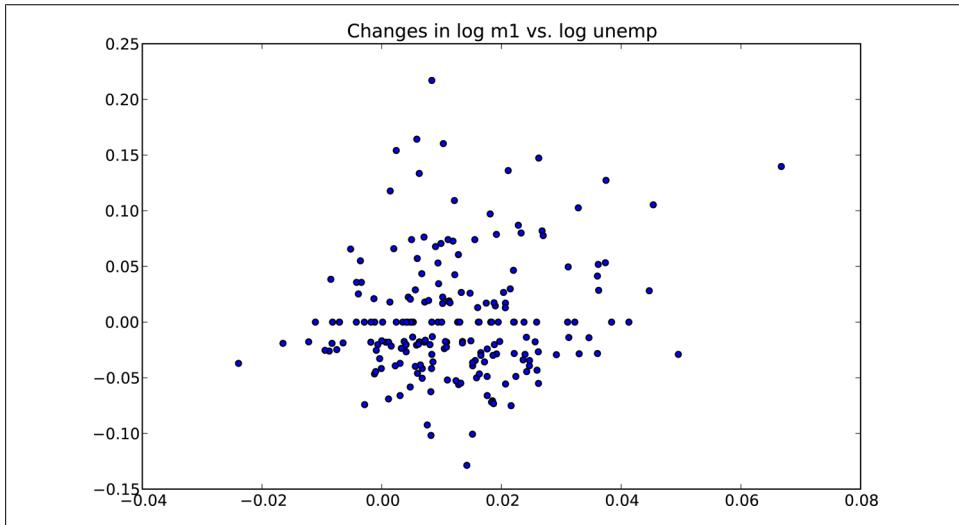


Figure 8-22. A simple scatter plot

In exploratory data analysis it's helpful to be able to look at all the scatter plots among a group of variables; this is known as a *pairs* plot or *scatter plot matrix*. Making such a plot from scratch is a bit of work, so pandas has a `scatter_matrix` function for creating one from a DataFrame. It also supports placing histograms or density plots of each variable along the diagonal. See [Figure 8-23](#) for the resulting plot:

```
In [93]: scatter_matrix(trans_data, diagonal='kde', color='k', alpha=0.3)
```

Plotting Maps: Visualizing Haiti Earthquake Crisis Data

Ushahidi is a non-profit software company that enables crowdsourcing of information related to natural disasters and geopolitical events via text message. Many of these data sets are then published on their [website](#) for analysis and visualization. I downloaded

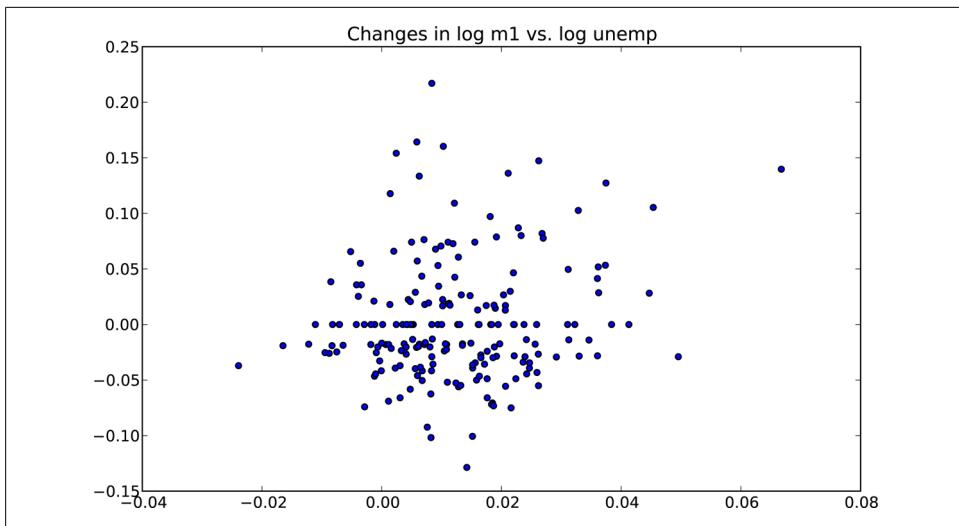


Figure 8-23. Scatter plot matrix of statsmodels macro data

the data collected during the 2010 Haiti earthquake crisis and aftermath, and I'll show you how I prepared the data for analysis and visualization using pandas and other tools we have looked at thus far. After downloading the CSV file from the above link, we can load it into a DataFrame using `read_csv`:

```
In [94]: data = pd.read_csv('ch08/Haiti.csv')
```

```
In [95]: data
Out[95]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3593 entries, 0 to 3592
Data columns:
Serial          3593 non-null values
INCIDENT TITLE  3593 non-null values
INCIDENT DATE   3593 non-null values
LOCATION        3593 non-null values
DESCRIPTION     3593 non-null values
CATEGORY        3587 non-null values
LATITUDE        3593 non-null values
LONGITUDE       3593 non-null values
APPROVED        3593 non-null values
VERIFIED        3593 non-null values
dtypes: float64(2), int64(1), object(7)
```

It's easy now to tinker with this data set to see what kinds of things we might want to do with it. Each row represents a report sent from someone's mobile phone indicating an emergency or some other problem. Each has an associated timestamp and a location as latitude and longitude:

```
In [96]: data[['INCIDENT DATE', 'LATITUDE', 'LONGITUDE']][:10]
Out[96]:
  INCIDENT DATE    LATITUDE    LONGITUDE
```

```

0 05/07/2010 17:26 18.233333 -72.533333
1 28/06/2010 23:06 50.226029 5.729886
2 24/06/2010 16:21 22.278381 114.174287
3 20/06/2010 21:59 44.407062 8.933989
4 18/05/2010 16:26 18.571084 -72.334671
5 26/04/2010 13:14 18.593707 -72.310079
6 26/04/2010 14:19 18.482800 -73.638800
7 26/04/2010 14:27 18.415000 -73.195000
8 15/03/2010 10:58 18.517443 -72.236841
9 15/03/2010 11:00 18.547790 -72.410010

```

The CATEGORY field contains a comma-separated list of codes indicating the type of message:

```

In [97]: data['CATEGORY'][::6]
Out[97]:
0           1. Urgences | Emergency, 3. Public Health,
1           1. Urgences | Emergency, 2. Urgences logistiques
2           2. Urgences logistiques | Vital Lines, 8. Autre |
3                           1. Urgences | Emergency,
4                           1. Urgences | Emergency,
5           5e. Communication lines down,
Name: CATEGORY

```

If you notice above in the data summary, some of the categories are missing, so we might want to drop these data points. Additionally, calling `describe` shows that there are some aberrant locations:

```

In [98]: data.describe()
Out[98]:
   Serial      LATITUDE     LONGITUDE
count 3593.000000  3593.000000  3593.000000
mean  2080.277484   18.611495 -72.322680
std   1171.100360    0.738572   3.650776
min    4.000000   18.041313 -74.452757
25%  1074.000000   18.524070 -72.417500
50%  2163.000000   18.539269 -72.335000
75%  3088.000000   18.561820 -72.293570
max   4052.000000   50.226029  114.174287

```

Cleaning the bad locations and removing the missing categories is now fairly simple:

```

In [99]: data = data[(data.LATITUDE > 18) & (data.LATITUDE < 20) &
.....          (data.LONGITUDE > -75) & (data.LONGITUDE < -70)
.....          & data.CATEGORY.notnull()]

```

Now we might want to do some analysis or visualization of this data by category, but each category field may have multiple categories. Additionally, each category is given as a code plus an English and possibly also a French code name. Thus, a little bit of wrangling is required to get the data into a more agreeable form. First, I wrote these two functions to get a list of all the categories and to split each category into a code and an English name:

```

def to_cat_list(catstr):
    stripped = (x.strip() for x in catstr.split(','))

```

```

        return [x for x in stripped if x]

def get_all_categories(cat_series):
    cat_sets = (set(to_cat_list(x)) for x in cat_series)
    return sorted(set.union(*cat_sets))

def get_english(cat):
    code, names = cat.split('.')
    if '|' in names:
        names = names.split(' | ')[1]
    return code, names.strip()

```

You can test out that the `get_english` function does what you expect:

```
In [101]: get_english('2. Urgences logistiques | Vital Lines')
Out[101]: ('2', 'Vital Lines')
```

Now, I make a `dict` mapping code to name because we'll use the codes for analysis. We'll use this later when adorning plots (note the use of a generator expression in lieu of a list comprehension):

```

In [102]: all_cats = get_all_categories(data.CATEGORY)

# Generator expression
In [103]: english_mapping = dict(get_english(x) for x in all_cats)

In [104]: english_mapping['2a']
Out[104]: 'Food Shortage'

In [105]: english_mapping['6c']
Out[105]: 'Earthquake and aftershocks'

```

There are many ways to go about augmenting the data set to be able to easily select records by category. One way is to add indicator (or dummy) columns, one for each category. To do that, first extract the unique category codes and construct a DataFrame of zeros having those as its columns and the same index as `data`:

```

def get_code(seq):
    return [x.split('.')[0] for x in seq if x]

all_codes = get_code(all_cats)
code_index = pd.Index(np.unique(all_codes))
dummy_frame = DataFrame(np.zeros((len(data), len(code_index))),
                        index=data.index, columns=code_index)

```

If all goes well, `dummy_frame` should look something like this:

```

In [107]: dummy_frame.ix[:, :6]
Out[107]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3569 entries, 0 to 3592
Data columns:
1      3569 non-null values
1a     3569 non-null values
1b     3569 non-null values
1c     3569 non-null values

```

```
1d    3569 non-null values
2    3569 non-null values
dtypes: float64(6)
```

As you recall, the trick is then to set the appropriate entries of each row to 1, lastly joining this with data:

```
for row, cat in zip(data.index, data.CATEGORY):
    codes = get_code(to_cat_list(cat))
    dummy_frame.ix[row, codes] = 1

data = data.join(dummy_frame.add_prefix('category_'))
```

data finally now has new columns like:

```
In [109]: data.ix[:, 10:15]
Out[109]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3569 entries, 0 to 3592
Data columns:
category_1    3569 non-null values
category_1a   3569 non-null values
category_1b   3569 non-null values
category_1c   3569 non-null values
category_1d   3569 non-null values
dtypes: float64(5)
```

Let's make some plots! As this is spatial data, we'd like to plot the data by category on a map of Haiti. The `basemap` toolkit (<http://matplotlib.github.com/basemap>), an add-on to matplotlib, enables plotting 2D data on maps in Python. `basemap` provides many different globe projections and a means for transforming projecting latitude and longitude coordinates on the globe onto a two-dimensional matplotlib plot. After some trial and error and using the above data as a guideline, I wrote this function which draws a simple black and white map of Haiti:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

def basic_haiti_map(ax=None, lllat=17.25, urlat=20.25,
                    lllon=-75, urlon=-71):
    # create polar stereographic Basemap instance.
    m = Basemap(ax=ax, projection='stere',
                lon_0=(urlon + lllon) / 2,
                lat_0=(urlat + lllat) / 2,
                llcrnrlat=lllat, urcrnrlat=urlat,
                llcrnrlon=lllon, urcrnrlon=urlon,
                resolution='f')
    # draw coastlines, state and country boundaries, edge of map.
    m.drawcoastlines()
    m.drawstates()
    m.drawcountries()
    return m
```

The idea, now, is that the returned `Basemap` object, knows how to transform coordinates onto the canvas. I wrote the following code to plot the data observations for a number

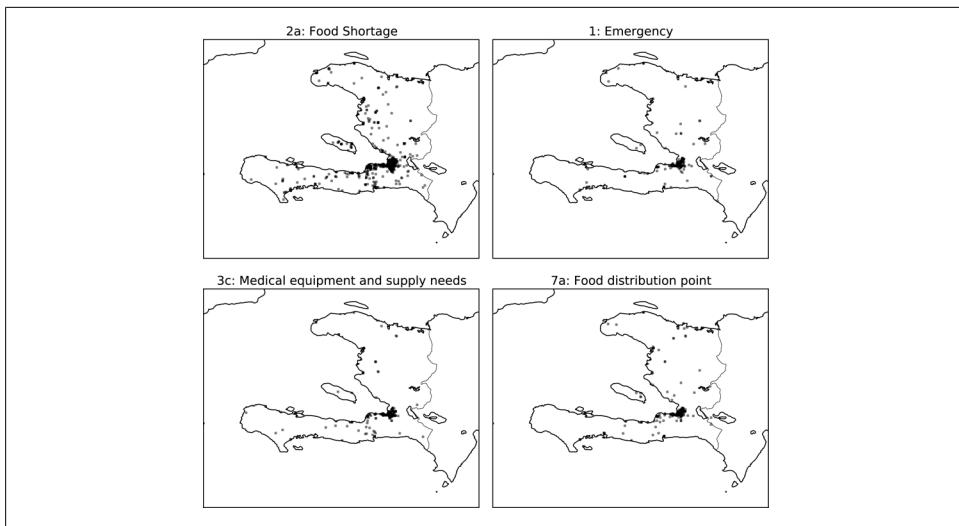


Figure 8-24. Haiti crisis data for 4 categories

of report categories. For each category, I filter down the data set to the coordinates labeled by that category, plot a Basemap on the appropriate subplot, transform the coordinates, then plot the points using the Basemap's plot method:

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 10))
fig.subplots_adjust(hspace=0.05, wspace=0.05)

to_plot = ['2a', '1', '3c', '7a']

lllat=17.25; urlat=20.25; llon=-75; urlon=-71

for code, ax in zip(to_plot, axes.flat):
    m = basic_haiti_map(ax, lllat=lllat, urlat=urlat,
                         llon=llon, urlon=urlon)

    cat_data = data[data['category_%s' % code] == 1]

    # compute map proj coordinates.
    x, y = m(cat_data.LONGITUDE, cat_data.LATITUDE)

    m.plot(x, y, 'k.', alpha=0.5)
    ax.set_title('%s: %s' % (code, english_mapping[code]))
```

The resulting figure can be seen in Figure 8-24.

It seems from the plot that most of the data is concentrated around the most populous city, Port-au-Prince. `basemap` allows you to overlap additional map data which comes from what are called *shapefiles*. I first downloaded a shapefile with roads in Port-au-Prince (see http://cegrp.cga.harvard.edu/haiti/?q=resources_data). The `Basemap` object conveniently has a `readshapefile` method so that, after extracting the road data archive, I added just the following lines to my code:

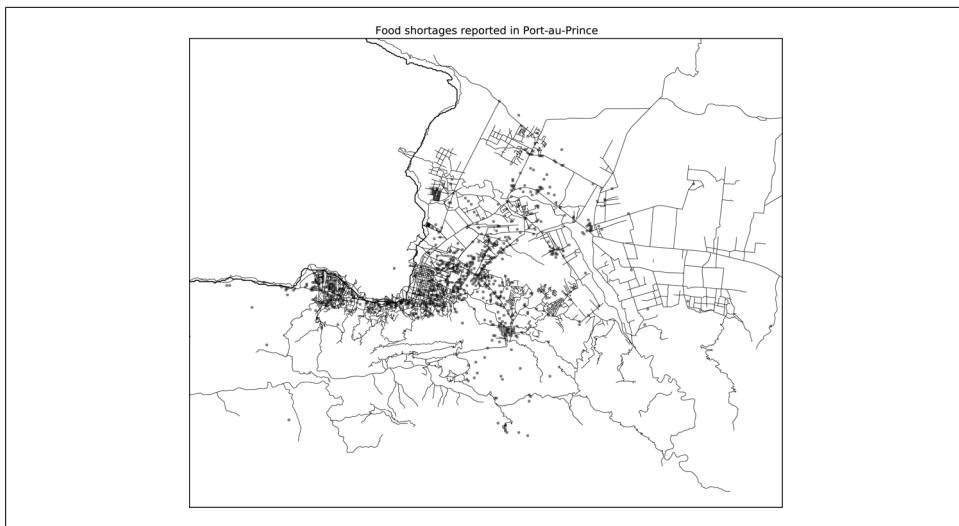


Figure 8-25. Food shortage reports in Port-au-Prince during the Haiti earthquake crisis

```
shapefile_path = 'ch08/PortAuPrince_Roads/PortAuPrince_Roads'
m.readshapefile(shapefile_path, 'roads')
```

After a little more trial and error with the latitude and longitude boundaries, I was able to make [Figure 8-25](#) for the “Food shortage” category.

Python Visualization Tool Ecosystem

As is common with open source, there are a plethora of options for creating graphics in Python (too many to list). In addition to open source, there are numerous commercial libraries with Python bindings.

In this chapter and throughout the book, I have been primarily concerned with matplotlib as it is the most widely used plotting tool in Python. While it's an important part of the scientific Python ecosystem, matplotlib has plenty of shortcomings when it comes to the creation and display of statistical graphics. MATLAB users will likely find matplotlib familiar, while R users (especially users of the excellent `ggplot2` and `trellis` packages) may be somewhat disappointed (at least as of this writing). It is possible to make beautiful plots for display on the web in matplotlib, but doing so often requires significant effort as the library is designed for the printed page. Aesthetics aside, it is sufficient for most needs. In pandas, I, along with the other developers, have sought to build a convenient user interface that makes it easier to make most kinds of plots commonplace in data analysis.

There are a number of other visualization tools in wide use. I list a few of them here and encourage you to explore the ecosystem.

Chaco

Chaco (<http://code.enthought.com/chaco/>), developed by Enthought, is a plotting toolkit suitable both for static plotting and interactive visualizations. It is especially well-suited for expressing complex visualizations with data interrelationships. Compared with matplotlib, Chaco has much better support for interacting with plot elements and rendering is very fast, making it a good choice for building interactive GUI applications.

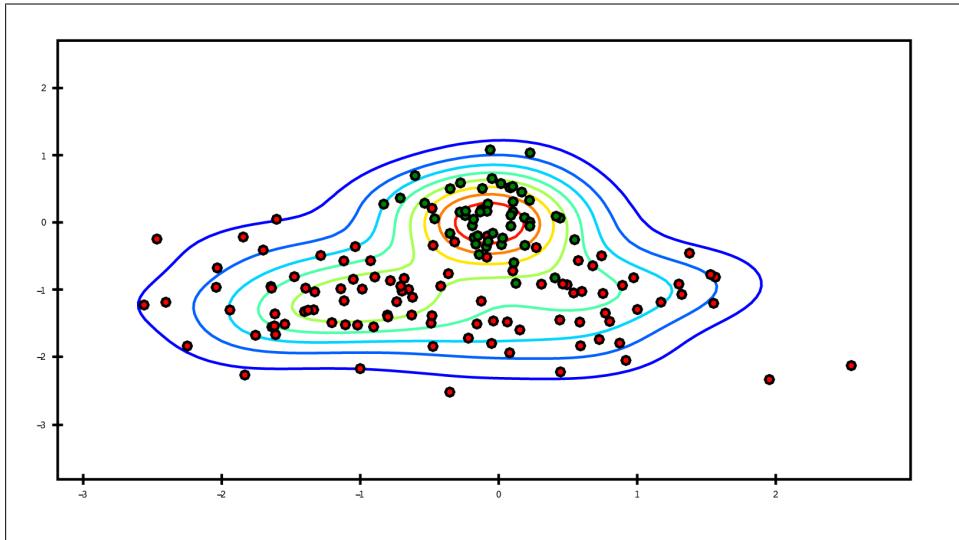


Figure 8-26. A Chaco example plot

mayavi

The mayavi project, developed by Prabhu Ramachandran, Gaël Varoquaux, and others, is a 3D graphics toolkit built on the open source C++ graphics library VTK. mayavi, like matplotlib, integrates with IPython so that it is easy to use interactively. The plots can be panned, rotated, and zoomed using the mouse and keyboard. I used mayavi to make one of the illustrations of broadcasting in [Chapter 12](#). While I don't show any mayavi-using code here, there is plenty of documentation and examples available online. In many cases, I believe it is a good alternative to a technology like WebGL, though the graphics are harder to share in interactive form.

Other Packages

Of course, there are numerous other visualization libraries and applications available in Python: PyQwt, Veusz, gnuplot-py, biggles, and others. I have seen PyQwt put to good use in GUI applications built using the Qt application framework using PyQt. While many of these libraries continue to be under active development (some of them

are part of much larger applications), I have noted in the last few years a general trend toward web-based technologies and away from desktop graphics. I'll say a few more words about this in the next section.

The Future of Visualization Tools?

Visualizations built on web technologies (that is, JavaScript-based) appear to be the inevitable future. Doubtlessly you have used many different kinds of static or interactive visualizations built in Flash or JavaScript over the years. New toolkits (such as d3.js and its numerous off-shoot projects) for building such displays are appearing all the time. In contrast, development in non web-based visualization has slowed significantly in recent years. This holds true of Python as well as other data analysis and statistical computing environments like R.

The development challenge, then, will be in building tighter integration between data analysis and preparation tools, such as pandas, and the web browser. I am hopeful that this will become a fruitful point of collaboration between Python and non-Python users as well.