CHAPTER 9

Plotting and Visualization

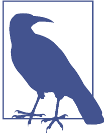
Making informative visualizations (sometimes called *plots*) is one of the most important tasks in data analysis. It may be a part of the exploratory process—for example, to help identify outliers or needed data transformations, or as a way of generating ideas for models. For others, building an interactive visualization for the web may be the end goal. Python has many add-on libraries for making static or dynamic visualizations, but I'll be mainly focused on matplotlib and libraries that build on top of it.

matplotlib is a desktop plotting package designed for creating plots and figures suitable for publication. The project was started by John Hunter in 2002 to enable a MATLAB-like plotting interface in Python. The matplotlib and IPython communities have collaborated to simplify interactive plotting from the IPython shell (and now, Jupyter notebook). matplotlib supports various GUI backends on all operating systems and can export visualizations to all of the common vector and raster graphics formats (PDF, SVG, JPG, PNG, BMP, GIF, etc.). With the exception of a few diagrams, nearly all of the graphics in this book were produced using matplotlib.

Over time, matplotlib has spawned a number of add-on toolkits for data visualization that use matplotlib for their underlying plotting. One of these is seaborn, which we explore later in this chapter.

The simplest way to follow the code examples in the chapter is to output plots in the Jupyter notebook. To set this up, execute the following statement in a Jupyter notebook:

```
%matplotlib inline
```



Since this book's first edition in 2012, many new data visualization libraries have been created, some of which (like Bokeh and Altair) take advantage of modern web technology to create interactive visualizations that integrate well with the Jupyter notebook. Rather than use multiple visualization tools in this book, I decided to stick with matplotlib for teaching the fundamentals, in particular since pandas has good integration with matplotlib. You can adapt the principles from this chapter to learn how to use other visualization libraries as well.

9.1 A Brief matplotlib API Primer

With matplotlib, we use the following import convention:

```
In [13]: import matplotlib.pyplot as plt
```

After running `%matplotlib notebook` in Jupyter (or simply `%matplotlib` in IPython), we can try creating a simple plot. If everything is set up right, a line plot like Figure 9-1 should appear:

```
In [14]: data = np.arange(10)

In [15]: data
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [16]: plt.plot(data)
```

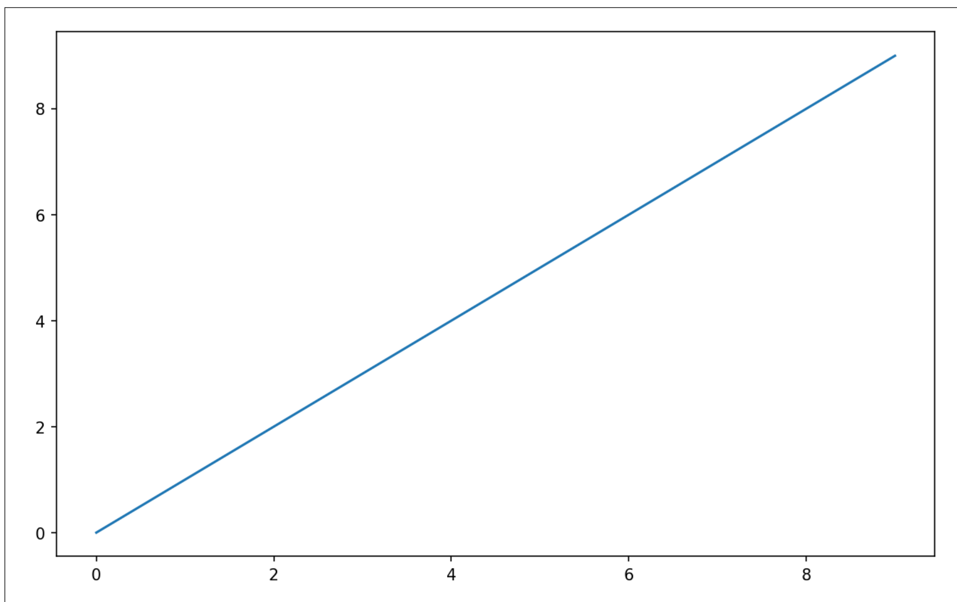
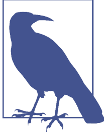


Figure 9-1. Simple line plot

While libraries like seaborn and pandas's built-in plotting functions will 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 comprehensive treatment of 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 learning advanced features.

Figures and Subplots

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

```
In [17]: fig = plt.figure()
```

In IPython, if you first run `%matplotlib` to set up the matplotlib integration, an empty plot window will appear, but in Jupyter nothing will be shown until we use a few more commands.

`plt.figure` has a number of options; notably, `figsize` will guarantee the figure has a certain size and aspect ratio if saved to disk.

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

```
In [18]: ax1 = fig.add_subplot(2, 2, 1)
```

This means that the figure should be 2 × 2 (so up to four plots in total), and we're selecting the first of four subplots (numbered from 1). If you create the next two subplots, you'll end up with a visualization that looks like Figure 9-2:

```
In [19]: ax2 = fig.add_subplot(2, 2, 2)

In [20]: ax3 = fig.add_subplot(2, 2, 3)
```

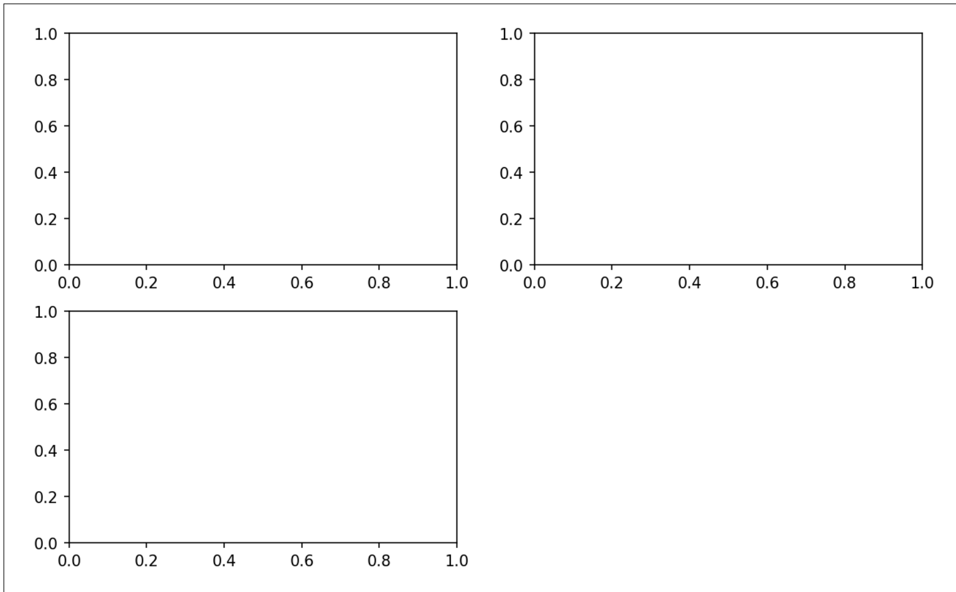


Figure 9-2. An empty matplotlib figure with three subplots

One nuance of using Jupyter notebooks is that plots are reset after each cell is evaluated, so you must put all of the plotting commands in a single notebook cell.

Here we run all of these commands in the same cell:

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

These plot axis objects have various methods that create different types of plots, and it is preferred to use the axis methods over the top-level plotting functions like `plt.plot`. For example, we could make a line plot with the `plot` method (see Figure 9-3):

```
In [21]: ax3.plot(np.random.standard_normal(50).cumsum(), color="black",
   ....:          linestyle="dashed")
```

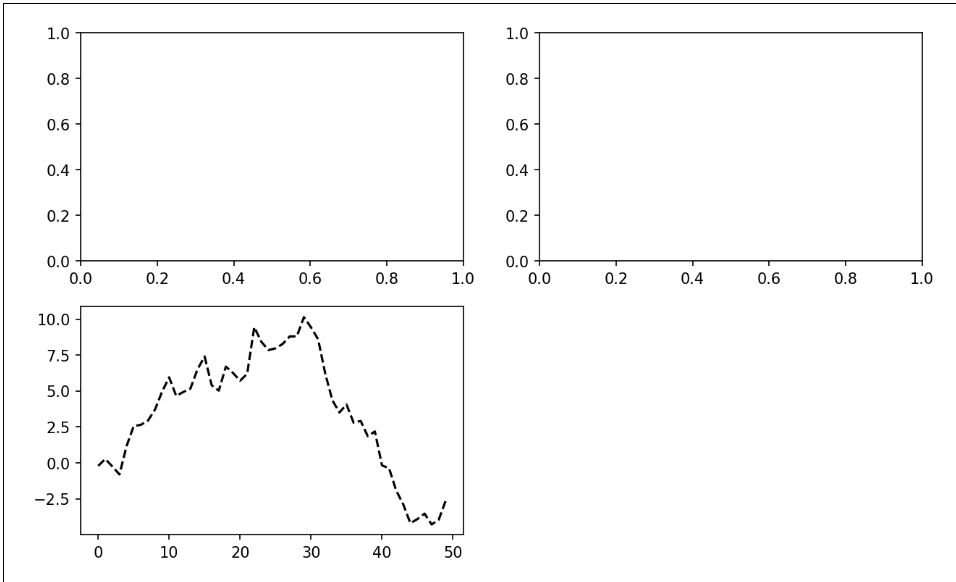


Figure 9-3. Data visualization after a single plot

You may notice output like `<matplotlib.lines.Line2D at ...>` when you run this. matplotlib returns objects that reference the plot subcomponent that was just added. A lot of the time you can safely ignore this output, or you can put a semicolon at the end of the line to suppress the output.

The additional options instruct matplotlib to plot a black dashed line. The objects returned by `fig.add_subplot` here are `AxesSubplot` objects, on which you can directly plot on the other empty subplots by calling each one's instance method (see Figure 9-4):

```
In [22]: ax1.hist(np.random.standard_normal(100), bins=20, color="black", alpha=0
.3);
In [23]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.standard_normal
(30));
```

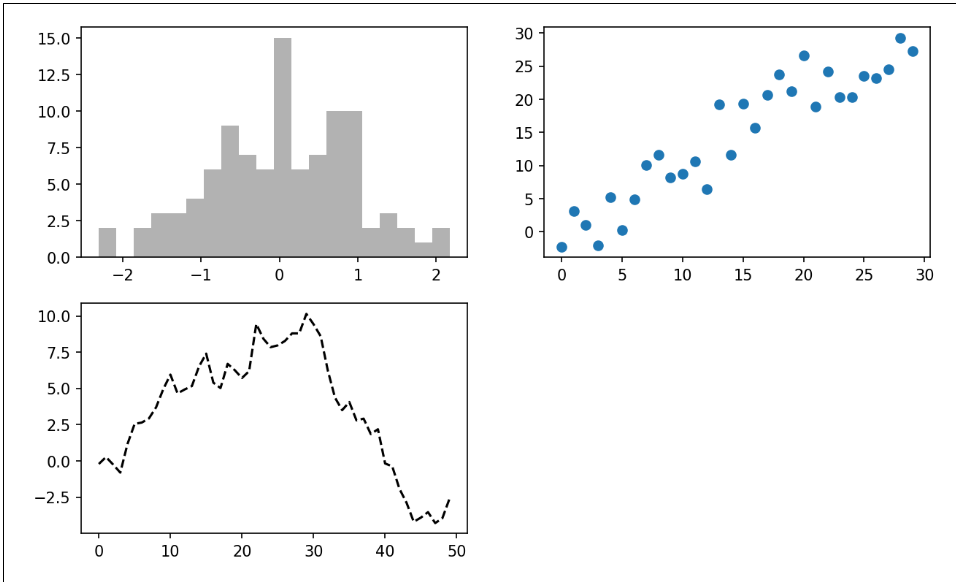


Figure 9-4. Data visualization after additional plots

The style option `alpha=0.3` sets the transparency of the overlaid plot.

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

To make creating a grid of subplots more convenient, matplotlib includes a `plt.subplots` method that creates a new figure and returns a NumPy array containing the created subplot objects:

```
In [25]: fig, axes = plt.subplots(2, 3)

In [26]: axes
Out[26]:
array([[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```

The `axes` array can then be indexed like a two-dimensional array; for example, `axes[0, 1]` refers to the subplot in the top row at the center. You can also indicate that subplots should have the same x- or y-axis using `sharex` and `sharey`, respectively. This can be useful when you're comparing data on the same scale; otherwise, matplotlib autoscales plot limits independently. See Table 9-1 for more on this method.

Table 9-1. `matplotlib.pyplot.subplots` options

Argument	Description
`nrows`	Number of rows of subplots
`ncols`	Number of columns of subplots
`sharex`	All subplots should use the same x-axis ticks (adjusting the `xlim` will affect all subplots)
`sharey`	All subplots should use the same y-axis ticks (adjusting the `ylim` will affect all subplots)
`subplot_kw`	Dictionary of keywords passed to `add_subplot` call used to create each subplot
`**fig_kw`	Additional keywords to `subplots` are used when creating the figure, such as `plt.subplots(2, 2, figsize=(8, 6))`

Adjusting the spacing around subplots

By default, matplotlib leaves a certain amount of padding around the outside of the subplots and in 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. You can change the spacing using the `subplots_adjust` method on `Figure` objects:

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

`wspace` and `hspace` control the percent of the figure width and figure height, respectively, to use as spacing between subplots. Here is a small example you can execute in Jupyter where I shrink the spacing all the way to zero (see Figure 9-5):

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

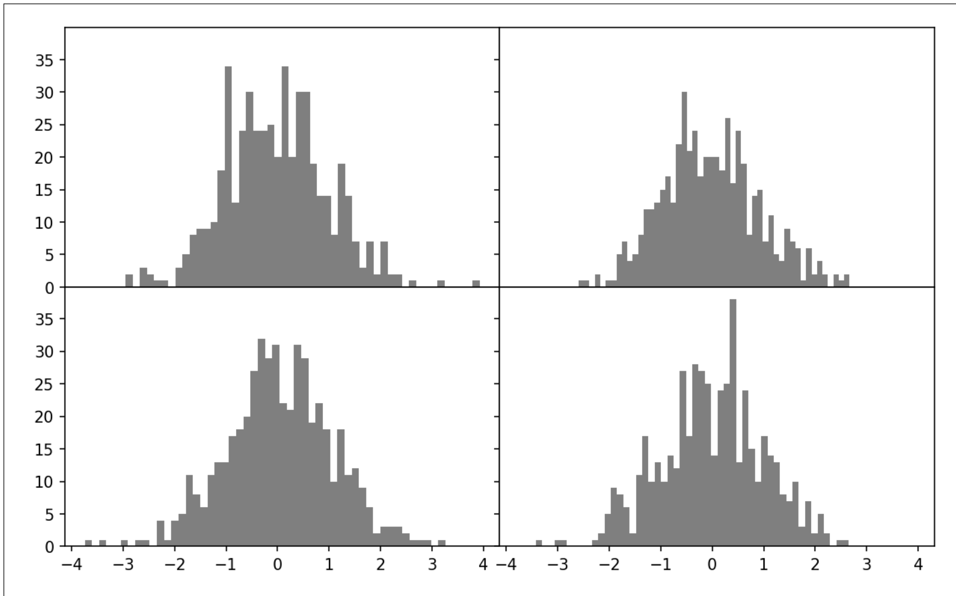


Figure 9-5. Data visualization with no inter-subplot spacing

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 (we'll look at how to do this in the later section "Ticks, Labels, and Legends" on page 290).

Colors, Markers, and Line Styles

matplotlib's line `plot` function accepts arrays of x and y coordinates and optional color styling options. For example, to plot `x` versus `y` with green dashes, you would execute:

```
ax.plot(x, y, linestyle="--", color="green")
```

A number of color names are provided for commonly used colors, but you can use any color on the spectrum by specifying its hex code (e.g., `"#CECECE"`). You can see some of the supported line styles by looking at the docstring for `plt.plot` (use `plt.plot?` in IPython or Jupyter). A more comprehensive reference is available in the online documentation.

Line plots can additionally have *markers* to highlight the actual data points. Since matplotlib's `plot` function creates a continuous line plot, interpolating between points, it can occasionally be unclear where the points lie. The marker can be supplied as an additional styling option (see Figure 9-6):

```
In [31]: ax = fig.add_subplot()

In [32]: ax.plot(np.random.standard_normal(30).cumsum(), color="black",
   ....:         linestyle="dashed", marker="o");
```

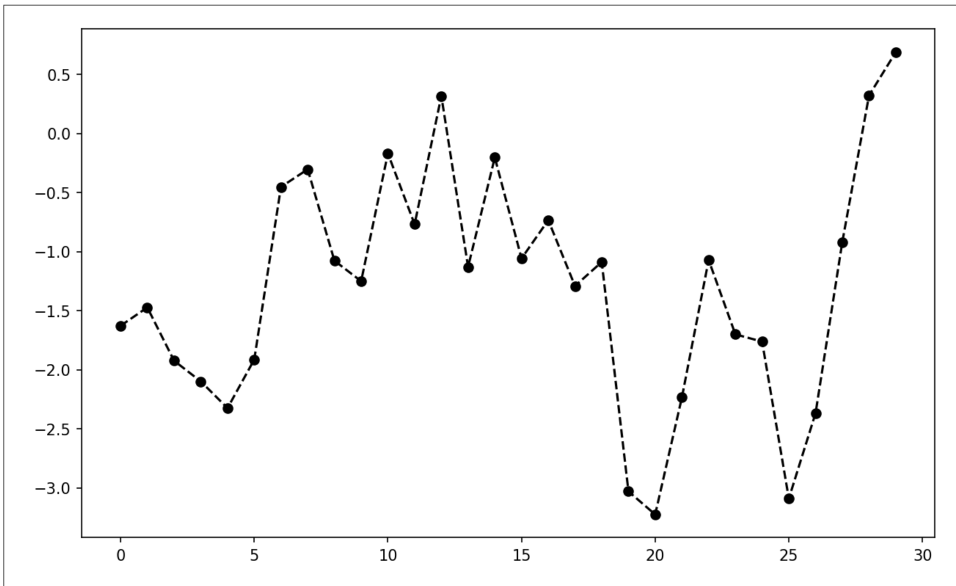


Figure 9-6. Line plot with markers

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the `drawstyle` option (see Figure 9-7):

```
In [34]: fig = plt.figure()

In [35]: ax = fig.add_subplot()

In [36]: data = np.random.standard_normal(30).cumsum()

In [37]: ax.plot(data, color="black", linestyle="dashed", label="Default");
In [38]: ax.plot(data, color="black", linestyle="dashed",
   ....:         drawstyle="steps-post", label="steps-post");
In [39]: ax.legend()
```

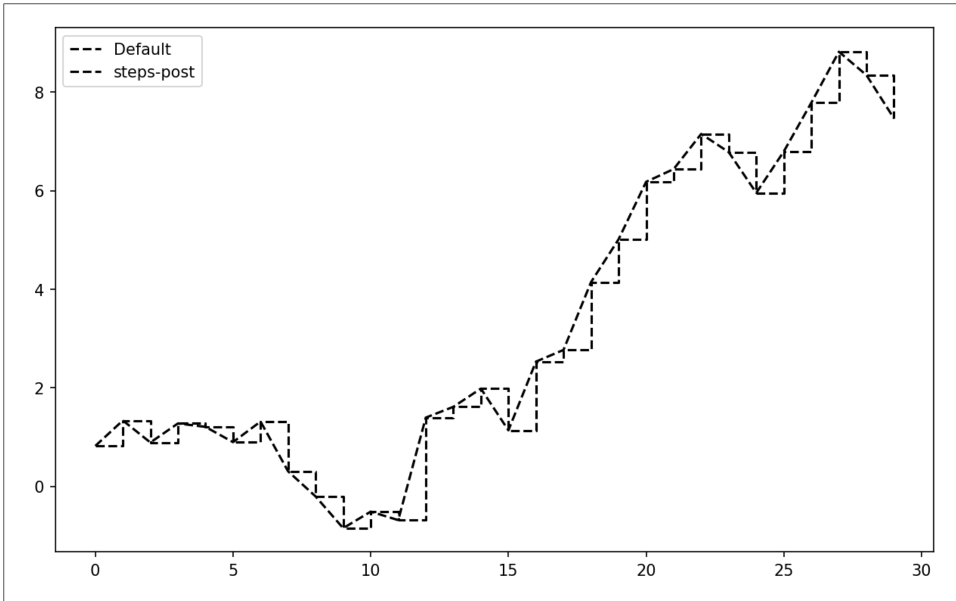
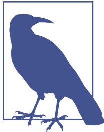


Figure 9-7. Line plot with different drawstyle options

Here, since we passed the `label` arguments to `plot`, we are able to create a plot legend to identify each line using `ax.legend`. I discuss legends more in "Ticks, Labels, and Legends" on page 290.



You must call `ax.legend` to create the legend, whether or not you passed the `label` options when plotting the data.

Ticks, Labels, and Legends

Most kinds of plot decorations can be accessed through methods on matplotlib axes objects. This includes 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 (e.g., `ax.xlim()` returns the current x-axis plotting range)
- Called with parameters sets the parameter value (e.g., `ax.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 corresponds to two methods on the subplot object itself; in the case of xlim, these are ax.get_xlim and ax.set_xlim.

Setting the title, axis labels, ticks, and tick labels

To illustrate customizing the axes, I'll create a simple figure and plot of a random walk (see Figure 9-8):

```
In [40]: fig, ax = plt.subplots()

In [41]: ax.plot(np.random.standard_normal(1000).cumsum());
```

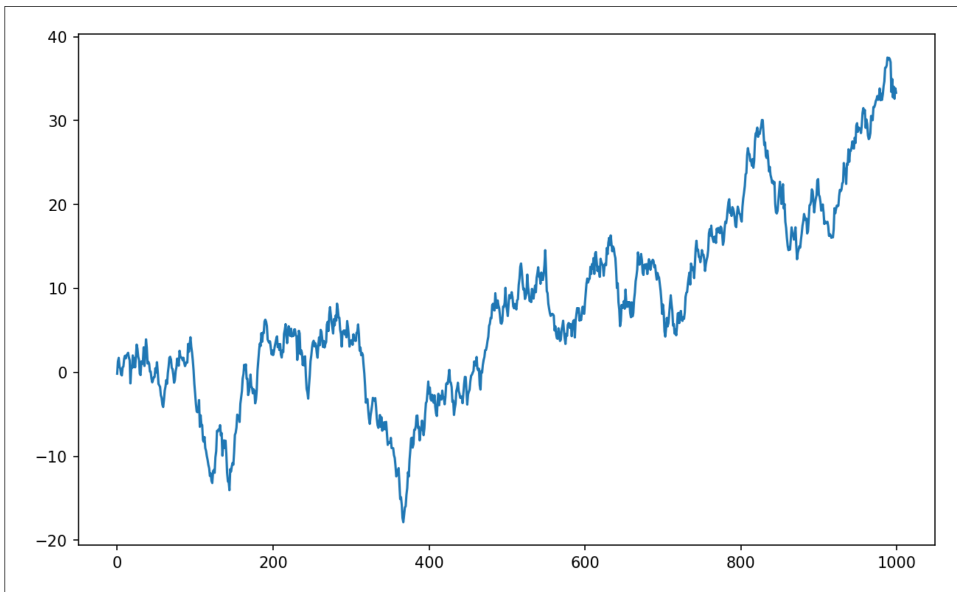


Figure 9-8. Simple plot for illustrating xticks (with default labels)

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

In [43]: labels = ax.set_xticklabels(["one", "two", "three", "four", "five"],
   ....:                             rotation=30, fontsize=8)
```

The `rotation` option sets the x tick labels at a 30-degree rotation. Lastly, `set_xlabel` gives a name to the x-axis, and `set_title` is the subplot title (see Figure 9-9 for the resulting figure):

```
In [44]: ax.set_xlabel("Stages")
Out[44]: Text(0.5, 6.666666666666652, 'Stages')

In [45]: ax.set_title("My first matplotlib plot")
```

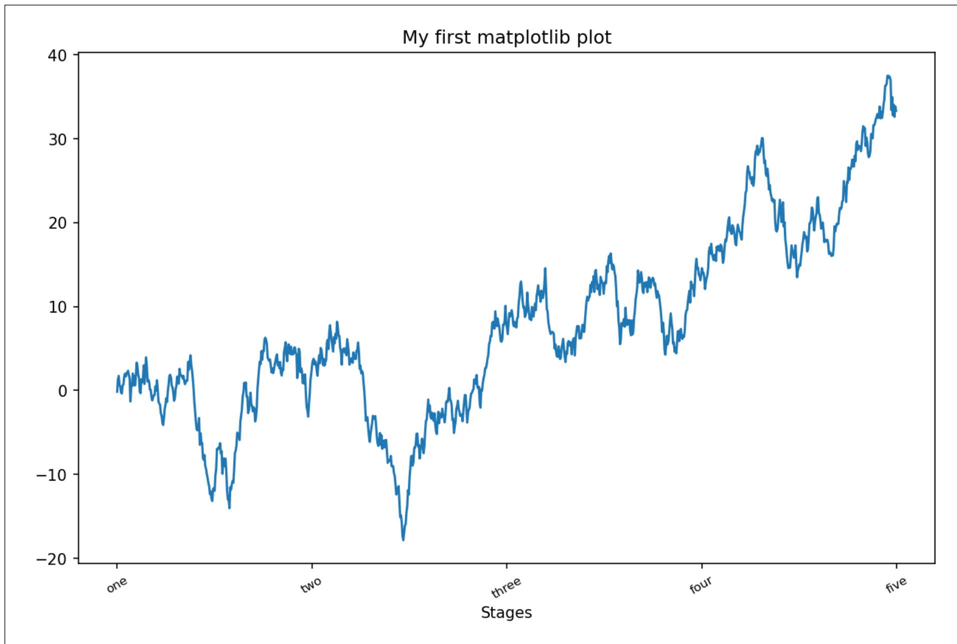


Figure 9-9. Simple plot for illustrating custom xticks

Modifying the y-axis consists of the same process, substituting `y` for `x` in this example. The axes class has a `set` method that allows batch setting of plot properties. From the prior example, we could also have written:

```
ax.set(title="My first matplotlib plot", xlabel="Stages")
```

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 [46]: fig, ax = plt.subplots()

In [47]: ax.plot(np.random.randn(1000).cumsum(), color="black", label="one");
In [48]: ax.plot(np.random.randn(1000).cumsum(), color="black", linestyle="dashed
```

```
",
   ....:          label="two");
In [49]: ax.plot(np.random.randn(1000).cumsum(), color="black", linestyle="dotted
",
   ....:          label="three");
```

Once you've done this, you can call `ax.legend()` to automatically create a legend. The resulting plot is in Figure 9-10:

```
In [50]: ax.legend()
```

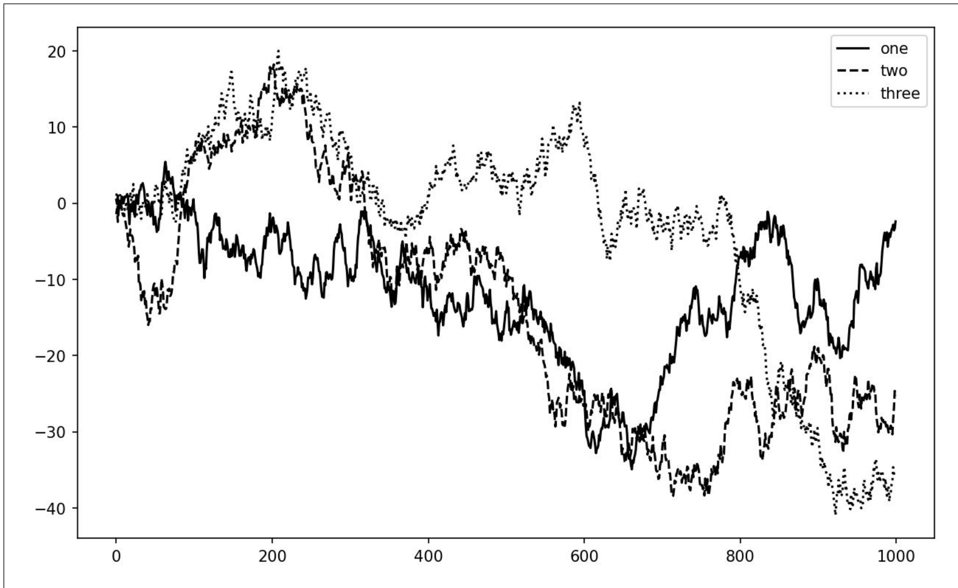


Figure 9-10. Simple plot with three lines and legend

The `legend` method has several other choices for the location `loc` argument. See the docstring (with `ax.legend?`) for more information.

The `loc` legend option tells matplotlib where to place the plot. The default is `"best"`, which tries to 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. You can add annotations and text 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. You can run this code example in a single cell in a Jupyter notebook. See Figure 9-11 for the result:

```
from datetime import datetime

fig, ax = plt.subplots()

data = pd.read_csv("examples/spx.csv", index_col=0, parse_dates=True)
spx = data["SPX"]

spx.plot(ax=ax, color="black")

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) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor="black", headwidth=4, width=2,
                                headlength=4),
                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 the 2008-2009 financial crisis")
```

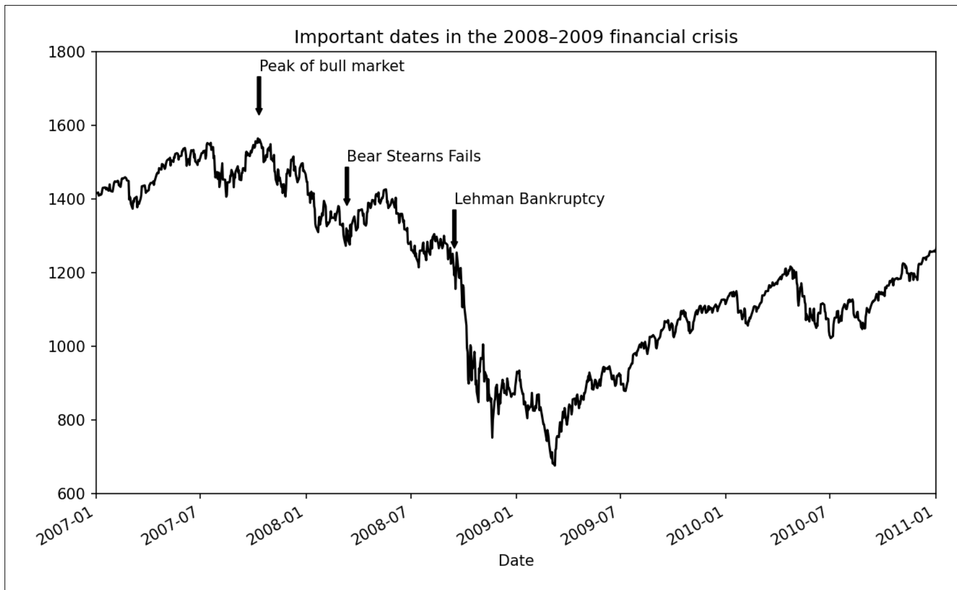


Figure 9-11. Important dates in the 2008–2009 financial crisis

There are a couple of important points to highlight in this plot. The `ax.annotate` method can draw labels at the indicated x and y coordinates. We use the `set_xlim` and `set_ylim` methods to manually set the start and end boundaries for the plot rather than using matplotlib's default. Lastly, `ax.set_title` adds a main title to the plot.

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 and add it to a subplot `ax` by passing the patch to `ax.add_patch` (see Figure 9-12):

```
fig, ax = plt.subplots()

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color="black", alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color="blue", alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                   color="green", alpha=0.5)

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

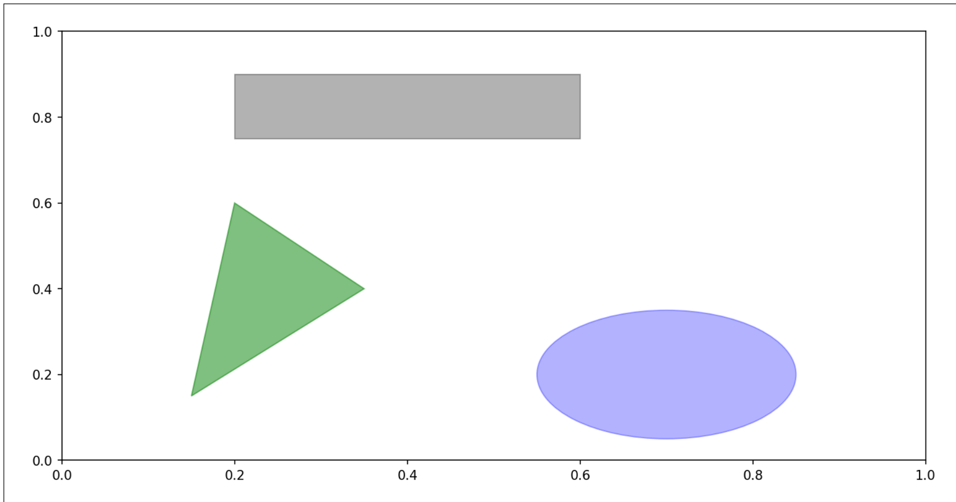


Figure 9-12. Data visualization composed from three different patches

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

Saving Plots to File

You can save the active figure to file using the figure object's `savefig` instance method. For example, to save an SVG version of a figure, you need only type:

```
fig.savefig("figpath.svg")
```

The file type is inferred from the file extension. So if you used `.pdf` instead, you would get a PDF. One important option that I use frequently for publishing graphics is `dpi`, which controls the dots-per-inch resolution. To get the same plot as a PNG at 400 DPI, you would do:

```
fig.savefig("figpath.png", dpi=400)
```

See Table 9-2 for a list of some other options for `savefig`. For a comprehensive listing, refer to the docstring in IPython or Jupyter.

Table 9-2. Some `fig.savefig` options

Argument	Description
`fname`	String containing a filepath or a Python file-like object. The figure format is inferred from the file extension (e.g., `.pdf` for PDF or `.png` for PNG).
`dpi`	The figure resolution in dots per inch; defaults to 100 in IPython or 72 in Jupyter out of the box but can be configured.
`facecolor, edgecolor`	The color of the figure background outside of the subplots; `"w"` (white), by default.
`format`	The explicit file format to use (`"png"`, `"pdf"`, `"svg"`, `"ps"`, `"eps"`, ...).

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 global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on. One way to modify the configuration programmatically from Python is to use the `rc` method; for example, to set the global default figure size to be 10 × 10, you could enter:

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

All of the current configuration settings are found in the `plt.rcParams` dictionary, and they can be restored to their default values by calling the `plt.rcdefaults()` function.

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. A convenient way to write down the options in your program is as a dictionary:

```
plt.rc("font", family="monospace", weight="bold", size=8)
```

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.

As we'll see in the next section, the seaborn package has several built-in plot themes or *styles* that use matplotlib's configuration system internally.

9.2 Plotting with pandas and seaborn

matplotlib can be a fairly low-level tool. You assemble a plot from its base components: the data display (i.e., the type of plot: line, bar, box, scatter, contour, etc.), legend, title, tick labels, and other annotations.

In pandas, we may have multiple columns of data, along with row and column labels. pandas itself has built-in methods that simplify creating visualizations from DataFrame and Series objects. Another library is `seaborn`, a high-level statistical graphics library built on matplotlib. seaborn simplifies creating many common visualization types.

Line Plots

Series and DataFrame have a `plot` attribute for making some basic plot types. By default, `plot()` makes line plots (see Figure 9-13):

```
In [61]: s = pd.Series(np.random.standard_normal(10).cumsum(), index=np.arange(0,
 100, 10))

In [62]: s.plot()
```

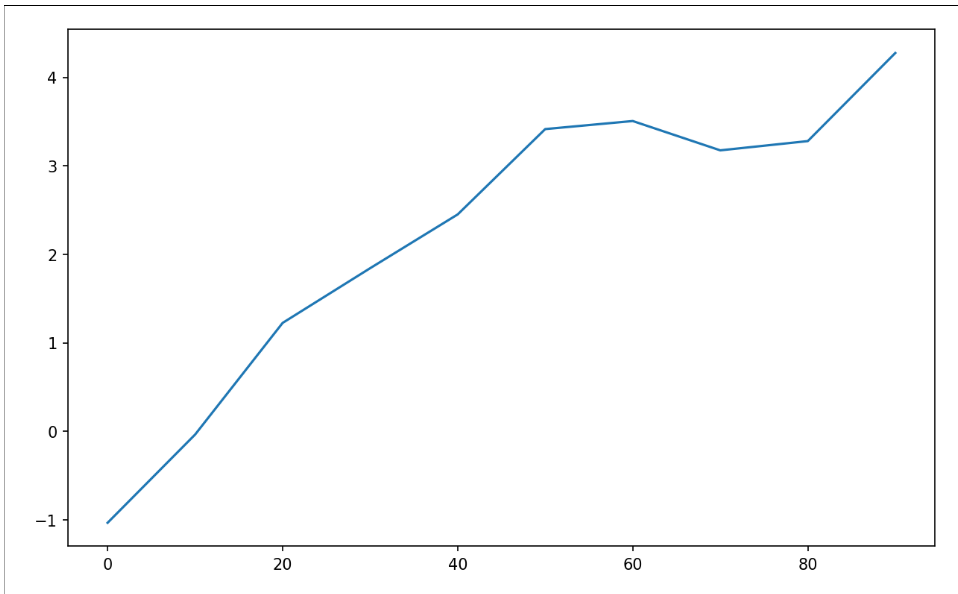


Figure 9-13. Simple Series plot

The Series object's index is passed to matplotlib for plotting on the x-axis, though you can disable this by passing `use_index=False`. The x-axis ticks and limits can be adjusted with the `xticks` and `xlim` options, and the y-axis respectively with `yticks`

and `ylim`. See Table 9-3 for a partial listing of `plot` options. I'll comment on a few more of them throughout this section and leave the rest for you to explore.

Table 9-3. `Series.plot` method arguments

Argument	Description
`label`	Label for plot legend
`ax`	matplotlib subplot object to plot on; if nothing passed, uses active matplotlib subplot
`style`	Style string, like `"ko--"`, to be passed to matplotlib
`alpha`	The plot fill opacity (from 0 to 1)
`kind`	Can be `"area"`, `"bar"`, `"barh"`, `"density"`, `"hist"`, `"kde"`, `"line"`, or `"pie"`; defaults to `"line"`
`figsize`	Size of the figure object to create
`logx`	Pass `True` for logarithmic scaling on the x axis; pass `"sym"` for symmetric logarithm that permits negative values
`logy`	Pass `True` for logarithmic scaling on the y axis; pass `"sym"` for symmetric logarithm that permits negative values
`title`	Title to use for the plot
`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 (off by default)

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.

DataFrame's `plot` method plots each of its columns as a different line on the same subplot, creating a legend automatically (see Figure 9-14):

```
In [63]: df = pd.DataFrame(np.random.standard_normal((10, 4)).cumsum(0),
   ....:                   columns=["A", "B", "C", "D"],
   ....:                   index=np.arange(0, 100, 10))

In [64]: plt.style.use('grayscale')

In [65]: df.plot()
```

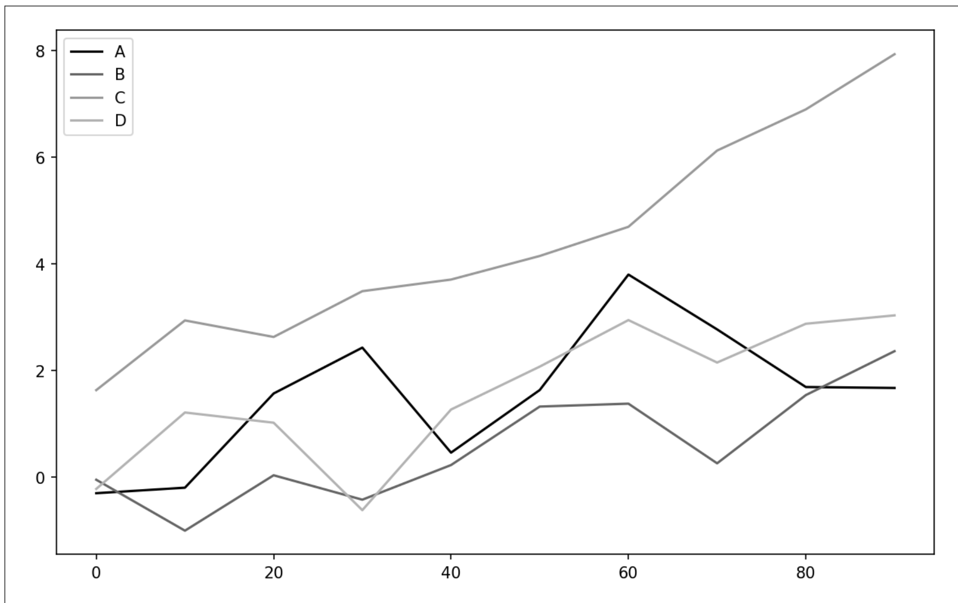
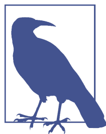
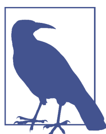


Figure 9-14. Simple DataFrame plot



Here I used `plt.style.use('grayscale')` to switch to a color scheme more suitable for black and white publication, since some readers will not be able to see the full color plots.

The `plot` attribute contains a "family" of methods for different plot types. For example, `df.plot()` is equivalent to `df.plot.line()`. We'll explore some of these methods next.

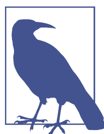


Additional keyword arguments to `plot` are passed through to the respective matplotlib plotting function, so you can further customize these plots by learning more about the matplotlib API.

DataFrame has a number of options allowing some flexibility for how the columns are handled, for example, whether to plot them all on the same subplot or to create separate subplots. See Table 9-4 for more on these.

Table 9-4. DataFrame-specific plot arguments

Argument	Description
`subplots`	Plot each DataFrame column in a separate subplot
`layouts`	2-tuple (rows, columns) providing layout of subplots
`sharex`	If `subplots=True`, share the same x-axis, linking ticks and limits
`sharey`	If `subplots=True`, share the same y-axis
`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 11.

Bar Plots

The `plot.bar()` and `plot.barh()` make vertical and horizontal bar plots, respectively. In this case, the Series or DataFrame index will be used as the x (`bar`) or y (`barh`) ticks (see Figure 9-15):

```
In [66]: fig, axes = plt.subplots(2, 1)

In [67]: data = pd.Series(np.random.uniform(size=16), index=list("abcdefghijklmno
p"))

In [68]: data.plot.bar(ax=axes[0], color="black", alpha=0.7)
Out[68]: <AxesSubplot:>

In [69]: data.plot.barh(ax=axes[1], color="black", alpha=0.7)
```

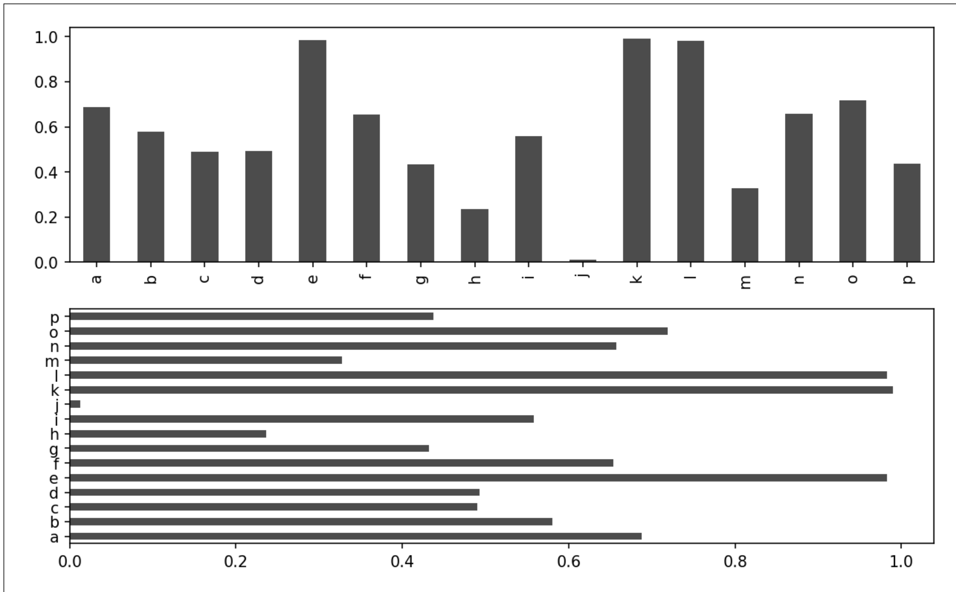


Figure 9-15. Horizonal and vertical bar plot

With a DataFrame, bar plots group the values in each row in bars, side by side, for each value. See Figure 9-16:

```
In [71]: df = pd.DataFrame(np.random.uniform(size=(6, 4)),
   ....:                   index=["one", "two", "three", "four", "five", "six"],
   ....:                   columns=pd.Index(["A", "B", "C", "D"], name="Genus"))

In [72]: df
Out[72]:
Genus         A         B         C         D
one    0.370670  0.602792  0.229159  0.486744
two    0.420082  0.571653  0.049024  0.880592
three  0.814568  0.277160  0.880316  0.431326
four   0.374020  0.899420  0.460304  0.100843
five   0.433270  0.125107  0.494675  0.961825
six    0.601648  0.478576  0.205690  0.560547

In [73]: df.plot.bar()
```

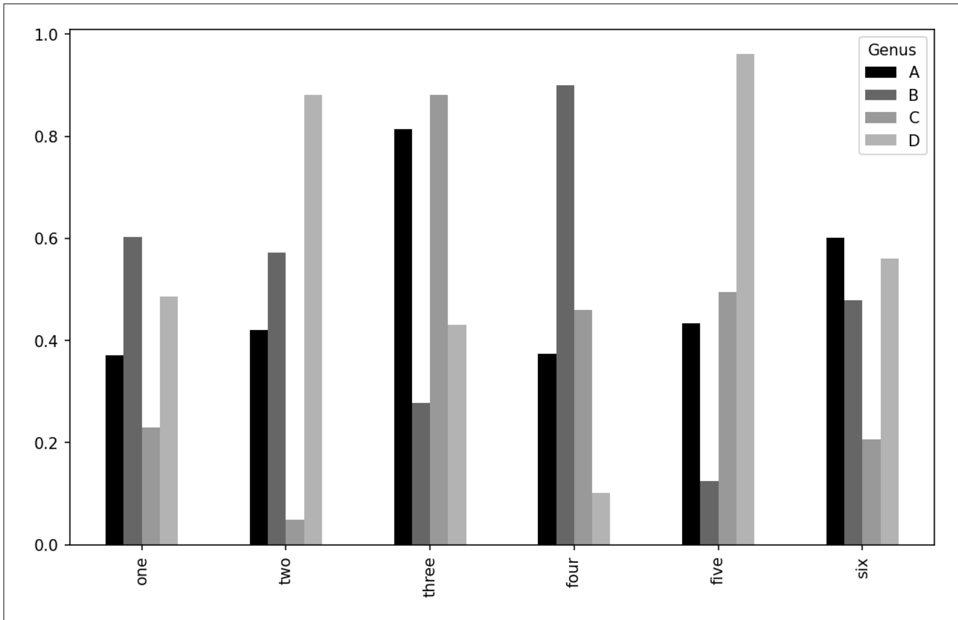


Figure 9-16. DataFrame bar plot

Note that the name "Genus" on the DataFrame's columns is used to title the legend.

We create stacked bar plots from a DataFrame by passing `stacked=True`, resulting in the value in each row being stacked together horizontally (see Figure 9-17):

```
In [75]: df.plot.barh(stacked=True, alpha=0.5)
```

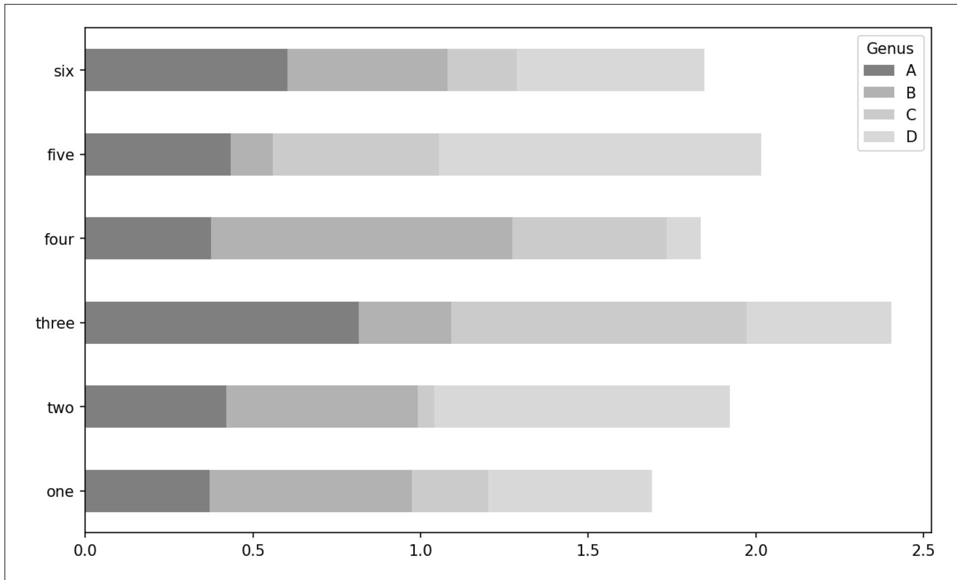
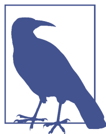


Figure 9-17. DataFrame stacked bar plot



A useful recipe for bar plots is to visualize a Series's value frequency using `value_counts`: `s.value_counts().plot.bar()`.

Let's have a look at an example dataset about restaurant tipping. Suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size for each day. I load the data using `read_csv` and make a cross-tabulation by day and party size. The `pandas.crosstab` function is a convenient way to compute a simple frequency table from two DataFrame columns:

```
In [77]: tips = pd.read_csv("examples/tips.csv")

In [78]: tips.head()
Out[78]:
   total_bill   tip smoker  day    time  size
0       16.99  1.01     No  Sun  Dinner     2
1       10.34  1.66     No  Sun  Dinner     3
2       21.01  3.50     No  Sun  Dinner     3
3       23.68  3.31     No  Sun  Dinner     2
4       24.59  3.61     No  Sun  Dinner     4
```

```
In [79]: party_counts = pd.crosstab(tips["day"], tips["size"])

In [80]: party_counts = party_counts.reindex(index=["Thur", "Fri", "Sat", "Sun"])

In [81]: party_counts
Out[81]:
size  1   2   3   4  5  6
day
Thur  1  48   4   5  1  3
Fri   1  16   1   1  0  0
Sat   2  53  18  13  1  0
Sun   0  39  15  18  3  1
```

Since there are not many one- and six-person parties, I remove them here:

```
In [82]: party_counts = party_counts.loc[:, 2:5]
```

Then, normalize so that each row sums to 1, and make the plot (see Figure 9-18):

```
# Normalize to sum to 1
In [83]: party_pcts = party_counts.div(party_counts.sum(axis="columns"),
   ....:                               axis="index")

In [84]: party_pcts
Out[84]:
size         2         3         4         5
day
Thur  0.827586  0.068966  0.086207  0.017241
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

In [85]: party_pcts.plot.bar(stacked=True)
```

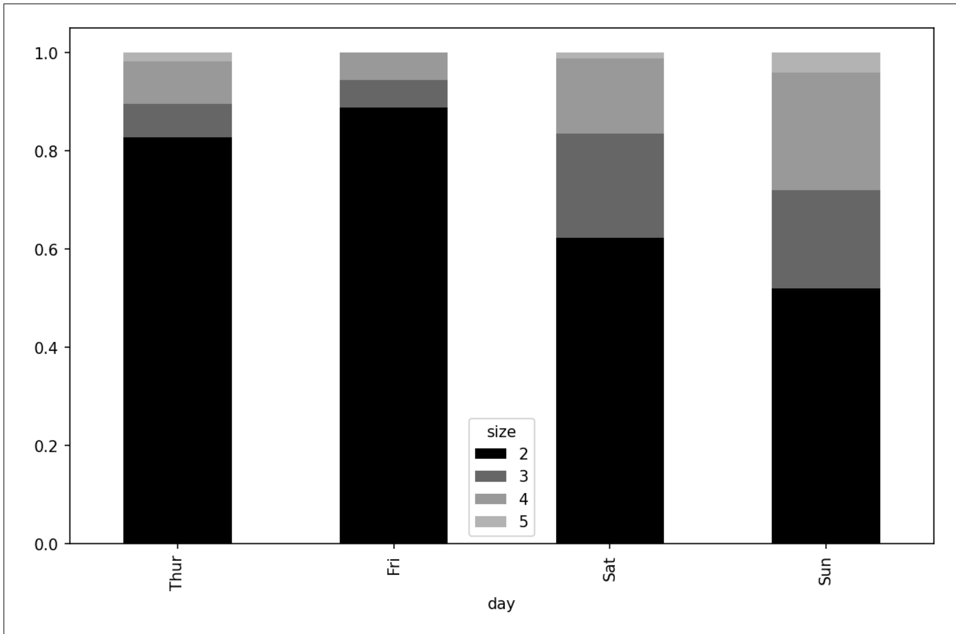


Figure 9-18. Fraction of parties by size within each day

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

With data that requires aggregation or summarization before making a plot, using the `seaborn` package can make things much simpler (install it with `conda install seaborn`). Let's look now at the tipping percentage by day with seaborn (see Figure 9-19 for the resulting plot):

```
In [87]: import seaborn as sns

In [88]: tips["tip_pct"] = tips["tip"] / (tips["total_bill"] - tips["tip"])

In [89]: tips.head()
Out[89]:
   total_bill   tip smoker  day    time  size   tip_pct
0       16.99  1.01     No  Sun  Dinner     2  0.063204
1       10.34  1.66     No  Sun  Dinner     3  0.191244
2       21.01  3.50     No  Sun  Dinner     3  0.199886
3       23.68  3.31     No  Sun  Dinner     2  0.162494
4       24.59  3.61     No  Sun  Dinner     4  0.172069

In [90]: sns.barplot(x="tip_pct", y="day", data=tips, orient="h")
```

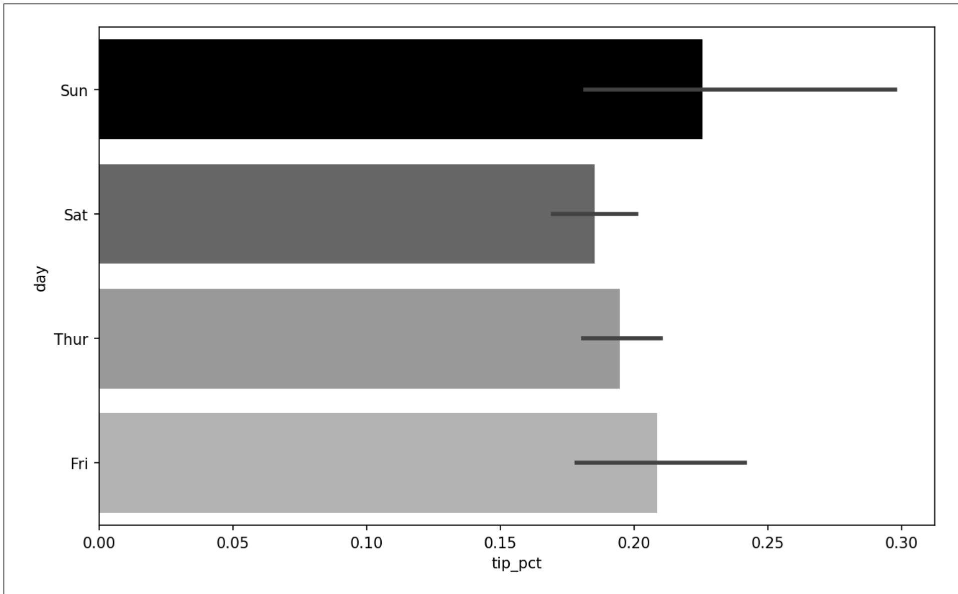


Figure 9-19. Tipping percentage by day with error bars

Plotting functions in seaborn take a `data` argument, which can be a pandas DataFrame. The other arguments refer to column names. Because there are multiple observations for each value in the `day`, the bars are the average value of `tip_pct`. The black lines drawn on the bars represent the 95% confidence interval (this can be configured through optional arguments).

`seaborn.barplot` has a `hue` option that enables us to split by an additional categorical value (see Figure 9-20):

```
In [92]: sns.barplot(x="tip_pct", y="day", hue="time", data=tips, orient="h")
```

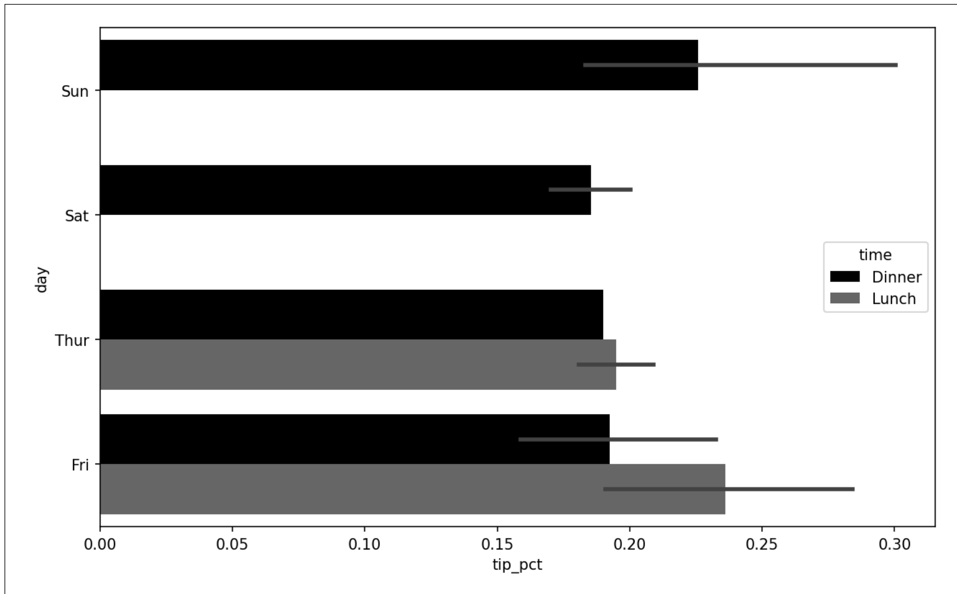


Figure 9-20. Tipping percentage by day and time

Notice that seaborn has automatically changed the aesthetics of plots: the default color palette, plot background, and grid line colors. You can switch between different plot appearances using `seaborn.set_style`:

```
In [94]: sns.set_style("whitegrid")
```

When producing plots for black-and-white print medium, you may find it useful to set a greyscale color palette, like so:

```
sns.set_palette("Greys_r")
```

Histograms and Density Plots

A *histogram* 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 `plot.hist` method on the Series (see Figure 9-21):

```
In [96]: tips["tip_pct"].plot.hist(bins=50)
```

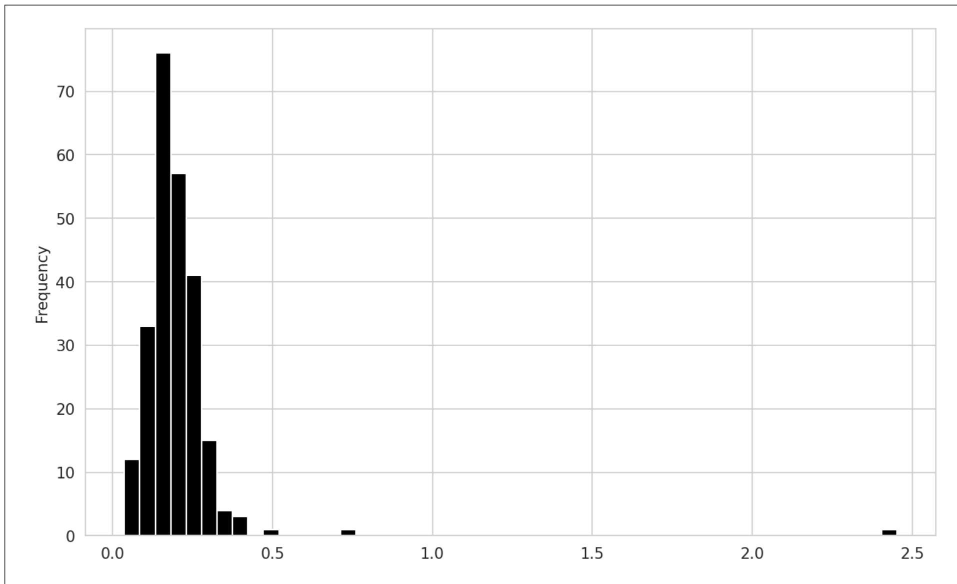


Figure 9-21. Histogram of tip percentages

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. The usual procedure is to approximate this distribution as a mixture of "kernels"—that is, simpler distributions like the normal distribution. Thus, density plots are also known as kernel density estimate (KDE) plots. Using `plot.density` makes a density plot using the conventional mixture-of-normals estimate (see Figure 9-22):

```
In [98]: tips["tip_pct"].plot.density()
```

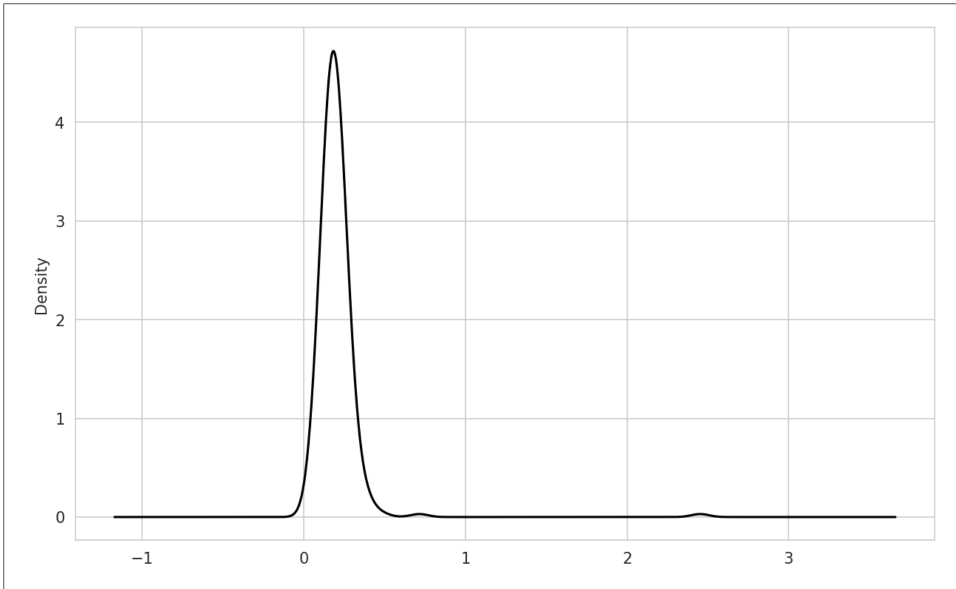


Figure 9-22. Density plot of tip percentages

This kind of plot requires SciPy, so if you do not have it installed already, you can pause and do that now:

```
conda install scipy
```

seaborn makes histograms and density plots even easier through its `histplot` method, which can plot both a histogram and a continuous density estimate simultaneously. As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions (see Figure 9-23):

```
In [100]: comp1 = np.random.standard_normal(200)

In [101]: comp2 = 10 + 2 * np.random.standard_normal(200)

In [102]: values = pd.Series(np.concatenate([comp1, comp2]))

In [103]: sns.histplot(values, bins=100, color="black")
```

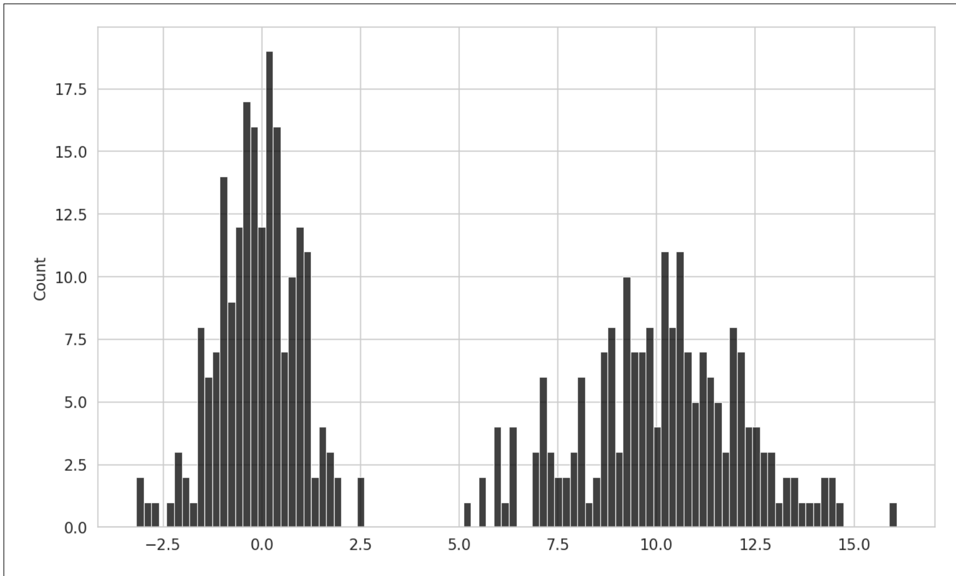


Figure 9-23. Normalized histogram of normal mixture

Scatter or Point Plots

Point plots or scatter plots can be a useful way of examining the relationship between two one-dimensional data series. For example, here we load the `macrodata` dataset from the statsmodels project, select a few variables, then compute log differences:

```
In [104]: macro = pd.read_csv("examples/macrodata.csv")

In [105]: data = macro[["cpi", "m1", "tbilrate", "unemp"]]

In [106]: trans_data = np.log(data).diff().dropna()

In [107]: trans_data.tail()
Out[107]:
          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
```

We can then use seaborn's `regplot` method, which makes a scatter plot and fits a linear regression line (see Figure 9-24):

```
In [109]: ax = sns.regplot(x="m1", y="unemp", data=trans_data)

In [110]: ax.title("Changes in log(m1) versus log(unemp)")
```

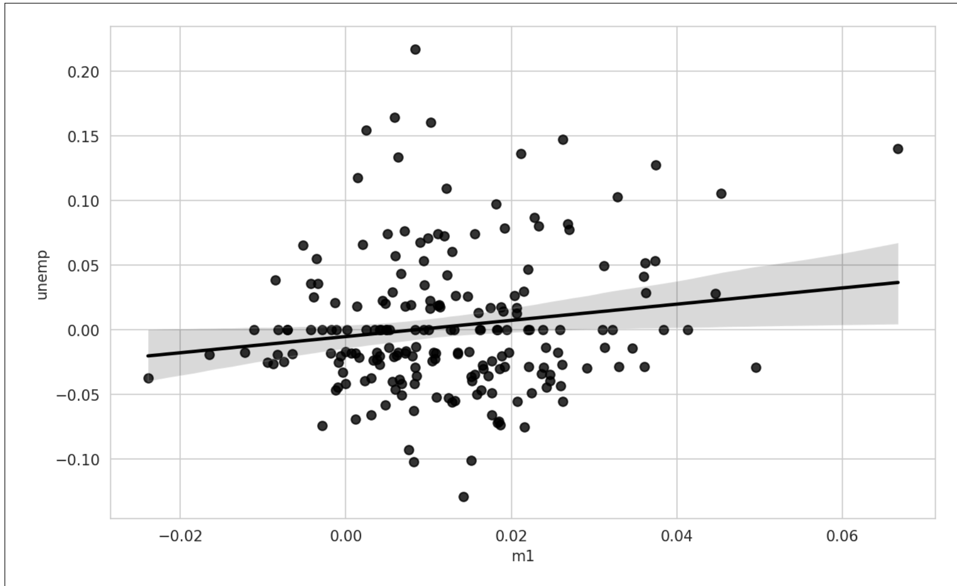


Figure 9-24. A seaborn regression/scatter plot

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 seaborn has a convenient `pairplot` function that supports placing histograms or density estimates of each variable along the diagonal (see Figure 9-25 for the resulting plot):

```
In [111]: sns.pairplot(trans_data, diag_kind="kde", plot_kws={"alpha": 0.2})
```

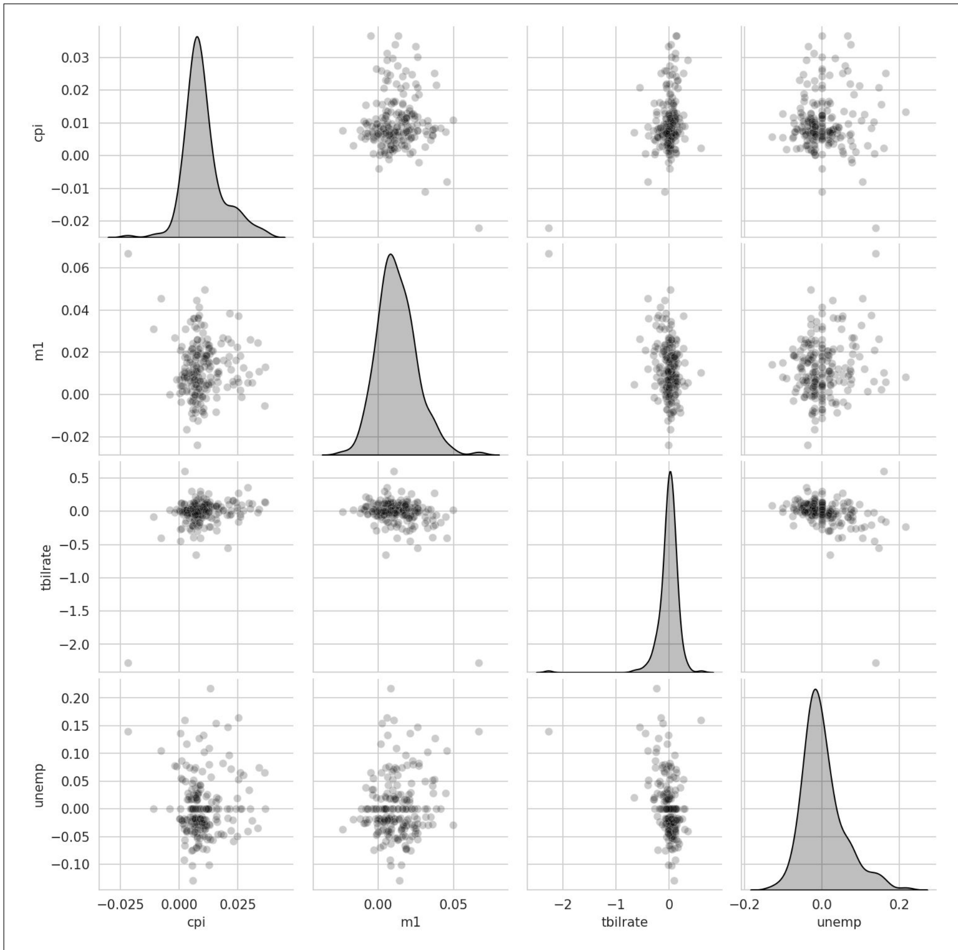


Figure 9-25. Pair plot matrix of statsmodels macro data

You may notice the `plot_kws` argument. This enables us to pass down configuration options to the individual plotting calls on the off-diagonal elements. Check out the `seaborn.pairplot` docstring for more granular configuration options.

Facet Grids and Categorical Data

What about datasets where we have additional grouping dimensions? One way to visualize data with many categorical variables is to use a *facet grid*, which is a two-dimensional layout of plots where the data is split across the plots on each axis based on the distinct values of a certain variable. seaborn has a useful built-in function `catplot` that simplifies making many kinds of faceted plots split by categorical variables (see Figure 9-26 for the resulting plot):

```
In [112]: sns.catplot(x="day", y="tip_pct", hue="time", col="smoker",
   .....:             kind="bar", data=tips[tips.tip_pct < 1])
```

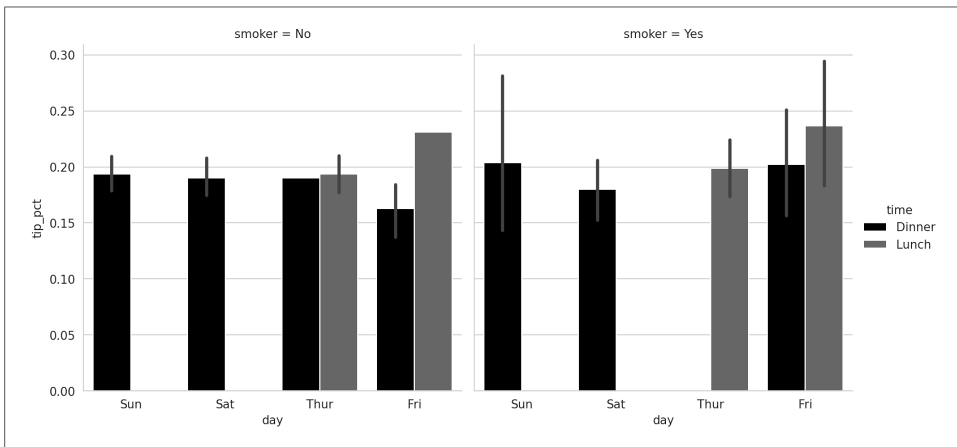


Figure 9-26. Tipping percentage by day/time/smoker

Instead of grouping by `"time"` by different bar colors within a facet, we can also expand the facet grid by adding one row per `time` value (see Figure 9-27):

```
In [113]: sns.catplot(x="day", y="tip_pct", row="time",
   .....:             col="smoker",
   .....:             kind="bar", data=tips[tips.tip_pct < 1])
```

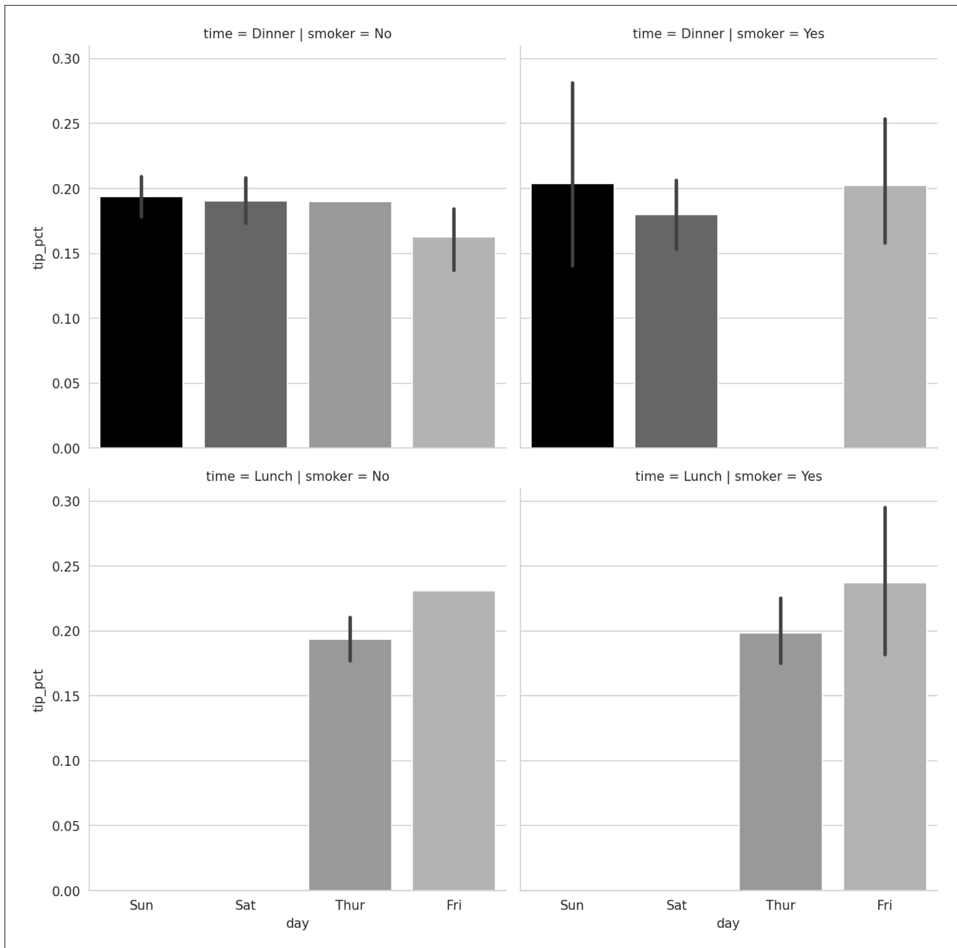


Figure 9-27. Tipping percentage by day split by time/smoker

`catplot` supports other plot types that may be useful depending on what you are trying to display. For example, *box plots* (which show the median, quartiles, and outliers) can be an effective visualization type (see Figure 9-28):

```
In [114]: sns.catplot(x="tip_pct", y="day", kind="box",
   .....:             data=tips[tips.tip_pct < 0.5])
```

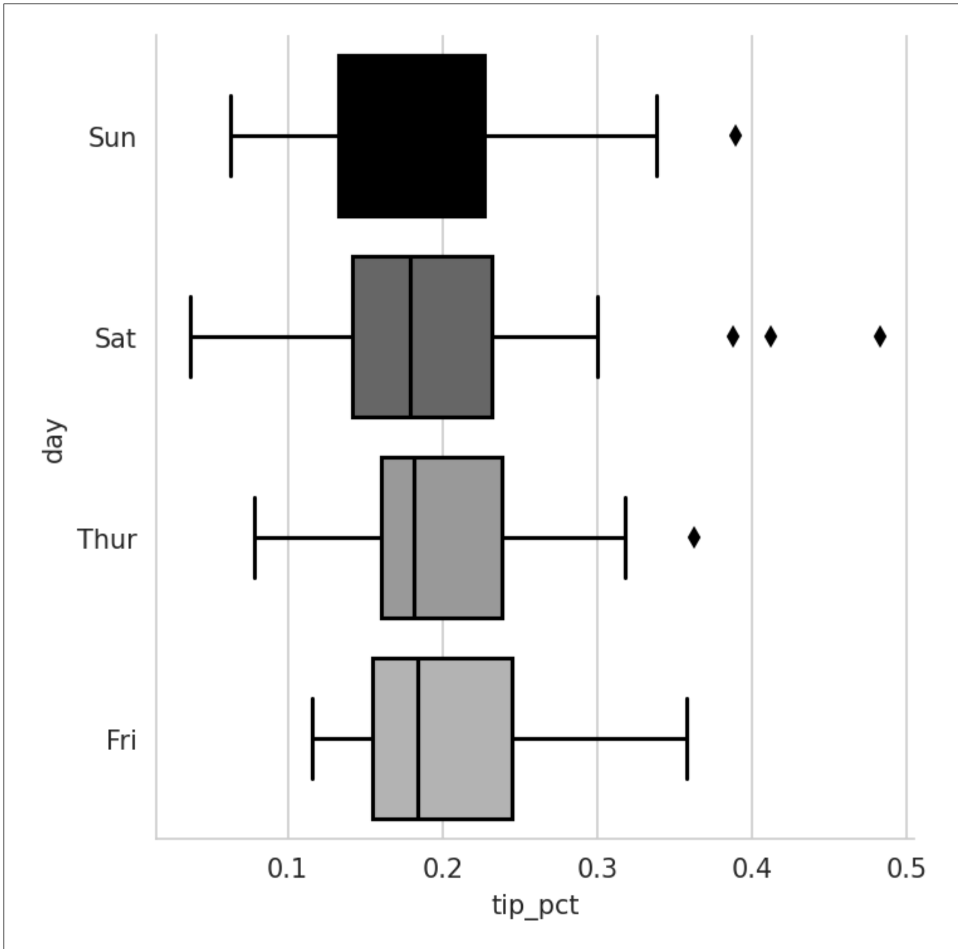


Figure 9-28. Box plot of tipping percentage by day

You can create your own facet grid plots using the more general `seaborn.FacetGrid` class. See the seaborn documentation for more.

9.3 Other Python Visualization Tools

As is common with open source, there many options for creating graphics in Python (too many to list). Since 2010, much development effort has been focused on creating interactive graphics for publication on the web. With tools like Altair, Bokeh, and Plotly, it's now possible to specify dynamic, interactive graphics in Python that are intended for use with web browsers.

For creating static graphics for print or web, I recommend using matplotlib and libraries that build on matplotlib, like pandas and seaborn, for your needs. For other data visualization requirements, it may be useful to learn how to use one of the other available tools. I encourage you to explore the ecosystem as it continues to evolve and innovate into the future.

An excellent book on data visualization is *Fundamentals of Data Visualization* by Claus O. Wilke (O'Reilly), which is available in print or on Claus's website at *https://clauswilke.com/dataviz*.

9.4 Conclusion

The goal of this chapter was to get your feet wet with some basic data visualization using pandas, matplotlib, and seaborn. If visually communicating the results of data analysis is important in your work, I encourage you to seek out resources to learn more about effective data visualization. It is an active field of research, and you can practice with many excellent learning resources available online and in print.

In the next chapter, we turn our attention to data aggregation and group operations with pandas.

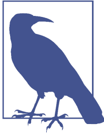
CHAPTER 10

Data Aggregation and Group Operations

Categorizing a dataset and applying a function to each group, whether an aggregation or transformation, can be a critical component of a data analysis workflow. After loading, merging, and preparing a dataset, you may need to compute group statistics or possibly *pivot tables* for reporting or visualization purposes. pandas provides a versatile `groupby` interface, enabling you to slice, dice, and summarize datasets in a natural way.

One reason for the popularity of relational databases and SQL (which stands for "structured query language") is the ease with which data can be joined, filtered, transformed, and aggregated. However, query languages like SQL impose certain limitations on the kinds of group operations that can be performed. As you will see, with the expressiveness of Python and pandas, we can perform quite complex group operations by expressing them as custom Python functions that manipulate the data associated with each group. In this chapter, you will learn how to:

- Split a pandas object into pieces using one or more keys (in the form of functions, arrays, or DataFrame column names)
- Calculate group summary statistics, like count, mean, or standard deviation, or a user-defined function
- Apply within-group transformations or other manipulations, like normalization, linear regression, rank, or subset selection
- Compute pivot tables and cross-tabulations
- Perform quantile analysis and other statistical group analyses



Time-based aggregation of time series data, a special use case of groupby, is referred to as *resampling* in this book and will receive separate treatment in Chapter 11.

As with the rest of the chapters, we start by importing NumPy and pandas:

```
In [12]: import numpy as np

In [13]: import pandas as pd
```

10.1 How to Think About Group Operations

Hadley Wickham, an author of many popular packages for the R programming language, coined the term *split-apply-combine* for describing group operations. In the first stage of the process, data contained in a pandas object, whether a Series, DataFrame, or otherwise, is *split* into groups based on one or more *keys* that you provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (axis="index") or its columns (axis="columns"). Once this is done, a function is *applied* to each group, producing a new value. Finally, the results of all those function applications are *combined* into a result object. The form of the resulting object will usually depend on what's being done to the data. See Figure 10-1 for a mockup of a simple group aggregation.

Each grouping key can take many forms, and the keys do not have to be all of the same type:

- A list or array of values that is the same length as the axis being grouped
- A value indicating a column name in a DataFrame
- A dictionary or Series giving a correspondence between the values on the axis being grouped and the group names
- A function to be invoked on the axis index or the individual labels in the index

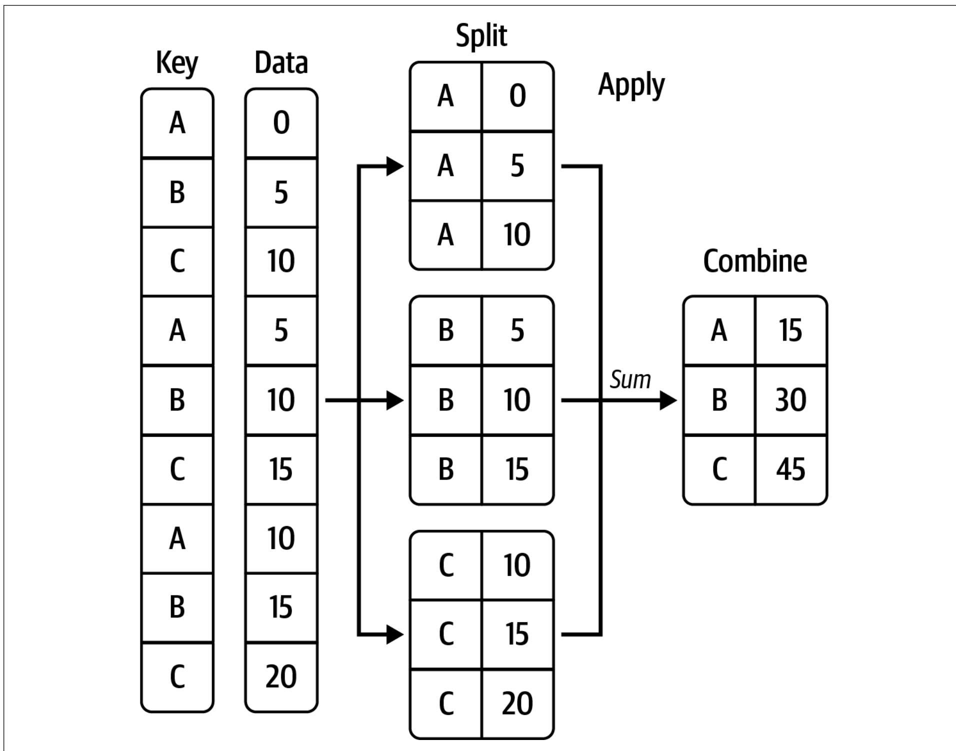


Figure 10-1. Illustration of a group aggregation

Note that the latter three methods are shortcuts for producing an array of values to be used to split up the object. Don't worry if this all seems abstract. Throughout this chapter, I will give many examples of all these methods. To get started, here is a small tabular dataset as a DataFrame:

```
In [14]: df = pd.DataFrame({"key1" : ["a", "a", None, "b", "b", "a", None],
   ....:                    "key2" : pd.Series([1, 2, 1, 2, 1, None, 1], dtype="I
nt64"),
   ....:                    "data1" : np.random.standard_normal(7),
   ....:                    "data2" : np.random.standard_normal(7)})

In [15]: df
Out[15]:
   key1  key2     data1     data2
0     a     1 -0.204708  0.281746
1     a     2  0.478943  0.769023
2  None     1 -0.519439  1.246435
3     b     2 -0.555730  1.007189
4     b     1  1.965781 -1.296221
5     a  <NA>  1.393406  0.274992
6  None     1  0.092908  0.228913
```

Suppose you wanted to compute the mean of the `data1` column using the labels from `key1`. There are a number of ways to do this. One is to access `data1` and call `groupby` with the column (a Series) at `key1`:

```
In [16]: grouped = df["data1"].groupby(df["key1"])

In [17]: grouped
Out[17]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7fa9270e0a00>
```

This `grouped` variable is now a special *"GroupBy"* object. It has not actually computed anything yet except for some intermediate data about the group key `df["key1"]`. The idea is that this object has all of the information needed to then apply some operation to each of the groups. For example, to compute group means we can call the GroupBy's `mean` method:

```
In [18]: grouped.mean()
Out[18]:
key1
a    0.555881
b    0.705025
Name: data1, dtype: float64
```

Later in Section 10.2, "Data Aggregation," on page 329, I'll explain more about what happens when you call `.mean()`. The important thing here is that the data (a Series) has been aggregated by splitting the data on the group key, producing a new Series that is now indexed by the unique values in the `key1` column. The result index has the name `"key1"` because the DataFrame column `df["key1"]` did.

If instead we had passed multiple arrays as a list, we'd get something different:

```
In [19]: means = df["data1"].groupby([df["key1"], df["key2"]]).mean()

In [20]: means
Out[20]:
key1  key2
a     1      -0.204708
      2       0.478943
b     1       1.965781
      2      -0.555730
Name: data1, dtype: float64
```

Here we grouped the data using two keys, and the resulting Series now has a hierarchical index consisting of the unique pairs of keys observed:

```
In [21]: means.unstack()
Out[21]:
key2         1         2
key1
a    -0.204708  0.478943
b     1.965781 -0.555730
```

In this example, the group keys are all Series, though they could be any arrays of the right length:

```
In [22]: states = np.array(["OH", "CA", "CA", "OH", "OH", "CA", "OH"])

In [23]: years = [2005, 2005, 2006, 2005, 2006, 2005, 2006]

In [24]: df["data1"].groupby([states, years]).mean()
Out[24]:
CA  2005    0.936175
    2006   -0.519439
OH  2005   -0.380219
    2006    1.029344
Name: data1, dtype: float64
```

Frequently, the grouping information is found in the same DataFrame as the data you want to work on. In that case, you can pass column names (whether those are strings, numbers, or other Python objects) as the group keys:

```
In [25]: df.groupby("key1").mean()
Out[25]:
      key2     data1     data2
key1
a      1.5  0.555881  0.441920
b      1.5  0.705025 -0.144516

In [26]: df.groupby("key2").mean()
Out[26]:
         data1     data2
key2
1     0.333636  0.115218
2    -0.038393  0.888106

In [27]: df.groupby(["key1", "key2"]).mean()
Out[27]:
              data1     data2
key1 key2
a    1    -0.204708  0.281746
     2     0.478943  0.769023
b    1     1.965781 -1.296221
     2    -0.555730  1.007189
```

You may have noticed in the second case, `df.groupby("key2").mean()`, that there is no `key1` column in the result. Because `df["key1"]` is not numeric data, it is said to be a *nuisance column*, which is therefore automatically excluded from the result. By default, all of the numeric columns are aggregated, though it is possible to filter down to a subset, as you'll see soon.

Regardless of the objective in using `groupby`, a generally useful GroupBy method is `size`, which returns a Series containing group sizes:

```
In [28]: df.groupby(["key1", "key2"]).size()
Out[28]:
key1  key2
a     1       1
      2       1
b     1       1
      2       1
dtype: int64
```

Note that any missing values in a group key are excluded from the result by default. This behavior can be disabled by passing `dropna=False` to `groupby`:

```
In [29]: df.groupby("key1", dropna=False).size()
Out[29]:
key1
a      3
b      2
NaN    2
dtype: int64

In [30]: df.groupby(["key1", "key2"], dropna=False).size()
Out[30]:
key1  key2
a     1       1
      2       1
      <NA>    1
b     1       1
      2       1
NaN   1       2
dtype: int64
```

A group function similar in spirit to `size` is count, which computes the number of nonnull values in each group:

```
In [31]: df.groupby("key1").count()
Out[31]:
      key2  data1  data2
key1
a        2      3      3
b        2      2      2
```

Iterating over Groups

The object returned by `groupby` supports iteration, generating a sequence of 2-tuples containing the group name along with the chunk of data. Consider the following:

```
In [32]: for name, group in df.groupby("key1"):
   ....:     print(name)
   ....:     print(group)
   ....:
a
  key1  key2     data1     data2
0    a     1 -0.204708  0.281746
```

```
1    a     2  0.478943  0.769023
5    a  <NA>  1.393406  0.274992
b
  key1  key2     data1     data2
3    b     2 -0.555730  1.007189
4    b     1  1.965781 -1.296221
```

In the case of multiple keys, the first element in the tuple will be a tuple of key values:

```
In [33]: for (k1, k2), group in df.groupby(["key1", "key2"]):
   ....:     print((k1, k2))
   ....:     print(group)
   ....:
('a', 1)
  key1  key2     data1     data2
0    a     1 -0.204708  0.281746
('a', 2)
  key1  key2     data1     data2
1    a     2  0.478943  0.769023
('b', 1)
  key1  key2     data1     data2
4    b     1  1.965781 -1.296221
('b', 2)
  key1  key2    data1     data2
3    b     2 -0.55573  1.007189
```

Of course, you can choose to do whatever you want with the pieces of data. A recipe you may find useful is computing a dictionary of the data pieces as a one-liner:

```
In [34]: pieces = {name: group for name, group in df.groupby("key1")}

In [35]: pieces["b"]
Out[35]:
  key1  key2     data1     data2
3    b     2 -0.555730  1.007189
4    b     1  1.965781 -1.296221
```

By default `groupby` groups on `axis="index"`, but you can group on any of the other axes. For example, we could group the columns of our example `df` here by whether they start with `"key"` or `"data"`:

```
In [36]: grouped = df.groupby({"key1": "key", "key2": "key",
   ....:                       "data1": "data", "data2": "data"}, axis="columns")
```

We can print out the groups like so:

```
In [37]: for group_key, group_values in grouped:
   ....:     print(group_key)
   ....:     print(group_values)
   ....:
data
      data1     data2
0 -0.204708  0.281746
1  0.478943  0.769023
```

```
2 -0.519439  1.246435
3 -0.555730  1.007189
4  1.965781 -1.296221
5  1.393406  0.274992
6  0.092908  0.228913
key
   key1  key2
0     a     1
1     a     2
2  None     1
3     b     2
4     b     1
5     a  <NA>
6  None     1
```

Selecting a Column or Subset of Columns

Indexing a GroupBy object created from a DataFrame with a column name or array of column names has the effect of column subsetting for aggregation. This means that:

```
df.groupby("key1")["data1"]
df.groupby("key1")[["data2"]]
```

are conveniences for:

```
df["data1"].groupby(df["key1"])
df[["data2"]].groupby(df["key1"])
```

Especially for large datasets, it may be desirable to aggregate only a few columns. For example, in the preceding dataset, to compute the means for just the `data2` column and get the result as a DataFrame, we could write:

```
In [38]: df.groupby(["key1", "key2"])[["data2"]].mean()
Out[38]:
              data2
key1 key2
a    1     0.281746
     2     0.769023
b    1    -1.296221
     2     1.007189
```

The object returned by this indexing operation is a grouped DataFrame if a list or array is passed, or a grouped Series if only a single column name is passed as a scalar:

```
In [39]: s_grouped = df.groupby(["key1", "key2"])["data2"]

In [40]: s_grouped
Out[40]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7fa9270e3520>

In [41]: s_grouped.mean()
Out[41]:
key1  key2
```

```
a     1       0.281746
      2       0.769023
b     1      -1.296221
      2       1.007189
Name: data2, dtype: float64
```

Grouping with Dictionaries and Series

Grouping information may exist in a form other than an array. Let's consider another example DataFrame:

```
In [42]: people = pd.DataFrame(np.random.standard_normal((5, 5)),
   ....:                       columns=["a", "b", "c", "d", "e"],
   ....:                       index=["Joe", "Steve", "Wanda", "Jill", "Trey"])

In [43]: people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values

In [44]: people
Out[44]: 
              a         b         c         d         e
Joe    1.352917  0.886429 -2.001637 -0.371843  1.669025
Steve -0.438570 -0.539741  0.476985  3.248944 -1.021228
Wanda -0.577087       NaN       NaN  0.523772  0.000940
Jill   1.343810 -0.713544 -0.831154 -2.370232 -1.860761
Trey  -0.860757  0.560145 -1.265934  0.119827 -1.063512
```

Now, suppose I have a group correspondence for the columns and want to sum the columns by group:

```
In [45]: mapping = {"a": "red", "b": "red", "c": "blue",
   ....:            "d": "blue", "e": "red", "f" : "orange"}
```

Now, you could construct an array from this dictionary to pass to `groupby`, but instead we can just pass the dictionary (I included the key `"f"` to highlight that unused grouping keys are OK):

```
In [46]: by_column = people.groupby(mapping, axis="columns")

In [47]: by_column.sum()
Out[47]: 
           blue       red
Joe   -2.373480  3.908371
Steve  3.725929 -1.999539
Wanda  0.523772 -0.576147
Jill  -3.201385 -1.230495
Trey  -1.146107 -1.364125
```

The same functionality holds for Series, which can be viewed as a fixed-size mapping:

```
In [48]: map_series = pd.Series(mapping)

In [49]: map_series
Out[49]: 
```

```
a        red
b        red
c       blue
d       blue
e        red
f     orange
dtype: object

In [50]: people.groupby(map_series, axis="columns").count()
Out[50]: 
       blue  red
Joe       2    3
Steve     2    3
Wanda     1    2
Jill      2    3
Trey      2    3
```

Grouping with Functions

Using Python functions is a more generic way of defining a group mapping compared with a dictionary or Series. Any function passed as a group key will be called once per index value (or once per column value if using `axis="columns"`), with the return values being used as the group names. More concretely, consider the example DataFrame from the previous section, which has people's first names as index values. Suppose you wanted to group by name length. While you could compute an array of string lengths, it's simpler to just pass the `len` function:

```
In [51]: people.groupby(len).sum()
Out[51]: 
          a         b         c         d         e
3  1.352917  0.886429 -2.001637 -0.371843  1.669025
4  0.483052 -0.153399 -2.097088 -2.250405 -2.924273
5 -1.015657 -0.539741  0.476985  3.772716 -1.020287
```

Mixing functions with arrays, dictionaries, or Series is not a problem, as everything gets converted to arrays internally:

```
In [52]: key_list = ["one", "one", "one", "two", "two"]

In [53]: people.groupby([len, key_list]).min()
Out[53]: 
              a         b         c         d         e
3 one  1.352917  0.886429 -2.001637 -0.371843  1.669025
4 two -0.860757 -0.713544 -1.265934 -2.370232 -1.860761
5 one -0.577087 -0.539741  0.476985  0.523772 -1.021228
```

Grouping by Index Levels

A final convenience for hierarchically indexed datasets is the ability to aggregate using one of the levels of an axis index. Let's look at an example:

```
In [54]: columns = pd.MultiIndex.from_arrays([["US", "US", "US", "JP", "JP"],
   ....:                                     [1, 3, 5, 1, 3]],
   ....:                                     names=["cty", "tenor"])

In [55]: hier_df = pd.DataFrame(np.random.standard_normal((4, 5)), columns=column
s)

In [56]: hier_df
Out[56]:
cty          US                            JP          
tenor         1         3         5         1         3
0      0.332883 -2.359419 -0.199543 -1.541996 -0.970736
1     -1.307030  0.286350  0.377984 -0.753887  0.331286
2      1.349742  0.069877  0.246674 -0.011862  1.004812
3      1.327195 -0.919262 -1.549106  0.022185  0.758363
```

To group by level, pass the level number or name using the `level` keyword:

```
In [57]: hier_df.groupby(level="cty", axis="columns").count()
Out[57]:
cty  JP  US
0     2   3
1     2   3
2     2   3
3     2   3
```

10.2 Data Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays. The preceding examples have used several of them, including `mean`, `count`, `min`, and `sum`. You may wonder what is going on when you invoke `mean()` on a GroupBy object. Many common aggregations, such as those found in Table 10-1, have optimized implementations. However, you are not limited to only this set of methods.

Table 10-1. Optimized `groupby` *methods*

Function name	Description
`any, all`	Return `True` if any (one or more values) or all non-NA values are "truthy"
`count`	Number of non-NA values
`cummin, cummax`	Cumulative minimum and maximum of non-NA values
`cumsum`	Cumulative sum of non-NA values
`cumprod`	Cumulative product of non-NA values
`first, last`	First and last non-NA values
`mean`	Mean of non-NA values
`median`	Arithmetic median of non-NA values
`min, max`	Minimum and maximum of non-NA values
`nth`	Retrieve value that would appear at position `n` with the data in sorted order
`ohlc`	Compute four "open-high-low-close" statistics for time series-like data

Function name	Description
`prod`	Product of non-NA values
`quantile`	Compute sample quantile
`rank`	Ordinal ranks of non-NA values, like calling `Series.rank`
`size`	Compute group sizes, returning result as a Series
`sum`	Sum of non-NA values
`std, var`	Sample standard deviation and variance

You can use aggregations of your own devising and additionally call any method that is also defined on the object being grouped. For example, the `nsmallest` Series method selects the smallest requested number of values from the data. While `nsmallest` is not explicitly implemented for GroupBy, we can still use it with a nonoptimized implementation. Internally, GroupBy slices up the Series, calls `piece.nsmallest(n)` for each piece, and then assembles those results into the result object:

```
In [58]: df
Out[58]:
   key1  key2     data1     data2
0     a     1 -0.204708  0.281746
1     a     2  0.478943  0.769023
2  None     1 -0.519439  1.246435
3     b     2 -0.555730  1.007189
4     b     1  1.965781 -1.296221
5     a  <NA>  1.393406  0.274992
6  None     1  0.092908  0.228913

In [59]: grouped = df.groupby("key1")

In [60]: grouped["data1"].nsmallest(2)
Out[60]:
key1
a     0   -0.204708
      1    0.478943
b     3   -0.555730
      4    1.965781
Name: data1, dtype: float64
```

To use your own aggregation functions, pass any function that aggregates an array to the `aggregate` method or its short alias `agg`:

```
In [61]: def peak_to_peak(arr):
   ....:     return arr.max() - arr.min()

In [62]: grouped.agg(peak_to_peak)
Out[62]:
      key2     data1     data2
key1
```

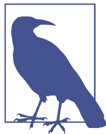
```
a          1  1.598113  0.494031
b          1  2.521511  2.303410
```

You may notice that some methods, like `describe`, also work, even though they are not aggregations, strictly speaking:

```
In [63]: grouped.describe()
Out[63]:
      key2                                          data1             ...  \
     count mean       std  min   25%  50%   75%  max count      mean  ...
key1                                                                  ...
a      2.0  1.5  0.707107  1.0  1.25  1.5  1.75  2.0   3.0  0.555881  ...
b      2.0  1.5  0.707107  1.0  1.25  1.5  1.75  2.0   2.0  0.705025  ...
                                 data2                                   \
           75%       max count      mean       std       min       25%
key1
a     0.936175  1.393406   3.0  0.441920  0.283299  0.274992  0.278369
b     1.335403  1.965781   2.0 -0.144516  1.628757 -1.296221 -0.720368

           50%       75%       max
key1
a     0.281746  0.525384  0.769023
b    -0.144516  0.431337  1.007189
[2 rows x 24 columns]
```

I will explain in more detail what has happened here in Section 10.3, “Apply: General split-apply-combine,” on page 335.



Custom aggregation functions are generally much slower than the optimized functions found in Table 10-1. This is because there is some extra overhead (function calls, data rearrangement) in constructing the intermediate group data chunks.

Column-Wise and Multiple Function Application

Let’s return to the tipping dataset used in the last chapter. After loading it with `pandas.read_csv`, we add a tipping percentage column:

```
In [64]: tips = pd.read_csv("examples/tips.csv")

In [65]: tips.head()
Out[65]:
   total_bill   tip smoker  day    time  size
0       16.99  1.01     No  Sun  Dinner     2
1       10.34  1.66     No  Sun  Dinner     3
2       21.01  3.50     No  Sun  Dinner     3
3       23.68  3.31     No  Sun  Dinner     2
4       24.59  3.61     No  Sun  Dinner     4
```

Now I will add a `tip_pct` column with the tip percentage of the total bill:

```
In [66]: tips["tip_pct"] = tips["tip"] / tips["total_bill"]

In [67]: tips.head()
Out[67]:
   total_bill   tip smoker  day    time  size   tip_pct
0       16.99  1.01     No  Sun  Dinner     2  0.059447
1       10.34  1.66     No  Sun  Dinner     3  0.160542
2       21.01  3.50     No  Sun  Dinner     3  0.166587
3       23.68  3.31     No  Sun  Dinner     2  0.139780
4       24.59  3.61     No  Sun  Dinner     4  0.146808
```

As you've already seen, aggregating a Series or all of the columns of a DataFrame is a matter of using `aggregate` (or `agg`) with the desired function or calling a method like `mean` or `std`. However, you may want to aggregate using a different function, depending on the column, or multiple functions at once. Fortunately, this is possible to do, which I'll illustrate through a number of examples. First, I'll group the `tips` by `day` and `smoker`:

```
In [68]: grouped = tips.groupby(["day", "smoker"])
```

Note that for descriptive statistics like those in Table 10-1, you can pass the name of the function as a string:

```
In [69]: grouped_pct = grouped["tip_pct"]

In [70]: grouped_pct.agg("mean")
Out[70]:
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

If you pass a list of functions or function names instead, you get back a DataFrame with column names taken from the functions:

```
In [71]: grouped_pct.agg(["mean", "std", peak_to_peak])
Out[71]:
                 mean       std  peak_to_peak
day  smoker
Fri  No      0.151650  0.028123      0.067349
     Yes     0.174783  0.051293      0.159925
Sat  No      0.158048  0.039767      0.235193
     Yes     0.147906  0.061375      0.290095
Sun  No      0.160113  0.042347      0.193226
```

```
     Yes       0.187250  0.154134       0.644685
Thur No        0.160298  0.038774       0.193350
     Yes       0.163863  0.039389       0.151240
```

Here we passed a list of aggregation functions to agg to evaluate independently on the data groups.

You don't need to accept the names that GroupBy gives to the columns; notably, lambda functions have the name "<lambda>", which makes them hard to identify (you can see for yourself by looking at a function's __name__ attribute). Thus, if you pass a list of (name, function) tuples, the first element of each tuple will be used as the DataFrame column names (you can think of a list of 2-tuples as an ordered mapping):

```
In [72]: grouped_pct.agg([("average", "mean"), ("stdev", np.std)])
Out[72]:
              average     stdev
day  smoker
Fri  No      0.151650  0.028123
     Yes     0.174783  0.051293
Sat  No      0.158048  0.039767
     Yes     0.147906  0.061375
Sun  No      0.160113  0.042347
     Yes     0.187250  0.154134
Thur No      0.160298  0.038774
     Yes     0.163863  0.039389
```

With a DataFrame you have more options, as you can specify a list of functions to apply to all of the columns or different functions per column. To start, suppose we wanted to compute the same three statistics for the tip_pct and total_bill columns:

```
In [73]: functions = ["count", "mean", "max"]

In [74]: result = grouped[["tip_pct", "total_bill"]].agg(functions)

In [75]: result
Out[75]:
            tip_pct                     total_bill
              count      mean       max      count       mean    max
day  smoker
Fri  No           4  0.151650  0.187735          4  18.420000  22.75
     Yes         15  0.174783  0.263480         15  16.813333  40.17
Sat  No          45  0.158048  0.291990         45  19.661778  48.33
     Yes         42  0.147906  0.325733         42  21.276667  50.81
Sun  No          57  0.160113  0.252672         57  20.506667  48.17
     Yes         19  0.187250  0.710345         19  24.120000  45.35
Thur No          45  0.160298  0.266312         45  17.113111  41.19
     Yes         17  0.163863  0.241255         17  19.190588  43.11
```

As you can see, the resulting DataFrame has hierarchical columns, the same as you would get aggregating each column separately and using `concat` to glue the results together using the column names as the `keys` argument:

```
In [76]: result["tip_pct"]
Out[76]:
             count      mean       max
day  smoker
Fri  No          4  0.151650  0.187735
     Yes        15  0.174783  0.263480
Sat  No         45  0.158048  0.291990
     Yes        42  0.147906  0.325733
Sun  No         57  0.160113  0.252672
     Yes        19  0.187250  0.710345
Thur No         45  0.160298  0.266312
     Yes        17  0.163863  0.241255
```

As before, a list of tuples with custom names can be passed:

```
In [77]: ftuples = [("Average", "mean"), ("Variance", np.var)]

In [78]: grouped[["tip_pct", "total_bill"]].agg(ftuples)
Out[78]:
              tip_pct           total_bill
              Average  Variance    Average    Variance
day  smoker
Fri  No      0.151650  0.000791  18.420000   25.596333
     Yes     0.174783  0.002631  16.813333   82.562438
Sat  No      0.158048  0.001581  19.661778   79.908965
     Yes     0.147906  0.003767  21.276667  101.387535
Sun  No      0.160113  0.001793  20.506667   66.099980
     Yes     0.187250  0.023757  24.120000  109.046044
Thur No      0.160298  0.001503  17.113111   59.625081
     Yes     0.163863  0.001551  19.190588   69.808518
```

Now, suppose you wanted to apply potentially different functions to one or more of the columns. To do this, pass a dictionary to `agg` that contains a mapping of column names to any of the function specifications listed so far:

```
In [79]: grouped.agg({"tip" : np.max, "size" : "sum"})
Out[79]:
               tip  size
day  smoker
Fri  No       3.50     9
     Yes      4.73    31
Sat  No       9.00   115
     Yes     10.00   104
Sun  No       6.00   167
     Yes      6.50    49
Thur No       6.70   112
     Yes      5.00    40

In [80]: grouped.agg({"tip_pct" : ["min", "max", "mean", "std"],
```

```
   ....:                   "size" : "sum"})
Out[80]:
              tip_pct                                  size
                  min       max      mean       std   sum
day  smoker
Fri  No      0.120385  0.187735  0.151650  0.028123     9
     Yes     0.103555  0.263480  0.174783  0.051293    31
Sat  No      0.056797  0.291990  0.158048  0.039767   115
     Yes     0.035638  0.325733  0.147906  0.061375   104
Sun  No      0.059447  0.252672  0.160113  0.042347   167
     Yes     0.065660  0.710345  0.187250  0.154134    49
Thur No      0.072961  0.266312  0.160298  0.038774   112
     Yes     0.090014  0.241255  0.163863  0.039389    40
```

A DataFrame will have hierarchical columns only if multiple functions are applied to at least one column.

Returning Aggregated Data Without Row Indexes

In all of the examples up until now, the aggregated data comes back with an index, potentially hierarchical, composed from the unique group key combinations. Since this isn't always desirable, you can disable this behavior in most cases by passing `as_index=False` to `groupby`:

```
In [81]: tips.groupby(["day", "smoker"], as_index=False).mean()
Out[81]:
    day smoker  total_bill       tip      size   tip_pct
0   Fri     No   18.420000  2.812500  2.250000  0.151650
1   Fri    Yes   16.813333  2.714000  2.066667  0.174783
2   Sat     No   19.661778  3.102889  2.555556  0.158048
3   Sat    Yes   21.276667  2.875476  2.476190  0.147906
4   Sun     No   20.506667  3.167895  2.929825  0.160113
5   Sun    Yes   24.120000  3.516842  2.578947  0.187250
6  Thur     No   17.113111  2.673778  2.488889  0.160298
7  Thur    Yes   19.190588  3.030000  2.352941  0.163863
```

Of course, it's always possible to obtain the result in this format by calling `reset_index` on the result. Using the `as_index=False` argument avoids some unnecessary computations.

10.3 Apply: General split-apply-combine

The most general-purpose GroupBy method is `apply`, which is the subject of this section. `apply` splits the object being manipulated into pieces, invokes the passed function on each piece, and then attempts to concatenate the pieces.

Returning to the tipping dataset from before, suppose you wanted to select the top five `tip_pct` values by group. First, write a function that selects the rows with the largest values in a particular column:

```
In [82]: def top(df, n=5, column="tip_pct"):
   ....:     return df.sort_values(column, ascending=False)[:n]

In [83]: top(tips, n=6)
Out[83]:
     total_bill   tip smoker  day    time  size   tip_pct
172        7.25  5.15    Yes  Sun  Dinner     2  0.710345
178        9.60  4.00    Yes  Sun  Dinner     2  0.416667
67         3.07  1.00    Yes  Sat  Dinner     1  0.325733
232       11.61  3.39     No  Sat  Dinner     2  0.291990
183       23.17  6.50    Yes  Sun  Dinner     4  0.280535
109       14.31  4.00    Yes  Sat  Dinner     2  0.279525
```

Now, if we group by `smoker`, say, and call `apply` with this function, we get the following:

```
In [84]: tips.groupby("smoker").apply(top)
Out[84]:
            total_bill   tip smoker   day    time  size   tip_pct
smoker
No     232       11.61  3.39     No   Sat  Dinner     2  0.291990
       149        7.51  2.00     No  Thur   Lunch     2  0.266312
       51        10.29  2.60     No   Sun  Dinner     2  0.252672
       185       20.69  5.00     No   Sun  Dinner     5  0.241663
       88        24.71  5.85     No  Thur   Lunch     2  0.236746
Yes    172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
       178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
       67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
       183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
       109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
```

What has happened here? First, the `tips` DataFrame is split into groups based on the value of `smoker`. Then the `top` function is called on each group, and the results of each function call are glued together using `pandas.concat`, labeling the pieces with the group names. The result therefore has a hierarchical index with an inner level that contains index values from the original DataFrame.

If you pass a function to `apply` that takes other arguments or keywords, you can pass these after the function:

```
In [85]: tips.groupby(["smoker", "day"]).apply(top, n=1, column="total_bill")
Out[85]:
                 total_bill    tip smoker   day    time  size   tip_pct
smoker day
No     Fri  94        22.75   3.25     No   Fri  Dinner     2  0.142857
       Sat  212       48.33   9.00     No   Sat  Dinner     4  0.186220
       Sun  156       48.17   5.00     No   Sun  Dinner     6  0.103799
       Thur 142       41.19   5.00     No  Thur   Lunch     5  0.121389
Yes    Fri  95        40.17   4.73    Yes   Fri  Dinner     4  0.117750
       Sat  170       50.81  10.00    Yes   Sat  Dinner     3  0.196812
       Sun  182       45.35   3.50    Yes   Sun  Dinner     3  0.077178
       Thur 197       43.11   5.00    Yes  Thur   Lunch     4  0.115982
```

Beyond these basic usage mechanics, getting the most out of `apply` may require some creativity. What occurs inside the function passed is up to you; it must either return a pandas object or a scalar value. The rest of this chapter will consist mainly of examples showing you how to solve various problems using `groupby`.

For example, you may recall that I earlier called `describe` on a GroupBy object:

```
In [86]: result = tips.groupby("smoker")["tip_pct"].describe()

In [87]: result
Out[87]:
        count      mean       std       min       25%       50%       75%  \
smoker
No      151.0  0.159328  0.039910  0.056797  0.136906  0.155625  0.185014
Yes      93.0  0.163196  0.085119  0.035638  0.106771  0.153846  0.195059
             max
smoker
No      0.291990
Yes     0.710345

In [88]: result.unstack("smoker")
Out[88]:
       smoker
count  No        151.000000
       Yes        93.000000
mean   No          0.159328
       Yes         0.163196
std    No          0.039910
       Yes         0.085119
min    No          0.056797
       Yes         0.035638
25%    No          0.136906
       Yes         0.106771
50%    No          0.155625
       Yes         0.153846
75%    No          0.185014
       Yes         0.195059
max    No          0.291990
       Yes         0.710345
dtype: float64
```

Inside GroupBy, when you invoke a method like `describe`, it is actually just a shortcut for:

```
def f(group):
    return group.describe()

grouped.apply(f)
```

Suppressing the Group Keys

In the preceding examples, you see that the resulting object has a hierarchical index formed from the group keys, along with the indexes of each piece of the original object. You can disable this by passing `group_keys=False` to `groupby`:

```
In [89]: tips.groupby("smoker", group_keys=False).apply(top)
Out[89]:
     total_bill   tip smoker   day    time  size   tip_pct
232       11.61  3.39     No   Sat  Dinner     2  0.291990
149        7.51  2.00     No  Thur   Lunch     2  0.266312
51        10.29  2.60     No   Sun  Dinner     2  0.252672
185       20.69  5.00     No   Sun  Dinner     5  0.241663
88        24.71  5.85     No  Thur   Lunch     2  0.236746
172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
```

Quantile and Bucket Analysis

As you may recall from Chapter 8, pandas has some tools, in particular `pandas.cut` and `pandas.qcut`, for slicing data up into buckets with bins of your choosing, or by sample quantiles. Combining these functions with `groupby` makes it convenient to perform bucket or quantile analysis on a dataset. Consider a simple random dataset and an equal-length bucket categorization using `pandas.cut`:

```
In [90]: frame = pd.DataFrame({"data1": np.random.standard_normal(1000),
   ....:                       "data2": np.random.standard_normal(1000)})

In [91]: frame.head()
Out[91]:
      data1     data2
0 -0.660524 -0.612905
1  0.862580  0.316447
2 -0.010032  0.838295
3  0.050009 -1.034423
4  0.670216  0.434304

In [92]: quartiles = pd.cut(frame["data1"], 4)

In [93]: quartiles.head(10)
Out[93]:
0     (-1.23, 0.489]
1     (0.489, 2.208]
2     (-1.23, 0.489]
3     (-1.23, 0.489]
4     (0.489, 2.208]
5     (0.489, 2.208]
6     (-1.23, 0.489]
```

```
7     (-1.23, 0.489]
8    (-2.956, -1.23]
9     (-1.23, 0.489]
Name: data1, dtype: category
Categories (4, interval[float64, right]): [(-2.956, -1.23] < (-1.23, 0.489] < (0.
489, 2.208] <
                                           (2.208, 3.928]]
```

The `Categorical` object returned by `cut` can be passed directly to `groupby`. So we could compute a set of group statistics for the quartiles, like so:

```
In [94]: def get_stats(group):
   ....:     return pd.DataFrame(
   ....:         {"min": group.min(), "max": group.max(),
   ....:         "count": group.count(), "mean": group.mean()}
   ....:     )

In [95]: grouped = frame.groupby(quartiles)

In [96]: grouped.apply(get_stats)
Out[96]:
                            min       max  count      mean
data1
(-2.956, -1.23] data1 -2.949343 -1.230179     94 -1.658818
                data2 -3.399312  1.670835     94 -0.033333
(-1.23, 0.489]  data1 -1.228918  0.488675    598 -0.329524
                data2 -2.989741  3.260383    598 -0.002622
(0.489, 2.208]  data1  0.489965  2.200997    298  1.065727
                data2 -3.745356  2.954439    298  0.078249
(2.208, 3.928]  data1  2.212303  3.927528     10  2.644253
                data2 -1.929776  1.765640     10  0.024750
```

Keep in mind the same result could have been computed more simply with:

```
In [97]: grouped.agg(["min", "max", "count", "mean"])
Out[97]:
                    data1                                data2                   \
                      min       max count      mean       min       max count
data1
(-2.956, -1.23] -2.949343 -1.230179    94 -1.658818 -3.399312  1.670835    94
(-1.23, 0.489]  -1.228918  0.488675   598 -0.329524 -2.989741  3.260383   598
(0.489, 2.208]   0.489965  2.200997   298  1.065727 -3.745356  2.954439   298
(2.208, 3.928]   2.212303  3.927528    10  2.644253 -1.929776  1.765640    10

                     mean
data1
(-2.956, -1.23] -0.033333
(-1.23, 0.489]  -0.002622
(0.489, 2.208]   0.078249
(2.208, 3.928]   0.024750
```

These were equal-length buckets; to compute equal-size buckets based on sample quantiles, use `pandas.qcut`. We can pass `4` as the number of bucket compute sam-

ple quartiles, and pass `labels=False` to obtain just the quartile indices instead of intervals:

```
In [98]: quartiles_samp = pd.qcut(frame["data1"], 4, labels=False)

In [99]: quartiles_samp.head()
Out[99]:
0    1
1    3
2    2
3    2
4    3
Name: data1, dtype: int64

In [100]: grouped = frame.groupby(quartiles_samp)

In [101]: grouped.apply(get_stats)
Out[101]:
                   min       max  count      mean
data1
0     data1 -2.949343 -0.685484    250 -1.212173
      data2 -3.399312  2.628441    250 -0.027045
1     data1 -0.683066 -0.030280    250 -0.368334
      data2 -2.630247  3.260383    250 -0.027845
2     data1 -0.027734  0.618965    250  0.295812
      data2 -3.056990  2.458842    250  0.014450
3     data1  0.623587  3.927528    250  1.248875
      data2 -3.745356  2.954439    250  0.115899
```

Example: Filling Missing Values with Group-Specific Values

When cleaning up missing data, in some cases you will remove data observations using `dropna`, but in others you may want to fill in the null (NA) values using a fixed value or some value derived from the data. `fillna` is the right tool to use; for example, here I fill in the null values with the mean:

```
In [102]: s = pd.Series(np.random.standard_normal(6))

In [103]: s[::2] = np.nan

In [104]: s
Out[104]:
0         NaN
1    0.227290
2         NaN
3   -2.153545
4         NaN
5   -0.375842
dtype: float64

In [105]: s.fillna(s.mean())
```

```
Out[105]:
0   -0.767366
1    0.227290
2   -0.767366
3   -2.153545
4   -0.767366
5   -0.375842
dtype: float64
```

Suppose you need the fill value to vary by group. One way to do this is to group the data and use `apply` with a function that calls `fillna` on each data chunk. Here is some sample data on US states divided into eastern and western regions:

```
In [106]: states = ["Ohio", "New York", "Vermont", "Florida",
   .....:           "Oregon", "Nevada", "California", "Idaho"]

In [107]: group_key = ["East", "East", "East", "East",
   .....:              "West", "West", "West", "West"]

In [108]: data = pd.Series(np.random.standard_normal(8), index=states)

In [109]: data
Out[109]:
Ohio          0.329939
New York      0.981994
Vermont       1.105913
Florida      -1.613716
Oregon        1.561587
Nevada        0.406510
California    0.359244
Idaho        -0.614436
dtype: float64
```

Let's set some values in the data to be missing:

```
In [110]: data[["Vermont", "Nevada", "Idaho"]] = np.nan

In [111]: data
Out[111]:
Ohio          0.329939
New York      0.981994
Vermont            NaN
Florida      -1.613716
Oregon        1.561587
Nevada             NaN
California    0.359244
Idaho              NaN
dtype: float64

In [112]: data.groupby(group_key).size()
Out[112]:
East    4
West    4
```

```
dtype: int64

In [113]: data.groupby(group_key).count()
Out[113]:
East    3
West    2
dtype: int64

In [114]: data.groupby(group_key).mean()
Out[114]:
East   -0.100594
West    0.960416
dtype: float64
```

We can fill the NA values using the group means, like so:

```
In [115]: def fill_mean(group):
   .....:     return group.fillna(group.mean())

In [116]: data.groupby(group_key).apply(fill_mean)
Out[116]:
Ohio          0.329939
New York      0.981994
Vermont      -0.100594
Florida      -1.613716
Oregon        1.561587
Nevada        0.960416
California    0.359244
Idaho         0.960416
dtype: float64
```

In another case, you might have predefined fill values in your code that vary by group. Since the groups have a `name` attribute set internally, we can use that:

```
In [117]: fill_values = {"East": 0.5, "West": -1}

In [118]: def fill_func(group):
   .....:     return group.fillna(fill_values[group.name])

In [119]: data.groupby(group_key).apply(fill_func)
Out[119]:
Ohio          0.329939
New York      0.981994
Vermont       0.500000
Florida      -1.613716
Oregon        1.561587
Nevada       -1.000000
California    0.359244
Idaho        -1.000000
dtype: float64
```

Example: Random Sampling and Permutation

Suppose you wanted to draw a random sample (with or without replacement) from a large dataset for Monte Carlo simulation purposes or some other application. There are a number of ways to perform the "draws"; here we use the `sample` method for Series.

To demonstrate, here's a way to construct a deck of English-style playing cards:

```
suits = ["H", "S", "C", "D"]  # Hearts, Spades, Clubs, Diamonds
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ["A"] + list(range(2, 11)) + ["J", "K", "Q"]
cards = []
for suit in suits:
    cards.extend(str(num) + suit for num in base_names)

deck = pd.Series(card_val, index=cards)
```

Now we have a Series of length 52 whose index contains card names, and values are the ones used in blackjack and other games (to keep things simple, I let the ace `"A"` be 1):

```
In [121]: deck.head(13)
Out[121]:
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H    10
JH     10
KH     10
QH     10
dtype: int64
```

Now, based on what I said before, drawing a hand of five cards from the deck could be written as:

```
In [122]: def draw(deck, n=5):
   .....:     return deck.sample(n)

In [123]: draw(deck)
Out[123]:
4D     4
QH    10
8S     8
7D     7
```

```
9C     9
dtype: int64
```

Suppose you wanted two random cards from each suit. Because the suit is the last character of each card name, we can group based on this and use `apply`:

```
In [124]: def get_suit(card):
   .....:     # last letter is suit
   .....:     return card[-1]

In [125]: deck.groupby(get_suit).apply(draw, n=2)
Out[125]:
C  6C     6
   KC    10
D  7D     7
   3D     3
H  7H     7
   9H     9
S  2S     2
   QS    10
dtype: int64
```

Alternatively, we could pass `group_keys=False` to drop the outer suit index, leaving in just the selected cards:

```
In [126]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[126]:
AC      1
3C      3
5D      5
4D      4
10H    10
7H      7
QS     10
7S      7
dtype: int64
```

Example: Group Weighted Average and Correlation

Under the split-apply-combine paradigm of `groupby`, operations between columns in a DataFrame or two Series, such as a group weighted average, are possible. As an example, take this dataset containing group keys, values, and some weights:

```
In [127]: df = pd.DataFrame({"category": ["a", "a", "a", "a",
   .....:                                 "b", "b", "b", "b"],
   .....:                    "data": np.random.standard_normal(8),
   .....:                    "weights": np.random.uniform(size=8)})

In [128]: df
Out[128]:
  category      data   weights
0        a -1.691656  0.955905
```

```
1         a  0.511622  0.012745
2         a -0.401675  0.137009
3         a  0.968578  0.763037
4         b -1.818215  0.492472
5         b  0.279963  0.832908
6         b -0.200819  0.658331
7         b -0.217221  0.612009
```

The weighted average by `category` would then be:

```
In [129]: grouped = df.groupby("category")

In [130]: def get_wavg(group):
   .....:     return np.average(group["data"], weights=group["weights"])

In [131]: grouped.apply(get_wavg)
Out[131]:
category
a   -0.495807
b   -0.357273
dtype: float64
```

As another example, consider a financial dataset originally obtained from Yahoo! Finance containing end-of-day prices for a few stocks and the S&P 500 index (the `SPX` symbol):

```
In [132]: close_px = pd.read_csv("examples/stock_px.csv", parse_dates=True,
   .....:                        index_col=0)

In [133]: close_px.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   AAPL    2214 non-null   float64
 1   MSFT    2214 non-null   float64
 2   XOM     2214 non-null   float64
 3   SPX     2214 non-null   float64
dtypes: float64(4)
memory usage: 86.5 KB

In [134]: close_px.tail(4)
Out[134]:
              AAPL   MSFT    XOM      SPX
2011-10-11  400.29  27.00  76.27  1195.54
2011-10-12  402.19  26.96  77.16  1207.25
2011-10-13  408.43  27.18  76.37  1203.66
2011-10-14  422.00  27.27  78.11  1224.58
```

The DataFrame `info()` method here is a convenient way to get an overview of the contents of a DataFrame.

One task of interest might be to compute a DataFrame consisting of the yearly correlations of daily returns (computed from percent changes) with SPX. As one way to do this, we first create a function that computes the pair-wise correlation of each column with the "SPX" column:

```
In [135]: def spx_corr(group):
   .....:     return group.corrwith(group["SPX"])
```

Next, we compute percent change on close_px using pct_change:

```
In [136]: rets = close_px.pct_change().dropna()
```

Lastly, we group these percent changes by year, which can be extracted from each row label with a one-line function that returns the year attribute of each datetime label:

```
In [137]: def get_year(x):
   .....:     return x.year

In [138]: by_year = rets.groupby(get_year)

In [139]: by_year.apply(spx_corr)
Out[139]:
          AAPL      MSFT       XOM  SPX
2003  0.541124  0.745174  0.661265  1.0
2004  0.374283  0.588531  0.557742  1.0
2005  0.467540  0.562374  0.631010  1.0
2006  0.428267  0.406126  0.518514  1.0
2007  0.508118  0.658770  0.786264  1.0
2008  0.681434  0.804626  0.828303  1.0
2009  0.707103  0.654902  0.797921  1.0
2010  0.710105  0.730118  0.839057  1.0
2011  0.691931  0.800996  0.859975  1.0
```

You could also compute intercolumn correlations. Here we compute the annual correlation between Apple and Microsoft:

```
In [140]: def corr_aapl_msft(group):
   .....:     return group["AAPL"].corr(group["MSFT"])

In [141]: by_year.apply(corr_aapl_msft)
Out[141]:
2003    0.480868
2004    0.259024
2005    0.300093
2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
dtype: float64
```

Example: Group-Wise Linear Regression

In the same theme as the previous example, you can use `groupby` to perform more complex group-wise statistical analysis, as long as the function returns a pandas object or scalar value. For example, I can define the following `regress` function (using the `statsmodels` econometrics library), which executes an ordinary least squares (OLS) regression on each chunk of data:

```
import statsmodels.api as sm
def regress(data, yvar=None, xvars=None):
    Y = data[yvar]
    X = data[xvars]
    X["intercept"] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

You can install `statsmodels` with conda if you don't have it already:

```
conda install statsmodels
```

Now, to run a yearly linear regression of `AAPL` on `SPX` returns, execute:

```
In [143]: by_year.apply(regress, yvar="AAPL", xvars=["SPX"])
Out[143]:
           SPX  intercept
2003  1.195406   0.000710
2004  1.363463   0.004201
2005  1.766415   0.003246
2006  1.645496   0.000080
2007  1.198761   0.003438
2008  0.968016  -0.001110
2009  0.879103   0.002954
2010  1.052608   0.001261
2011  0.806605   0.001514
```

10.4 Group Transforms and "Unwrapped" GroupBys

In Section 10.3, "Apply: General split-apply-combine," on page 335, we looked at the `apply` method in grouped operations for performing transformations. There is another built-in method called `transform`, which is similar to `apply` but imposes more constraints on the kind of function you can use:

- It can produce a scalar value to be broadcast to the shape of the group.
- It can produce an object of the same shape as the input group.
- It must not mutate its input.

Let's consider a simple example for illustration:

```
In [144]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,
   .....:                    'value': np.arange(12.)})

In [145]: df
Out[145]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

Here are the group means by key:

```
In [146]: g = df.groupby('key')['value']

In [147]: g.mean()
Out[147]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64
```

Suppose instead we wanted to produce a Series of the same shape as `df['value']` but with values replaced by the average grouped by `'key'`. We can pass a function that computes the mean of a single group to `transform`:

```
In [148]: def get_mean(group):
   .....:     return group.mean()

In [149]: g.transform(get_mean)
Out[149]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

For built-in aggregation functions, we can pass a string alias as with the GroupBy `agg` method:

```
In [150]: g.transform('mean')
Out[150]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

Like `apply`, `transform` works with functions that return Series, but the result must be the same size as the input. For example, we can multiply each group by 2 using a helper function:

```
In [151]: def times_two(group):
   .....:     return group * 2

In [152]: g.transform(times_two)
Out[152]:
0      0.0
1      2.0
2      4.0
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
11    22.0
Name: value, dtype: float64
```

As a more complicated example, we can compute the ranks in descending order for each group:

```
In [153]: def get_ranks(group):
   .....:     return group.rank(ascending=False)

In [154]: g.transform(get_ranks)
Out[154]:
0     4.0
1     4.0
2     4.0
```

```
3     3.0
4     3.0
5     3.0
6     2.0
7     2.0
8     2.0
9     1.0
10    1.0
11    1.0
Name: value, dtype: float64
```

Consider a group transformation function composed from simple aggregations:

```
In [155]: def normalize(x):
   .....:     return (x - x.mean()) / x.std()
```

We can obtain equivalent results in this case using either `transform` or `apply`:

```
In [156]: g.transform(normalize)
Out[156]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64

In [157]: g.apply(normalize)
Out[157]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

Built-in aggregate functions like `'mean'` or `'sum'` are often much faster than a general `apply` function. These also have a "fast path" when used with `transform`. This allows us to perform what is called an *unwrapped* group operation:

```
In [158]: g.transform('mean')
Out[158]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64

In [159]: normalized = (df['value'] - g.transform('mean')) / g.transform('std')

In [160]: normalized
Out[160]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

Here, we are doing arithmetic between the outputs of multiple GroupBy operations instead of writing a function and passing it to `groupby(...).apply`. That is what is meant by "unwrapped."

While an unwrapped group operation may involve multiple group aggregations, the overall benefit of vectorized operations often outweighs this.

10.5 Pivot Tables and Cross-Tabulation

A *pivot table* is a data summarization tool frequently found in spreadsheet programs and other data analysis software. It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns. Pivot tables in Python with pandas are made possible through the `groupby` facility described in this chapter, combined with reshape operations utilizing hierarchical indexing. DataFrame also has a `pivot_table` method, and

there is also a top-level `pandas.pivot_table` function. In addition to providing a convenience interface to `groupby`, `pivot_table` can add partial totals, also known as *margins*.

Returning to the tipping dataset, suppose you wanted to compute a table of group means (the default `pivot_table` aggregation type) arranged by `day` and `smoker` on the rows:

```
In [161]: tips.head()
Out[161]:
   total_bill   tip smoker  day    time  size   tip_pct
0       16.99  1.01     No  Sun  Dinner     2  0.059447
1       10.34  1.66     No  Sun  Dinner     3  0.160542
2       21.01  3.50     No  Sun  Dinner     3  0.166587
3       23.68  3.31     No  Sun  Dinner     2  0.139780
4       24.59  3.61     No  Sun  Dinner     4  0.146808

In [162]: tips.pivot_table(index=["day", "smoker"])
Out[162]:
                 size       tip   tip_pct  total_bill
day  smoker
Fri  No      2.250000  2.812500  0.151650   18.420000
     Yes     2.066667  2.714000  0.174783   16.813333
Sat  No      2.555556  3.102889  0.158048   19.661778
     Yes     2.476190  2.875476  0.147906   21.276667
Sun  No      2.929825  3.167895  0.160113   20.506667
     Yes     2.578947  3.516842  0.187250   24.120000
Thur No      2.488889  2.673778  0.160298   17.113111
     Yes     2.352941  3.030000  0.163863   19.190588
```

This could have been produced with `groupby` directly, using `tips.groupby(["day", "smoker"]).mean()`. Now, suppose we want to take the average of only `tip_pct` and `size`, and additionally group by `time`. I'll put `smoker` in the table columns and `time` and `day` in the rows:

```
In [163]: tips.pivot_table(index=["time", "day"], columns="smoker",
   .....:                  values=["tip_pct", "size"])
Out[163]:
                 size             tip_pct
smoker             No       Yes        No       Yes
time   day
Dinner Fri   2.000000  2.222222  0.139622  0.165347
       Sat   2.555556  2.476190  0.158048  0.147906
       Sun   2.929825  2.578947  0.160113  0.187250
       Thur  2.000000       NaN  0.159744       NaN
Lunch  Fri   3.000000  1.833333  0.187735  0.188937
       Thur  2.500000  2.352941  0.160311  0.163863
```

We could augment this table to include partial totals by passing `margins=True`. This has the effect of adding `All` row and column labels, with corresponding values being the group statistics for all the data within a single tier:

```
In [164]: tips.pivot_table(index=["time", "day"], columns="smoker",
   .....:                  values=["tip_pct", "size"], margins=True)
Out[164]: 
                 size                          tip_pct                    
smoker             No       Yes       All       No       Yes       All
time   day                                                            
Dinner Fri   2.000000  2.222222  2.166667  0.139622  0.165347  0.158916
       Sat   2.555556  2.476190  2.517241  0.158048  0.147906  0.153152
       Sun   2.929825  2.578947  2.842105  0.160113  0.187250  0.166897
       Thur  2.000000       NaN  2.000000  0.159744       NaN  0.159744
Lunch  Fri   3.000000  1.833333  2.000000  0.187735  0.188937  0.188765
       Thur  2.500000  2.352941  2.459016  0.160311  0.163863  0.161301
All          2.668874  2.408602  2.569672  0.159328  0.163196  0.160803
```

Here, the `All` values are means without taking into account smoker versus non-smoker (the `All` columns) or any of the two levels of grouping on the rows (the `All` row).

To use an aggregation function other than `mean`, pass it to the `aggfunc` keyword argument. For example, `"count"` or `len` will give you a cross-tabulation (count or frequency) of group sizes (though `"count"` will exclude null values from the count within data groups, while `len` will not):

```
In [165]: tips.pivot_table(index=["time", "smoker"], columns="day",
   .....:                  values="tip_pct", aggfunc=len, margins=True)
Out[165]: 
day             Fri   Sat   Sun  Thur  All
time   smoker                             
Dinner No       3.0  45.0  57.0   1.0  106
       Yes      9.0  42.0  19.0   NaN   70
Lunch  No       1.0   NaN   NaN  44.0   45
       Yes      6.0   NaN   NaN  17.0   23
All            19.0  87.0  76.0  62.0  244
```

If some combinations are empty (or otherwise NA), you may wish to pass a `fill_value`:

```
In [166]: tips.pivot_table(index=["time", "size", "smoker"], columns="day",
   .....:                  values="tip_pct", fill_value=0)
Out[166]: 
day                      Fri       Sat       Sun      Thur
time   size smoker                                        
Dinner 1    No      0.000000  0.137931  0.000000  0.000000
            Yes     0.000000  0.325733  0.000000  0.000000
       2    No      0.139622  0.162705  0.168859  0.159744
            Yes     0.171297  0.148668  0.207893  0.000000
       3    No      0.000000  0.154661  0.152663  0.000000
...                      ...       ...       ...       ...
Lunch  3    Yes     0.000000  0.000000  0.000000  0.204952
       4    No      0.000000  0.000000  0.000000  0.138919
            Yes     0.000000  0.000000  0.000000  0.155410
       5    No      0.000000  0.000000  0.000000  0.121389
```

```
       6    No      0.000000  0.000000  0.000000  0.173706
[21 rows x 4 columns]
```

See Table 10-2 for a summary of `pivot_table` options.

Table 10-2. `pivot_table` options

Argument	Description
`values`	Column name or names to aggregate; by default, aggregates all numeric columns
`index`	Column names or other group keys to group on the rows of the resulting pivot table
`columns`	Column names or other group keys to group on the columns of the resulting pivot table
`aggfunc`	Aggregation function or list of functions (`"mean"` by default); can be any function valid in a `groupby` context
`fill_value`	Replace missing values in the result table
`dropna`	If `True`, do not include columns whose entries are all NA
`margins`	Add row/column subtotals and grand total (`False` by default)
`margins_name`	Name to use for the margin row/column labels when passing `margins=True`; defaults to `"All"`
`observed`	With Categorical group keys, if `True`, show only the observed category values in the keys rather than all categories

Cross-Tabulations: Crosstab

A *cross-tabulation* (or *crosstab* for short) is a special case of a pivot table that computes group frequencies. Here is an example:

```
In [167]: from io import StringIO

In [168]: data = """Sample  Nationality  Handedness
   .....: 1   USA  Right-handed
   .....: 2   Japan    Left-handed
   .....: 3   USA  Right-handed
   .....: 4   Japan    Right-handed
   .....: 5   Japan    Left-handed
   .....: 6   Japan    Right-handed
   .....: 7   USA  Right-handed
   .....: 8   USA  Left-handed
   .....: 9   Japan    Right-handed
   .....: 10  USA  Right-handed"""
   .....:

In [169]: data = pd.read_table(StringIO(data), sep="\s+")

In [170]: data
Out[170]: 
   Sample Nationality    Handedness
0       1         USA  Right-handed
1       2       Japan   Left-handed
2       3         USA  Right-handed
3       4       Japan  Right-handed
4       5       Japan   Left-handed
```

```
5       6       Japan  Right-handed
6       7         USA  Right-handed
7       8         USA   Left-handed
8       9       Japan  Right-handed
9      10         USA  Right-handed
```

As part of some survey analysis, we might want to summarize this data by nationality and handedness. You could use `pivot_table` to do this, but the `pandas.crosstab` function can be more convenient:

```
In [171]: pd.crosstab(data["Nationality"], data["Handedness"], margins=True)
Out[171]: 
Handedness   Left-handed  Right-handed  All
Nationality                                
Japan                  2             3    5
USA                    1             4    5
All                    3             7   10
```

The first two arguments to `crosstab` can each be an array or Series or a list of arrays. As in the tips data:

```
In [172]: pd.crosstab([tips["time"], tips["day"]], tips["smoker"], margins=True)
Out[172]: 
smoker        No  Yes  All
time   day                
Dinner Fri     3    9   12
       Sat    45   42   87
       Sun    57   19   76
       Thur    1    0    1
Lunch  Fri     1    6    7
       Thur   44   17   61
All          151   93  244
```

10.6 Conclusion

Mastering pandas's data grouping tools can help with data cleaning and modeling or statistical analysis work. In Chapter 13 we will look at several more example use cases for `groupby` on real data.

In the next chapter, we turn our attention to time series data.