CHAPTER 10
Data Aggregation and Group Operations

Categorizing a dataset and applying a function to each group, whether an aggregation or transformation, is often 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 flexible `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 are somewhat constrained in 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 utilizing any function that accepts a pandas object or NumPy array. 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

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.

10.1 GroupBy Mechanics

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=0`) or its columns (`axis=1`). 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.

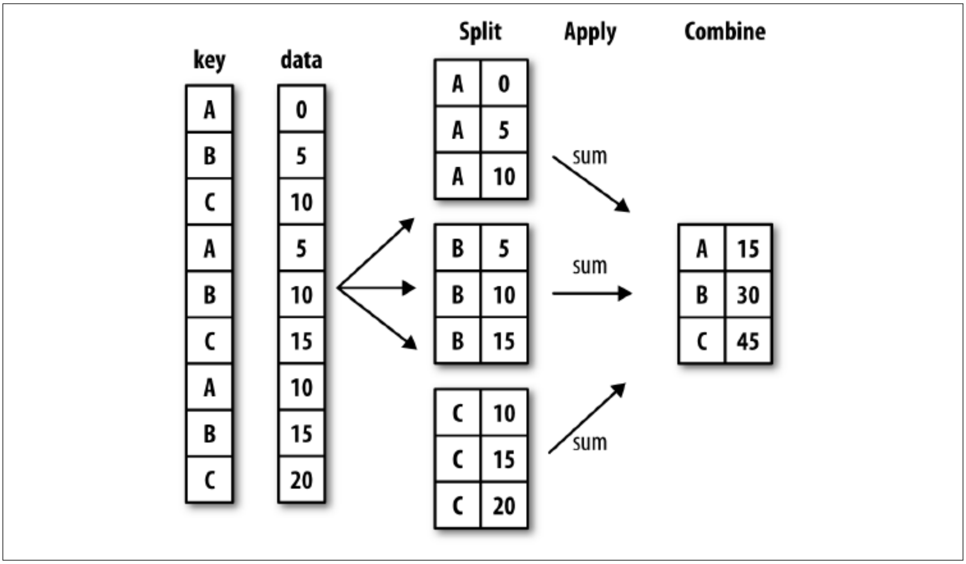

Figure 10-1. Illustration of a group aggregation

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 dict 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

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 [10]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
   ....:                    'key2' : ['one', 'two', 'one', 'two', 'one'],
   ....:                    'data1' : np.random.randn(5),
   ....:                    'data2' : np.random.randn(5)})

In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

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 [12]: grouped = df['data1'].groupby(df['key1'])

In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy object at 0x7faa31537390>
```

This `grouped` variable is now a *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 [14]: grouped.mean()
Out[14]:
key1
a    0.746672
b   -0.537585
Name: data1, dtype: float64
```

Later, I'll explain more about what happens when you call `.mean()`. The important thing here is that the data (a Series) has been aggregated according to 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 [15]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()

In [16]: means
Out[16]:
key1  key2
a     one     0.880536
      two     0.478943
b     one    -0.519439
      two    -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 [17]: means.unstack()
Out[17]:
key2       one       two
key1
a     0.880536  0.478943
b    -0.519439 -0.555730
```

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

```
In [18]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])

In [19]: years = np.array([2005, 2005, 2006, 2005, 2006])

In [20]: df['data1'].groupby([states, years]).mean()
Out[20]:
California  2005    0.478943
            2006   -0.519439
Ohio        2005   -0.380219
            2006    1.965781
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 [21]: df.groupby('key1').mean()
Out[21]:
         data1     data2
key1
a     0.746672  0.910916
b    -0.537585  0.525384

In [22]: df.groupby(['key1', 'key2']).mean()
```

```
Out[22]:
              data1     data2
key1 key2
a    one   0.880536  1.319920
     two   0.478943  0.092908
b    one  -0.519439  0.281746
     two  -0.555730  0.769023
```

You may have noticed in the first case `df.groupby('key1').mean()` that there is no `key2` column in the result. Because `df['key2']` is not numeric data, it is said to be a *nuisance column*, which is therefore 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 [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

Take note that any missing values in a group key will be excluded from the result.

Iterating Over Groups

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

```
In [24]: for name, group in df.groupby('key1'):
   ....:     print(name)
   ....:     print(group)
   ....:
a
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
4  1.965781  1.246435    a  one
b
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

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

```
In [25]: for (k1, k2), group in df.groupby(['key1', 'key2']):
   ....:     print((k1, k2))
   ....:     print(group)
```

```
   ....:
('a', 'one')
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
4  1.965781  1.246435    a  one
('a', 'two')
      data1     data2 key1 key2
1  0.478943  0.092908    a  two
('b', 'one')
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
('b', 'two')
     data1     data2 key1 key2
3 -0.55573  0.769023    b  two
```

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

```
In [26]: pieces = dict(list(df.groupby('key1')))

In [27]: pieces['b']
Out[27]:
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

By default `groupby` groups on `axis=0`, but you can group on any of the other axes. For example, we could group the columns of our example `df` here by `dtype` like so:

```
In [28]: df.dtypes
Out[28]:
data1    float64
data2    float64
key1      object
key2      object
dtype: object

In [29]: grouped = df.groupby(df.dtypes, axis=1)
```

We can print out the groups like so:

```
In [30]: for dtype, group in grouped:
   ....:     print(dtype)
   ....:     print(group)
   ....:
float64
      data1     data2
0 -0.204708  1.393406
1  0.478943  0.092908
2 -0.519439  0.281746
3 -0.555730  0.769023
4  1.965781  1.246435
object
  key1 key2
```

```
0     a  one
1     a  two
2     b  one
3     b  two
4     a  one
```

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 syntactic sugar 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 means for just the `data2` column and get the result as a DataFrame, we could write:

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[31]:
              data2
key1 key2
a    one   1.319920
     two   0.092908
b    one   0.281746
     two   0.769023
```

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 [32]: s_grouped = df.groupby(['key1', 'key2'])['data2']

In [33]: s_grouped
Out[33]: <pandas.core.groupby.SeriesGroupBy object at 0x7faa30c78da0>

In [34]: s_grouped.mean()
Out[34]:
key1  key2
a     one     1.319920
      two     0.092908
b     one     0.281746
      two     0.769023
Name: data2, dtype: float64
```

Grouping with Dicts and Series

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

```
In [35]: people = pd.DataFrame(np.random.randn(5, 5),
   ....:                       columns=['a', 'b', 'c', 'd', 'e'],
   ....:                       index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])

In [36]: people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values

In [37]: people
Out[37]:
               a         b         c         d         e
Joe     1.007189 -1.296221  0.274992  0.228913  1.352917
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes    -0.539741       NaN       NaN -1.021228 -0.577087
Jim     0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

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

```
In [38]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
   ....:            'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

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

```
In [39]: by_column = people.groupby(mapping, axis=1)

In [40]: by_column.sum()
Out[40]:
            blue       red
Joe     0.503905  1.063885
Steve   1.297183 -1.553778
Wes    -1.021228 -1.116829
Jim     0.524712  1.770545
Travis -4.230992 -2.405455
```

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

```
In [41]: map_series = pd.Series(mapping)

In [42]: map_series
Out[42]:
a       red
b       red
c      blue
d      blue
e       red
f    orange
```

```
dtype: object

In [43]: people.groupby(map_series, axis=1).count()
Out[43]:
        blue  red
Joe        2    3
Steve      2    3
Wes        1    2
Jim        2    3
Travis     2    3
```

Grouping with Functions

Using Python functions is a more generic way of defining a group mapping compared with a dict or Series. Any function passed as a group key will be called once per index value, 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 the length of the names; while you could compute an array of string lengths, it's simpler to just pass the `len` function:

```
In [44]: people.groupby(len).sum()
Out[44]:
          a         b         c         d         e
3  0.591569 -0.993608  0.798764 -0.791374  2.119639
5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

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

```
In [45]: key_list = ['one', 'one', 'one', 'two', 'two']

In [46]: people.groupby([len, key_list]).min()
Out[46]:
              a         b         c         d         e
3 one -0.539741 -1.296221  0.274992 -1.021228 -0.577087
  two  0.124121  0.302614  0.523772  0.000940  1.343810
5 one  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

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 [47]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
   ....:                                     [1, 3, 5, 1, 3]],
   ....:                                     names=['cty', 'tenor'])

In [48]: hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)
```

```
In [49]: hier_df
Out[49]:
cty          US                            JP          
tenor         1         3         5         1         3
0      0.560145 -1.265934  0.119827 -1.063512  0.332883
1     -2.359419 -0.199543 -1.541996 -0.970736 -1.307030
2      0.286350  0.377984 -0.753887  0.331286  1.349742
3      0.069877  0.246674 -0.011862  1.004812  1.327195
```

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

```
In [50]: hier_df.groupby(level='cty', axis=1).count()
Out[50]:
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
`count`	Number of non-NA values in the group
`sum`	Sum of non-NA values
`mean`	Mean of non-NA values
`median`	Arithmetic median of non-NA values
`std, var`	Unbiased (n – 1 denominator) standard deviation and variance
`min, max`	Minimum and maximum of non-NA values
`prod`	Product of non-NA values
`first, last`	First and last non-NA values

You can use aggregations of your own devising and additionally call any method that is also defined on the grouped object. For example, you might recall that `quantile` computes sample quantiles of a Series or a DataFrame's columns.

While `quantile` is not explicitly implemented for GroupBy, it is a Series method and thus available for use. Internally, GroupBy efficiently slices up the Series, calls `piece.quantile(0.9)` for each piece, and then assembles those results together into the result object:

```
In [51]: df
Out[51]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [52]: grouped = df.groupby('key1')

In [53]: grouped['data1'].quantile(0.9)
Out[53]:
key1
a    1.668413
b   -0.523068
Name: data1, dtype: float64
```

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

```
In [54]: def peak_to_peak(arr):
   ....:     return arr.max() - arr.min()

In [55]: grouped.agg(peak_to_peak)
Out[55]:
         data1     data2
key1
a     2.170488  1.300498
b     0.036292  0.487276
```

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

```
In [56]: grouped.describe()
Out[56]:
     data1                                                            \
     count      mean       std       min       25%       50%       75%
key1
a      3.0  0.746672  1.109736 -0.204708  0.137118  0.478943  1.222362
b      2.0 -0.537585  0.025662 -0.555730 -0.546657 -0.537585 -0.528512
                data2                                                  \
           max count      mean       std       min       25%       50%
key1
a     1.965781   3.0  0.910916  0.712217  0.092908  0.669671  1.246435
b    -0.519439   2.0  0.525384  0.344556  0.281746  0.403565  0.525384

           75%       max
key1
```

```
a      1.319920  1.393406
b      0.647203  0.769023
```

I will explain in more detail what has happened here in Section 10.3, "Apply: General split-apply-combine," on page 302.

> 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 from earlier examples. After loading it with `read_csv`, we add a tipping percentage column `tip_pct`:

```
In [57]: tips = pd.read_csv('examples/tips.csv')

# Add tip percentage of total bill
In [58]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [59]: tips[:6]
Out[59]:
   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
5       25.29  4.71     No  Sun  Dinner     4  0.186240
```

As you've already seen, aggregating a Series or all of the columns of a DataFrame is a matter of using `aggregate` 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 [60]: 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 [61]: grouped_pct = grouped['tip_pct']

In [62]: grouped_pct.agg('mean')
Out[62]:
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 [63]: grouped_pct.agg(['mean', 'std', peak_to_peak])
Out[63]:
                 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 indepedently 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 [64]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[64]:
                  foo       bar
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 [65]: functions = ['count', 'mean', 'max']

In [66]: result = grouped['tip_pct', 'total_bill'].agg(functions)

In [67]: result
Out[67]:
            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 [68]: result['tip_pct']
Out[68]:
             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 [69]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]

In [70]: grouped['tip_pct', 'total_bill'].agg(ftuples)
Out[70]:
                 tip_pct              total_bill
            Durchschnitt Abweichung Durchschnitt  Abweichung
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 dict to `agg` that contains a mapping of column names to any of the function specifications listed so far:

```
In [71]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
Out[71]:
               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 [72]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
   ....:              'size' : 'sum'})
Out[72]:
              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 [73]: tips.groupby(['day', 'smoker'], as_index=False).mean()
Out[73]:
    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` method avoids some unnecessary computations.

10.3 Apply: General split-apply-combine

The most general-purpose GroupBy method is `apply`, which is the subject of the rest of this section. As illustrated in Figure 10-2, `apply` splits the object being manipulated into pieces, invokes the passed function on each piece, and then attempts to concatenate the pieces together.

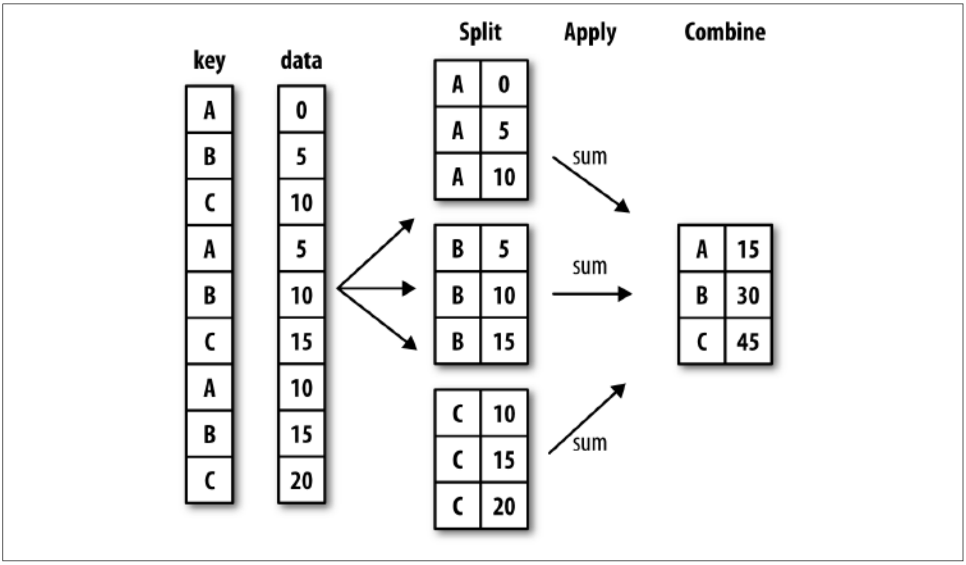

Figure 10-2. Illustration of a group aggregation

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 [74]: def top(df, n=5, column='tip_pct'):
   ....:     return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=6)
Out[75]:
     total_bill   tip smoker  day    time  size   tip_pct
109       14.31  4.00    Yes  Sat  Dinner     2  0.279525
183       23.17  6.50    Yes  Sun  Dinner     4  0.280535
232       11.61  3.39     No  Sat  Dinner     2  0.291990
```

```
67          3.07  1.00    Yes  Sat  Dinner     1  0.325733
178         9.60  4.00    Yes  Sun  Dinner     2  0.416667
172         7.25  5.15    Yes  Sun  Dinner     2  0.710345
```

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

```
In [76]: tips.groupby('smoker').apply(top)
Out[76]:
            total_bill   tip smoker   day    time  size   tip_pct
smoker
No     88        24.71  5.85     No  Thur   Lunch     2  0.236746
       185       20.69  5.00     No   Sun  Dinner     5  0.241663
       51        10.29  2.60     No   Sun  Dinner     2  0.252672
       149        7.51  2.00     No  Thur   Lunch     2  0.266312
       232       11.61  3.39     No   Sat  Dinner     2  0.291990
Yes    109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
       183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
       67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
       178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
       172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
```

What has happened here? The `top` function is called on each row group from the DataFrame, and then the results are glued together using `pandas.concat`, labeling the pieces with the group names. The result therefore has a hierarchical index whose inner level 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 [77]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
Out[77]:
                 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 only needs to return a pandas object or a scalar value. The rest of this chapter will mainly consist of examples showing you how to solve various problems using `groupby`.

You may recall that I earlier called `describe` on a GroupBy object:

```
In [78]: result = tips.groupby('smoker')['tip_pct'].describe()

In [79]: result
Out[79]:
        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 [80]: result.unstack('smoker')
Out[80]:
       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:

```
f = lambda x: x.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 [81]: tips.groupby('smoker', group_keys=False).apply(top)
Out[81]:
     total_bill   tip smoker   day    time  size   tip_pct
88        24.71  5.85     No  Thur   Lunch     2  0.236746
185       20.69  5.00     No   Sun  Dinner     5  0.241663
51        10.29  2.60     No   Sun  Dinner     2  0.252672
149        7.51  2.00     No  Thur   Lunch     2  0.266312
232       11.61  3.39     No   Sat  Dinner     2  0.291990
109       14.31  4.00    Yes   Sat  Dinner     2  0.279525
183       23.17  6.50    Yes   Sun  Dinner     4  0.280535
67         3.07  1.00    Yes   Sat  Dinner     1  0.325733
178        9.60  4.00    Yes   Sun  Dinner     2  0.416667
172        7.25  5.15    Yes   Sun  Dinner     2  0.710345
```

Quantile and Bucket Analysis

As you may recall from Chapter 8, pandas has some tools, in particular `cut` and `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 `cut`:

```
In [82]: frame = pd.DataFrame({'data1': np.random.randn(1000),
   ....:                       'data2': np.random.randn(1000)})

In [83]: quartiles = pd.cut(frame.data1, 4)

In [84]: quartiles[:10]
Out[84]:
0     (-1.23, 0.489]
1    (-2.956, -1.23]
2     (-1.23, 0.489]
3     (0.489, 2.208]
4     (-1.23, 0.489]
5     (0.489, 2.208]
6     (-1.23, 0.489]
7     (-1.23, 0.489]
8     (0.489, 2.208]
9     (0.489, 2.208]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-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 statistics for the `data2` column like so:

```
In [85]: def get_stats(group):
   ....:     return {'min': group.min(), 'max': group.max(),
   ....:             'count': group.count(), 'mean': group.mean()}

In [86]: grouped = frame.data2.groupby(quartiles)
```

```
In [87]: grouped.apply(get_stats).unstack()
Out[87]:
                 count       max      mean       min
data1
(-2.956, -1.23]   95.0  1.670835 -0.039521 -3.399312
(-1.23, 0.489]   598.0  3.260383 -0.002051 -2.989741
(0.489, 2.208]   297.0  2.954439  0.081822 -3.745356
(2.208, 3.928]    10.0  1.765640  0.024750 -1.929776
```

These were equal-length buckets; to compute equal-size buckets based on sample quantiles, use `qcut`. I'll pass `labels=False` to just get quantile numbers:

```
# Return quantile numbers
In [88]: grouping = pd.qcut(frame.data1, 10, labels=False)

In [89]: grouped = frame.data2.groupby(grouping)

In [90]: grouped.apply(get_stats).unstack()
Out[90]:
       count       max      mean       min
data1
0      100.0  1.670835 -0.049902 -3.399312
1      100.0  2.628441  0.030989 -1.950098
2      100.0  2.527939 -0.067179 -2.925113
3      100.0  3.260383  0.065713 -2.315555
4      100.0  2.074345 -0.111653 -2.047939
5      100.0  2.184810  0.052130 -2.989741
6      100.0  2.458842 -0.021489 -2.223506
7      100.0  2.954439 -0.026459 -3.056990
8      100.0  2.735527  0.103406 -3.745356
9      100.0  2.377020  0.220122 -2.064111
```

We will take a closer look at pandas's `Categorical` type in Chapter 12.

Example: Filling Missing Values with Group-Specific Values

When cleaning up missing data, in some cases you will replace data observations using `dropna`, but in others you may want to impute (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 NA values with the mean:

```
In [91]: s = pd.Series(np.random.randn(6))

In [92]: s[::2] = np.nan

In [93]: s
Out[93]:
0         NaN
1   -0.125921
2         NaN
3   -0.884475
```

```
4         NaN
5    0.227290
dtype: float64

In [94]: s.fillna(s.mean())
Out[94]:
0   -0.261035
1   -0.125921
2   -0.261035
3   -0.884475
4   -0.261035
5    0.227290
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 [95]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
   ....:           'Oregon', 'Nevada', 'California', 'Idaho']

In [96]: group_key = ['East'] * 4 + ['West'] * 4

In [97]: data = pd.Series(np.random.randn(8), index=states)

In [98]: data
Out[98]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.365757
Florida      -0.375842
Oregon        0.329939
Nevada        0.981994
California    1.105913
Idaho        -1.613716
dtype: float64
```

Note that the syntax `['East'] * 4` produces a list containing four copies of the elements in `['East']`. Adding lists together concatenates them.

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

```
In [99]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan

In [100]: data
Out[100]:
Ohio          0.922264
New York     -2.153545
Vermont            NaN
Florida      -0.375842
Oregon        0.329939
Nevada             NaN
California    1.105913
```

```
Idaho                  NaN
dtype: float64

In [101]: data.groupby(group_key).mean()
Out[101]:
East   -0.535707
West    0.717926
dtype: float64
```

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

```
In [102]: fill_mean = lambda g: g.fillna(g.mean())

In [103]: data.groupby(group_key).apply(fill_mean)
Out[103]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.535707
Florida      -0.375842
Oregon        0.329939
Nevada        0.717926
California    1.105913
Idaho         0.717926
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 [104]: fill_values = {'East': 0.5, 'West': -1}

In [105]: fill_func = lambda g: g.fillna(fill_values[g.name])

In [106]: data.groupby(group_key).apply(fill_func)
Out[106]:
Ohio          0.922264
New York     -2.153545
Vermont       0.500000
Florida      -0.375842
Oregon        0.329939
Nevada       -1.000000
California    1.105913
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:

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

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

So 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 just let the ace 'A' be 1):

```
In [108]: deck[:13]
Out[108]:
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 [109]: def draw(deck, n=5):
   .....:     return deck.sample(n)

In [110]: draw(deck)
Out[110]:
AD     1
8C     8
5H     5
KC    10
2C     2
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 [111]: get_suit = lambda card: card[-1] # last letter is suit

In [112]: deck.groupby(get_suit).apply(draw, n=2)
Out[112]:
```

```
C  2C     2
   3C     3
D  KD    10
   8D     8
H  KH    10
   3H     3
S  2S     2
   4S     4
dtype: int64
```

Alternatively, we could write:

```
In [113]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[113]:
KC    10
JC    10
AD     1
5D     5
5H     5
6H     6
7S     7
KS    10
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 [114]: df = pd.DataFrame({'category': ['a', 'a', 'a', 'a',
   .....:                                 'b', 'b', 'b', 'b'],
   .....:                    'data': np.random.randn(8),
   .....:                    'weights': np.random.rand(8)})

In [115]: df
Out[115]:
  category      data   weights
0        a  1.561587  0.957515
1        a  1.219984  0.347267
2        a -0.482239  0.581362
3        a  0.315667  0.217091
4        b -0.047852  0.894406
5        b -0.454145  0.918564
6        b -0.556774  0.277825
7        b  0.253321  0.955905
```

The group weighted average by `category` would then be:

```
In [116]: grouped = df.groupby('category')

In [117]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])
```

```
In [118]: grouped.apply(get_wavg)
Out[118]:
category
a    0.811643
b   -0.122262
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 [119]: close_px = pd.read_csv('examples/stock_px_2.csv', parse_dates=True,
   .....:                        index_col=0)

In [120]: close_px.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
AAPL    2214 non-null float64
MSFT    2214 non-null float64
XOM     2214 non-null float64
SPX     2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB

In [121]: close_px[-4:]
Out[121]:
              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
```

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 pairwise correlation of each column with the `'SPX'` column:

```
In [122]: spx_corr = lambda x: x.corrwith(x['SPX'])
```

Next, we compute percent change on `close_px` using `pct_change`:

```
In [123]: 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 [124]: get_year = lambda x: x.year

In [125]: by_year = rets.groupby(get_year)

In [126]: by_year.apply(spx_corr)
Out[126]:
```

```
           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 inter-column correlations. Here we compute the annual correlation between Apple and Microsoft:

```
In [127]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[127]:
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, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

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

```
In [129]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[129]:
           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 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 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 [130]: tips.pivot_table(index=['day', 'smoker'])
Out[130]:
                 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. Now, suppose we want to aggregate only `tip_pct` and `size`, and additionally group by `time`. I'll put `smoker` in the table columns and `day` in the rows:

```
In [131]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
   .....:                  columns='smoker')
Out[131]:
                 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 [132]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
   .....:                  columns='smoker', margins=True)
Out[132]: 
                 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 a different aggregation function, pass it to `aggfunc`. For example, `'count'` or `len` will give you a cross-tabulation (count or frequency) of group sizes:

```
In [133]: tips.pivot_table('tip_pct', index=['time', 'smoker'], columns='day',
   .....:                  aggfunc=len, margins=True)
Out[133]: 
day             Fri   Sat   Sun  Thur    All
time   smoker                               
Dinner No       3.0  45.0  57.0   1.0  106.0
       Yes      9.0  42.0  19.0   NaN   70.0
Lunch  No       1.0   NaN   NaN  44.0   45.0
       Yes      6.0   NaN   NaN  17.0   23.0
All            19.0  87.0  76.0  62.0  244.0
```

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

```
In [134]: tips.pivot_table('tip_pct', index=['time', 'size', 'smoker'],
   .....:                  columns='day', aggfunc='mean', fill_value=0)
Out[134]: 
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
```

```
              Yes      0.000000  0.144995  0.152660  0.000000
       4      No       0.000000  0.150096  0.148143  0.000000
              Yes      0.117750  0.124515  0.193370  0.000000
       5      No       0.000000  0.000000  0.206928  0.000000
              Yes      0.000000  0.106572  0.065660  0.000000
...                         ...       ...       ...       ...
Lunch  1      No       0.000000  0.000000  0.000000  0.181728
              Yes      0.223776  0.000000  0.000000  0.000000
       2      No       0.000000  0.000000  0.000000  0.166005
              Yes      0.181969  0.000000  0.000000  0.158843
       3      No       0.187735  0.000000  0.000000  0.084246
              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` methods.

Table 10-2. pivot_table options

Function name	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 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)

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 [138]: data
Out[138]:
   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 [139]: pd.crosstab(data.Nationality, data.Handedness, margins=True)
Out[139]:
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 either be an array or Series or a list of arrays. As in the tips data:

```
In [140]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
Out[140]:
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
```

10.5 Conclusion

Mastering pandas's data grouping tools can help both with data cleaning as well as modeling or statistical analysis work. In Chapter 14 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.

CHAPTER 11

Time Series

Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, and physics. Anything that is observed or measured at many points in time forms a time series. Many time series are *fixed frequency*, which is to say that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once per month. Time series can also be *irregular* without a fixed unit of time or offset between units. How you mark and refer to time series data depends on the application, and you may have one of the following:

- *Timestamps*, specific instants in time
- Fixed *periods*, such as the month January 2007 or the full year 2010
- *Intervals* of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals
- Experiment or elapsed time; each timestamp is a measure of time relative to a particular start time (e.g., the diameter of a cookie baking each second since being placed in the oven)

In this chapter, I am mainly concerned with time series in the first three categories, though many of the techniques can be applied to experimental time series where the index may be an integer or floating-point number indicating elapsed time from the start of the experiment. The simplest and most widely used kind of time series are those indexed by timestamp.

pandas also supports indexes based on timedeltas, which can be a useful way of representing experiment or elapsed time. We do not explore timedelta indexes in this book, but you can learn more in the pandas documentation.

pandas provides many built-in time series tools and data algorithms. You can efficiently work with very large time series and easily slice and dice, aggregate, and resample irregular- and fixed-frequency time series. Some of these tools are especially useful for financial and economics applications, but you could certainly use them to analyze server log data, too.

11.1 Date and Time Data Types and Tools

The Python standard library includes data types for date and time data, as well as calendar-related functionality. The `datetime`, `time`, and `calendar` modules are the main places to start. The `datetime.datetime` type, or simply `datetime`, is widely used:

```
In [10]: from datetime import datetime

In [11]: now = datetime.now()

In [12]: now
Out[12]: datetime.datetime(2017, 9, 25, 14, 5, 52, 72973)

In [13]: now.year, now.month, now.day
Out[13]: (2017, 9, 25)
```

`datetime` stores both the date and time down to the microsecond. `timedelta` represents the temporal difference between two `datetime` objects:

```
In [14]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [15]: delta
Out[15]: datetime.timedelta(926, 56700)

In [16]: delta.days
Out[16]: 926

In [17]: delta.seconds
Out[17]: 56700
```

You can add (or subtract) a `timedelta` or multiple thereof to a `datetime` object to yield a new shifted object:

```
In [18]: from datetime import timedelta

In [19]: start = datetime(2011, 1, 7)
```

```
In [20]: start + timedelta(12)
Out[20]: datetime.datetime(2011, 1, 19, 0, 0)

In [21]: start - 2 * timedelta(12)
Out[21]: datetime.datetime(2010, 12, 14, 0, 0)
```

Table 11-1 summarizes the data types in the `datetime` module. While this chapter is mainly concerned with the data types in pandas and higher-level time series manipulation, you may encounter the `datetime`-based types in many other places in Python in the wild.

Table 11-1. Types in datetime module

Type	Description
`date`	Store calendar date (year, month, day) using the Gregorian calendar
`time`	Store time of day as hours, minutes, seconds, and microseconds
`datetime`	Stores both date and time
`timedelta`	Represents the difference between two `datetime` values (as days, seconds, and microseconds)
`tzinfo`	Base type for storing time zone information

Converting Between String and Datetime

You can format `datetime` objects and pandas `Timestamp` objects, which I'll introduce later, as strings using `str` or the `strftime` method, passing a format specification:

```
In [22]: stamp = datetime(2011, 1, 3)

In [23]: str(stamp)
Out[23]: '2011-01-03 00:00:00'

In [24]: stamp.strftime('%Y-%m-%d')
Out[24]: '2011-01-03'
```

See Table 11-2 for a complete list of the format codes (reproduced from Chapter 2).

Table 11-2. Datetime format specification (ISO C89 compatible)

Type	Description
`%Y`	Four-digit year
`%y`	Two-digit year
`%m`	Two-digit month [01, 12]
`%d`	Two-digit day [01, 31]
`%H`	Hour (24-hour clock) [00, 23]
`%I`	Hour (12-hour clock) [01, 12]
`%M`	Two-digit minute [00, 59]
`%S`	Second [00, 61] (seconds 60, 61 account for leap seconds)
`%w`	Weekday as integer [0 (Sunday), 6]

Type	Description
%U	Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are "week 0"
%W	Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are "week 0"
%z	UTC time zone offset as +HHMM or -HHMM; empty if time zone naive
%F	Shortcut for %Y-%m-%d (e.g., 2012-4-18)
%D	Shortcut for %m/%d/%y (e.g., 04/18/12)

You can use these same format codes to convert strings to dates using `datetime.strptime`:

```
In [25]: value = '2011-01-03'

In [26]: datetime.strptime(value, '%Y-%m-%d')
Out[26]: datetime.datetime(2011, 1, 3, 0, 0)

In [27]: datestrs = ['7/6/2011', '8/6/2011']

In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[28]:
[datetime.datetime(2011, 7, 6, 0, 0),
 datetime.datetime(2011, 8, 6, 0, 0)]
```

`datetime.strptime` is a good way to parse a date with a known format. However, it can be a bit annoying to have to write a format spec each time, especially for common date formats. In this case, you can use the `parser.parse` method in the third-party `dateutil` package (this is installed automatically when you install pandas):

```
In [29]: from dateutil.parser import parse

In [30]: parse('2011-01-03')
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)
```

`dateutil` is capable of parsing most human-intelligible date representations:

```
In [31]: parse('Jan 31, 1997 10:45 PM')
Out[31]: datetime.datetime(1997, 1, 31, 22, 45)
```

In international locales, day appearing before month is very common, so you can pass `dayfirst=True` to indicate this:

```
In [32]: parse('6/12/2011', dayfirst=True)
Out[32]: datetime.datetime(2011, 12, 6, 0, 0)
```

pandas is generally oriented toward working with arrays of dates, whether used as an axis index or a column in a DataFrame. The `to_datetime` method parses many different kinds of date representations. Standard date formats like ISO 8601 can be parsed very quickly:

```
In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']

In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'], dtype='dat
etime64[ns]', freq=None)
```

It also handles values that should be considered missing (`None`, empty string, etc.):

```
In [35]: idx = pd.to_datetime(datestrs + [None])

In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', 'NaT'], dty
pe='datetime64[ns]', freq=None)

In [37]: idx[2]
Out[37]: NaT

In [38]: pd.isnull(idx)
Out[38]: array([False, False,  True], dtype=bool)
```

`NaT` (Not a Time) is pandas's null value for timestamp data.

> `dateutil.parser` is a useful but imperfect tool. Notably, it will recognize some strings as dates that you might prefer that it didn't—for example, `'42'` will be parsed as the year `2042` with today's calendar date.

`datetime` objects also have a number of locale-specific formatting options for systems in other countries or languages. For example, the abbreviated month names will be different on German or French systems compared with English systems. See Table 11-3 for a listing.

Table 11-3. Locale-specific date formatting

Type	Description
`%a`	Abbreviated weekday name
`%A`	Full weekday name
`%b`	Abbreviated month name
`%B`	Full month name
`%c`	Full date and time (e.g., 'Tue 01 May 2012 04:20:57 PM')
`%p`	Locale equivalent of AM or PM
`%x`	Locale-appropriate formatted date (e.g., in the United States, May 1, 2012 yields '05/01/2012')
`%X`	Locale-appropriate time (e.g., '04:24:12 PM')

11.2 Time Series Basics

A basic kind of time series object in pandas is a Series indexed by timestamps, which is often represented external to pandas as Python strings or `datetime` objects:

```
In [39]: from datetime import datetime

In [40]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),
   ....:          datetime(2011, 1, 7), datetime(2011, 1, 8),
   ....:          datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [41]: ts = pd.Series(np.random.randn(6), index=dates)

In [42]: ts
Out[42]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

Under the hood, these `datetime` objects have been put in a `DatetimeIndex`:

```
In [43]: ts.index
Out[43]:
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

Like other Series, arithmetic operations between differently indexed time series automatically align on the dates:

```
In [44]: ts + ts[::2]
Out[44]:
2011-01-02   -0.409415
2011-01-05         NaN
2011-01-07   -1.038877
2011-01-08         NaN
2011-01-10    3.931561
2011-01-12         NaN
dtype: float64
```

Recall that `ts[::2]` selects every second element in `ts`.

pandas stores timestamps using NumPy's `datetime64` data type at the nanosecond resolution:

```
In [45]: ts.index.dtype
Out[45]: dtype('<M8[ns]')
```

Scalar values from a `DatetimeIndex` are pandas `Timestamp` objects:

```
In [46]: stamp = ts.index[0]

In [47]: stamp
Out[47]: Timestamp('2011-01-02 00:00:00')
```

A `Timestamp` can be substituted anywhere you would use a `datetime` object. Additionally, it can store frequency information (if any) and understands how to do time zone conversions and other kinds of manipulations. More on both of these things later.

Indexing, Selection, Subsetting

Time series behaves like any other `pandas.Series` when you are indexing and selecting data based on label:

```
In [48]: stamp = ts.index[2]

In [49]: ts[stamp]
Out[49]: -0.51943871505673811
```

As a convenience, you can also pass a string that is interpretable as a date:

```
In [50]: ts['1/10/2011']
Out[50]: 1.9657805725027142

In [51]: ts['20110110']
Out[51]: 1.9657805725027142
```

For longer time series, a year or only a year and month can be passed to easily select slices of data:

```
In [52]: longer_ts = pd.Series(np.random.randn(1000),
   ....:                       index=pd.date_range('1/1/2000', periods=1000))

In [53]: longer_ts
Out[53]: 
2000-01-01    0.092908
2000-01-02    0.281746
2000-01-03    0.769023
2000-01-04    1.246435
2000-01-05    1.007189
2000-01-06   -1.296221
2000-01-07    0.274992
2000-01-08    0.228913
2000-01-09    1.352917
2000-01-10    0.886429
                ...   
2002-09-17   -0.139298
2002-09-18   -1.159926
2002-09-19    0.618965
2002-09-20    1.373890
2002-09-21   -0.983505
```

```
2002-09-22    0.930944
2002-09-23   -0.811676
2002-09-24   -1.830156
2002-09-25   -0.138730
2002-09-26    0.334088
Freq: D, Length: 1000, dtype: float64

In [54]: longer_ts['2001']
Out[54]:
2001-01-01    1.599534
2001-01-02    0.474071
2001-01-03    0.151326
2001-01-04   -0.542173
2001-01-05   -0.475496
2001-01-06    0.106403
2001-01-07   -1.308228
2001-01-08    2.173185
2001-01-09    0.564561
2001-01-10   -0.190481
                ...
2001-12-22    0.000369
2001-12-23    0.900885
2001-12-24   -0.454869
2001-12-25   -0.864547
2001-12-26    1.129120
2001-12-27    0.057874
2001-12-28   -0.433739
2001-12-29    0.092698
2001-12-30   -1.397820
2001-12-31    1.457823
Freq: D, Length: 365, dtype: float64
```

Here, the string '2001' is interpreted as a year and selects that time period. This also works if you specify the month:

```
In [55]: longer_ts['2001-05']
Out[55]:
2001-05-01   -0.622547
2001-05-02    0.936289
2001-05-03    0.750018
2001-05-04   -0.056715
2001-05-05    2.300675
2001-05-06    0.569497
2001-05-07    1.489410
2001-05-08    1.264250
2001-05-09   -0.761837
2001-05-10   -0.331617
                ...
2001-05-22    0.503699
2001-05-23   -1.387874
2001-05-24    0.204851
2001-05-25    0.603705
2001-05-26    0.545680
```

```
2001-05-27    0.235477
2001-05-28    0.111835
2001-05-29   -1.251504
2001-05-30   -2.949343
2001-05-31    0.634634
Freq: D, Length: 31, dtype: float64
```

Slicing with `datetime` objects works as well:

```
In [56]: ts[datetime(2011, 1, 7):]
Out[56]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

Because most time series data is ordered chronologically, you can slice with time-stamps not contained in a time series to perform a range query:

```
In [57]: ts
Out[57]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64

In [58]: ts['1/6/2011':'1/11/2011']
Out[58]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
dtype: float64
```

As before, you can pass either a string date, `datetime`, or timestamp. Remember that slicing in this manner produces views on the source time series like slicing NumPy arrays. This means that no data is copied and modifications on the slice will be reflected in the original data.

There is an equivalent instance method, `truncate`, that slices a Series between two dates:

```
In [59]: ts.truncate(after='1/9/2011')
Out[59]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
dtype: float64
```

All of this holds true for DataFrame as well, indexing on its rows:

```
In [60]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')

In [61]: long_df = pd.DataFrame(np.random.randn(100, 4),
   ....:                        index=dates,
   ....:                        columns=['Colorado', 'Texas',
   ....:                                 'New York', 'Ohio'])

In [62]: long_df.loc['5-2001']
Out[62]:
            Colorado     Texas  New York      Ohio
2001-05-02 -0.006045  0.490094 -0.277186 -0.707213
2001-05-09 -0.560107  2.735527  0.927335  1.513906
2001-05-16  0.538600  1.273768  0.667876 -0.969206
2001-05-23  1.676091 -0.817649  0.050188  1.951312
2001-05-30  3.260383  0.963301  1.201206 -1.852001
```

Time Series with Duplicate Indices

In some applications, there may be multiple data observations falling on a particular timestamp. Here is an example:

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000',
   ....:                           '1/2/2000', '1/3/2000'])

In [64]: dup_ts = pd.Series(np.arange(5), index=dates)

In [65]: dup_ts
Out[65]:
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int64
```

We can tell that the index is not unique by checking its `is_unique` property:

```
In [66]: dup_ts.index.is_unique
Out[66]: False
```

Indexing into this time series will now either produce scalar values or slices depending on whether a timestamp is duplicated:

```
In [67]: dup_ts['1/3/2000']  # not duplicated
Out[67]: 4

In [68]: dup_ts['1/2/2000']  # duplicated
Out[68]:
2000-01-02    1
2000-01-02    2
```

```
2000-01-02    3
dtype: int64
```

Suppose you wanted to aggregate the data having non-unique timestamps. One way to do this is to use `groupby` and pass `level=0`:

```
In [69]: grouped = dup_ts.groupby(level=0)

In [70]: grouped.mean()
Out[70]:
2000-01-01    0
2000-01-02    2
2000-01-03    4
dtype: int64

In [71]: grouped.count()
Out[71]:
2000-01-01    1
2000-01-02    3
2000-01-03    1
dtype: int64
```

11.3 Date Ranges, Frequencies, and Shifting

Generic time series in pandas are assumed to be irregular; that is, they have no fixed frequency. For many applications this is sufficient. However, it's often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes, even if that means introducing missing values into a time series. Fortunately pandas has a full suite of standard time series frequencies and tools for resampling, inferring frequencies, and generating fixed-frequency date ranges. For example, you can convert the sample time series to be fixed daily frequency by calling `resample`:

```
In [72]: ts
Out[72]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64

In [73]: resampler = ts.resample('D')
```

The string `'D'` is interpreted as daily frequency.

Conversion between frequencies or *resampling* is a big enough topic to have its own section later (Section 11.6, "Resampling and Frequency Conversion," on page 348). Here I'll show you how to use the base frequencies and multiples thereof.

Generating Date Ranges

While I used it previously without explanation, `pandas.date_range` is responsible for generating a `DatetimeIndex` with an indicated length according to a particular frequency:

```
In [74]: index = pd.date_range('2012-04-01', '2012-06-01')

In [75]: index
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
               '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
               '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
               '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
               '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
               '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
               '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
               '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
               '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
               '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
               '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
               '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

By default, `date_range` generates daily timestamps. If you pass only a start or end date, you must pass a number of periods to generate:

```
In [76]: pd.date_range(start='2012-04-01', periods=20)
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              dtype='datetime64[ns]', freq='D')

In [77]: pd.date_range(end='2012-06-01', periods=20)
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-05-16',
               '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',
               '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',
               '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',
               '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

The start and end dates define strict boundaries for the generated date index. For example, if you wanted a date index containing the last business day of each month, you would pass the `'BM'` frequency (business end of month; see more complete listing of frequencies in Table 11-4) and only dates falling on or inside the date interval will be included:

```
In [78]: pd.date_range('2000-01-01', '2000-12-01', freq='BM')
Out[78]:
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-04-28',
               '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',
               '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

Table 11-4. Base time series frequencies (not comprehensive)

Alias	Offset type	Description
`D`	`Day`	Calendar daily
`B`	`BusinessDay`	Business daily
`H`	`Hour`	Hourly
`T` or `min`	`Minute`	Minutely
`S`	`Second`	Secondly
`L` or `ms`	`Milli`	Millisecond (1/1,000 of 1 second)
`U`	`Micro`	Microsecond (1/1,000,000 of 1 second)
`M`	`MonthEnd`	Last calendar day of month
`BM`	`BusinessMonthEnd`	Last business day (weekday) of month
`MS`	`MonthBegin`	First calendar day of month
`BMS`	`BusinessMonthBegin`	First weekday of month
`W-MON, W-TUE, ...`	`Week`	Weekly on given day of week (MON, TUE, WED, THU, FRI, SAT, or SUN)
`WOM-1MON, WOM-2MON, ...`	`WeekOfMonth`	Generate weekly dates in the first, second, third, or fourth week of the month (e.g., `WOM-3FRI` for the third Friday of each month)
`Q-JAN, Q-FEB, ...`	`QuarterEnd`	Quarterly dates anchored on last calendar day of each month, for year ending in indicated month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
`BQ-JAN, BQ-FEB, ...`	`BusinessQuarterEnd`	Quarterly dates anchored on last weekday day of each month, for year ending in indicated month
`QS-JAN, QS-FEB, ...`	`QuarterBegin`	Quarterly dates anchored on first calendar day of each month, for year ending in indicated month
`BQS-JAN, BQS-FEB, ...`	`BusinessQuarterBegin`	Quarterly dates anchored on first weekday day of each month, for year ending in indicated month
`A-JAN, A-FEB, ...`	`YearEnd`	Annual dates anchored on last calendar day of given month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
`BA-JAN, BA-FEB, ...`	`BusinessYearEnd`	Annual dates anchored on last weekday of given month
`AS-JAN, AS-FEB, ...`	`YearBegin`	Annual dates anchored on first day of given month
`BAS-JAN, BAS-FEB, ...`	`BusinessYearBegin`	Annual dates anchored on first weekday of given month

`date_range` by default preserves the time (if any) of the start or end timestamp:

```
In [79]: pd.date_range('2012-05-02 12:56:31', periods=5)
Out[79]:
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
               '2012-05-04 12:56:31', '2012-05-05 12:56:31',
               '2012-05-06 12:56:31'],
              dtype='datetime64[ns]', freq='D')
```

Sometimes you will have start or end dates with time information but want to generate a set of timestamps *normalized* to midnight as a convention. To do this, there is a `normalize` option:

```
In [80]: pd.date_range('2012-05-02 12:56:31', periods=5, normalize=True)
Out[80]:
DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-05-05',
               '2012-05-06'],
              dtype='datetime64[ns]', freq='D')
```

Frequencies and Date Offsets

Frequencies in pandas are composed of a *base frequency* and a multiplier. Base frequencies are typically referred to by a string alias, like `'M'` for monthly or `'H'` for hourly. For each base frequency, there is an object defined generally referred to as a *date offset*. For example, hourly frequency can be represented with the `Hour` class:

```
In [81]: from pandas.tseries.offsets import Hour, Minute

In [82]: hour = Hour()

In [83]: hour
Out[83]: <Hour>
```

You can define a multiple of an offset by passing an integer:

```
In [84]: four_hours = Hour(4)

In [85]: four_hours
Out[85]: <4 * Hours>
```

In most applications, you would never need to explicitly create one of these objects, instead using a string alias like `'H'` or `'4H'`. Putting an integer before the base frequency creates a multiple:

```
In [86]: pd.date_range('2000-01-01', '2000-01-03 23:59', freq='4h')
Out[86]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
               '2000-01-01 08:00:00', '2000-01-01 12:00:00',
               '2000-01-01 16:00:00', '2000-01-01 20:00:00',
               '2000-01-02 00:00:00', '2000-01-02 04:00:00',
               '2000-01-02 08:00:00', '2000-01-02 12:00:00',
               '2000-01-02 16:00:00', '2000-01-02 20:00:00',
```

```
                   '2000-01-03 00:00:00', '2000-01-03 04:00:00',
                   '2000-01-03 08:00:00', '2000-01-03 12:00:00',
                   '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
                  dtype='datetime64[ns]', freq='4H')
```

Many offsets can be combined together by addition:

```
In [87]: Hour(2) + Minute(30)
Out[87]: <150 * Minutes>
```

Similarly, you can pass frequency strings, like `'1h30min'`, that will effectively be parsed to the same expression:

```
In [88]: pd.date_range('2000-01-01', periods=10, freq='1h30min')
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
               '2000-01-01 03:00:00', '2000-01-01 04:30:00',
               '2000-01-01 06:00:00', '2000-01-01 07:30:00',
               '2000-01-01 09:00:00', '2000-01-01 10:30:00',
               '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
              dtype='datetime64[ns]', freq='90T')
```

Some frequencies describe points in time that are not evenly spaced. For example, `'M'` (calendar month end) and `'BM'` (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. We refer to these as *anchored* offsets.

Refer back to Table 11-4 for a listing of frequency codes and date offset classes available in pandas.

> Users can define their own custom frequency classes to provide date logic not available in pandas, though the full details of that are outside the scope of this book.

Week of month dates

One useful frequency class is "week of month," starting with `WOM`. This enables you to get dates like the third Friday of each month:

```
In [89]: rng = pd.date_range('2012-01-01', '2012-09-01', freq='WOM-3FRI')

In [90]: list(rng)
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),
```

```
 Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

Shifting (Leading and Lagging) Data

"Shifting" refers to moving data backward and forward through time. Both Series and DataFrame have a `shift` method for doing naive shifts forward or backward, leaving the index unmodified:

```
In [91]: ts = pd.Series(np.random.randn(4),
   ....:                index=pd.date_range('1/1/2000', periods=4, freq='M'))

In [92]: ts
Out[92]:
2000-01-31   -0.066748
2000-02-29    0.838639
2000-03-31   -0.117388
2000-04-30   -0.517795
Freq: M, dtype: float64

In [93]: ts.shift(2)
Out[93]:
2000-01-31         NaN
2000-02-29         NaN
2000-03-31   -0.066748
2000-04-30    0.838639
Freq: M, dtype: float64

In [94]: ts.shift(-2)
Out[94]:
2000-01-31   -0.117388
2000-02-29   -0.517795
2000-03-31         NaN
2000-04-30         NaN
Freq: M, dtype: float64
```

When we shift like this, missing data is introduced either at the start or the end of the time series.

A common use of `shift` is computing percent changes in a time series or multiple time series as DataFrame columns. This is expressed as:

```
ts / ts.shift(1) - 1
```

Because naive shifts leave the index unmodified, some data is discarded. Thus if the frequency is known, it can be passed to `shift` to advance the timestamps instead of simply the data:

```
In [95]: ts.shift(2, freq='M')
Out[95]:
2000-03-31   -0.066748
2000-04-30    0.838639
```

```
2000-05-31   -0.117388
2000-06-30   -0.517795
Freq: M, dtype: float64
```

Other frequencies can be passed, too, giving you some flexibility in how to lead and lag the data:

```
In [96]: ts.shift(3, freq='D')
Out[96]:
2000-02-03   -0.066748
2000-03-03    0.838639
2000-04-03   -0.117388
2000-05-03   -0.517795
dtype: float64

In [97]: ts.shift(1, freq='90T')
Out[97]:
2000-01-31 01:30:00   -0.066748
2000-02-29 01:30:00    0.838639
2000-03-31 01:30:00   -0.117388
2000-04-30 01:30:00   -0.517795
Freq: M, dtype: float64
```

The `T` here stands for minutes.

Shifting dates with offsets

The pandas date offsets can also be used with `datetime` or `Timestamp` objects:

```
In [98]: from pandas.tseries.offsets import Day, MonthEnd

In [99]: now = datetime(2011, 11, 17)

In [100]: now + 3 * Day()
Out[100]: Timestamp('2011-11-20 00:00:00')
```

If you add an anchored offset like `MonthEnd`, the first increment will "roll forward" a date to the next date according to the frequency rule:

```
In [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')

In [102]: now + MonthEnd(2)
Out[102]: Timestamp('2011-12-31 00:00:00')
```

Anchored offsets can explicitly "roll" dates forward or backward by simply using their `rollforward` and `rollback` methods, respectively:

```
In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)
Out[104]: Timestamp('2011-11-30 00:00:00')
```

```
In [105]: offset.rollback(now)
Out[105]: Timestamp('2011-10-31 00:00:00')
```

A creative use of date offsets is to use these methods with `groupby`:

```
In [106]: ts = pd.Series(np.random.randn(20),
   .....:                index=pd.date_range('1/15/2000', periods=20, freq='4d'))

In [107]: ts
Out[107]:
2000-01-15   -0.116696
2000-01-19    2.389645
2000-01-23   -0.932454
2000-01-27   -0.229331
2000-01-31   -1.140330
2000-02-04    0.439920
2000-02-08   -0.823758
2000-02-12   -0.520930
2000-02-16    0.350282
2000-02-20    0.204395
2000-02-24    0.133445
2000-02-28    0.327905
2000-03-03    0.072153
2000-03-07    0.131678
2000-03-11   -1.297459
2000-03-15    0.997747
2000-03-19    0.870955
2000-03-23   -0.991253
2000-03-27    0.151699
2000-03-31    1.266151
Freq: 4D, dtype: float64

In [108]: ts.groupby(offset.rollforward).mean()
Out[108]:
2000-01-31   -0.005833
2000-02-29    0.015894
2000-03-31    0.150209
dtype: float64
```

Of course, an easier and faster way to do this is using `resample` (we'll discuss this in much more depth in Section 11.6, "Resampling and Frequency Conversion," on page 348):

```
In [109]: ts.resample('M').mean()
Out[109]:
2000-01-31   -0.005833
2000-02-29    0.015894
2000-03-31    0.150209
Freq: M, dtype: float64
```

11.4 Time Zone Handling

Working with time zones is generally considered one of the most unpleasant parts of time series manipulation. As a result, many time series users choose to work with time series in *coordinated universal time* or *UTC*, which is the successor to Greenwich Mean Time and is the current international standard. Time zones are expressed as offsets from UTC; for example, New York is four hours behind UTC during daylight saving time and five hours behind the rest of the year.

In Python, time zone information comes from the third-party `pytz` library (installable with pip or conda), which exposes the *Olson database*, a compilation of world time zone information. This is especially important for historical data because the daylight saving time (DST) transition dates (and even UTC offsets) have been changed numerous times depending on the whims of local governments. In the United States, the DST transition times have been changed many times since 1900!

For detailed information about the `pytz` library, you'll need to look at that library's documentation. As far as this book is concerned, pandas wraps `pytz`'s functionality so you can ignore its API outside of the time zone names. Time zone names can be found interactively and in the docs:

```
In [110]: import pytz

In [111]: pytz.common_timezones[-5:]
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

To get a time zone object from `pytz`, use `pytz.timezone`:

```
In [112]: tz = pytz.timezone('America/New_York')

In [113]: tz
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Methods in pandas will accept either time zone names or these objects.

Time Zone Localization and Conversion

By default, time series in pandas are *time zone naive*. For example, consider the following time series:

```
In [114]: rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')

In [115]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [116]: ts
Out[116]: 
2012-03-09 09:30:00   -0.202469
2012-03-10 09:30:00    0.050718
2012-03-11 09:30:00    0.639869
2012-03-12 09:30:00    0.597594
```

```
2012-03-13 09:30:00   -0.797246
2012-03-14 09:30:00    0.472879
Freq: D, dtype: float64
```

The index's `tz` field is `None`:

```
In [117]: print(ts.index.tz)
None
```

Date ranges can be generated with a time zone set:

```
In [118]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
Out[118]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',
               '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

Conversion from naive to *localized* is handled by the `tz_localize` method:

```
In [119]: ts
Out[119]:
2012-03-09 09:30:00   -0.202469
2012-03-10 09:30:00    0.050718
2012-03-11 09:30:00    0.639869
2012-03-12 09:30:00    0.597594
2012-03-13 09:30:00   -0.797246
2012-03-14 09:30:00    0.472879
Freq: D, dtype: float64

In [120]: ts_utc = ts.tz_localize('UTC')

In [121]: ts_utc
Out[121]:
2012-03-09 09:30:00+00:00   -0.202469
2012-03-10 09:30:00+00:00    0.050718
2012-03-11 09:30:00+00:00    0.639869
2012-03-12 09:30:00+00:00    0.597594
2012-03-13 09:30:00+00:00   -0.797246
2012-03-14 09:30:00+00:00    0.472879
Freq: D, dtype: float64

In [122]: ts_utc.index
Out[122]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

Once a time series has been localized to a particular time zone, it can be converted to another time zone with `tz_convert`:

```
In [123]: ts_utc.tz_convert('America/New_York')
Out[123]:
2012-03-09 04:30:00-05:00   -0.202469
2012-03-10 04:30:00-05:00    0.050718
2012-03-11 05:30:00-04:00    0.639869
2012-03-12 05:30:00-04:00    0.597594
2012-03-13 05:30:00-04:00   -0.797246
2012-03-14 05:30:00-04:00    0.472879
Freq: D, dtype: float64
```

In the case of the preceding time series, which straddles a DST transition in the `America/New_York` time zone, we could localize to EST and convert to, say, UTC or Berlin time:

```
In [124]: ts_eastern = ts.tz_localize('America/New_York')

In [125]: ts_eastern.tz_convert('UTC')
Out[125]:
2012-03-09 14:30:00+00:00   -0.202469
2012-03-10 14:30:00+00:00    0.050718
2012-03-11 13:30:00+00:00    0.639869
2012-03-12 13:30:00+00:00    0.597594
2012-03-13 13:30:00+00:00   -0.797246
2012-03-14 13:30:00+00:00    0.472879
Freq: D, dtype: float64

In [126]: ts_eastern.tz_convert('Europe/Berlin')
Out[126]:
2012-03-09 15:30:00+01:00   -0.202469
2012-03-10 15:30:00+01:00    0.050718
2012-03-11 14:30:00+01:00    0.639869
2012-03-12 14:30:00+01:00    0.597594
2012-03-13 14:30:00+01:00   -0.797246
2012-03-14 14:30:00+01:00    0.472879
Freq: D, dtype: float64
```

`tz_localize` and `tz_convert` are also instance methods on `DatetimeIndex`:

```
In [127]: ts.index.tz_localize('Asia/Shanghai')
Out[127]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',
               '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',
               '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],
              dtype='datetime64[ns, Asia/Shanghai]', freq='D')
```

Localizing naive timestamps also checks for ambiguous or non-existent times around daylight saving time transitions.

Operations with Time Zone–Aware Timestamp Objects

Similar to time series and date ranges, individual `Timestamp` objects similarly can be localized from naive to time zone–aware and converted from one time zone to another:

```
In [128]: stamp = pd.Timestamp('2011-03-12 04:00')

In [129]: stamp_utc = stamp.tz_localize('utc')

In [130]: stamp_utc.tz_convert('America/New_York')
Out[130]: Timestamp('2011-03-11 23:00:00-0500', tz='America/New_York')
```

You can also pass a time zone when creating the `Timestamp`:

```
In [131]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')

In [132]: stamp_moscow
Out[132]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')
```

Time zone–aware `Timestamp` objects internally store a UTC timestamp value as nanoseconds since the Unix epoch (January 1, 1970); this UTC value is invariant between time zone conversions:

```
In [133]: stamp_utc.value
Out[133]: 1299902400000000000

In [134]: stamp_utc.tz_convert('America/New_York').value
Out[134]: 1299902400000000000
```

When performing time arithmetic using pandas's `DateOffset` objects, pandas respects daylight saving time transitions where possible. Here we construct timestamps that occur right before DST transitions (forward and backward). First, 30 minutes before transitioning to DST:

```
In [135]: from pandas.tseries.offsets import Hour

In [136]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')

In [137]: stamp
Out[137]: Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern')

In [138]: stamp + Hour()
Out[138]: Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern')
```

Then, 90 minutes before transitioning out of DST:

```
In [139]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')

In [140]: stamp
Out[140]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')
```

```
In [141]: stamp + 2 * Hour()
Out[141]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')
```

Operations Between Different Time Zones

If two time series with different time zones are combined, the result will be UTC. Since the timestamps are stored under the hood in UTC, this is a straightforward operation and requires no conversion to happen:

```
In [142]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')

In [143]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [144]: ts
Out[144]: 
2012-03-07 09:30:00    0.522356
2012-03-08 09:30:00   -0.546348
2012-03-09 09:30:00   -0.733537
2012-03-12 09:30:00    1.302736
2012-03-13 09:30:00    0.022199
2012-03-14 09:30:00    0.364287
2012-03-15 09:30:00   -0.922839
2012-03-16 09:30:00    0.312656
2012-03-19 09:30:00   -1.128497
2012-03-20 09:30:00   -0.333488
Freq: B, dtype: float64

In [145]: ts1 = ts[:7].tz_localize('Europe/London')

In [146]: ts2 = ts1[2:].tz_convert('Europe/Moscow')

In [147]: result = ts1 + ts2

In [148]: result.index
Out[148]: 
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',
               '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='B')
```

11.5 Periods and Period Arithmetic

Periods represent timespans, like days, months, quarters, or years. The `Period` class represents this data type, requiring a string or integer and a frequency from Table 11-4:

```
In [149]: p = pd.Period(2007, freq='A-DEC')

In [150]: p
Out[150]: Period('2007', 'A-DEC')
```

In this case, the `Period` object represents the full timespan from January 1, 2007, to December 31, 2007, inclusive. Conveniently, adding and subtracting integers from periods has the effect of shifting by their frequency:

```
In [151]: p + 5
Out[151]: Period('2012', 'A-DEC')

In [152]: p - 2
Out[152]: Period('2005', 'A-DEC')
```

If two periods have the same frequency, their difference is the number of units between them:

```
In [153]: pd.Period('2014', freq='A-DEC') - p
Out[153]: 7
```

Regular ranges of periods can be constructed with the `period_range` function:

```
In [154]: rng = pd.period_range('2000-01-01', '2000-06-30', freq='M')

In [155]: rng
Out[155]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2000-05', '20
00-06'], dtype='period[M]', freq='M')
```

The `PeriodIndex` class stores a sequence of periods and can serve as an axis index in any pandas data structure:

```
In [156]: pd.Series(np.random.randn(6), index=rng)
Out[156]:
2000-01   -0.514551
2000-02   -0.559782
2000-03   -0.783408
2000-04   -1.797685
2000-05   -0.172670
2000-06    0.680215
Freq: M, dtype: float64
```

If you have an array of strings, you can also use the `PeriodIndex` class:

```
In [157]: values = ['2001Q3', '2002Q2', '2003Q1']

In [158]: index = pd.PeriodIndex(values, freq='Q-DEC')

In [159]: index
Out[159]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='period[Q-DEC]', freq
='Q-DEC')
```

Period Frequency Conversion

Periods and `PeriodIndex` objects can be converted to another frequency with their `asfreq` method. As an example, suppose we had an annual period and wanted to convert it into a monthly period either at the start or end of the year. This is fairly straightforward:

```
In [160]: p = pd.Period('2007', freq='A-DEC')

In [161]: p
Out[161]: Period('2007', 'A-DEC')

In [162]: p.asfreq('M', how='start')
Out[162]: Period('2007-01', 'M')

In [163]: p.asfreq('M', how='end')
Out[163]: Period('2007-12', 'M')
```

You can think of `Period('2007', 'A-DEC')` as being a sort of cursor pointing to a span of time, subdivided by monthly periods. See Figure 11-1 for an illustration of this. For a *fiscal year* ending on a month other than December, the corresponding monthly subperiods are different:

```
In [164]: p = pd.Period('2007', freq='A-JUN')

In [165]: p
Out[165]: Period('2007', 'A-JUN')

In [166]: p.asfreq('M', 'start')
Out[166]: Period('2006-07', 'M')

In [167]: p.asfreq('M', 'end')
Out[167]: Period('2007-06', 'M')
```

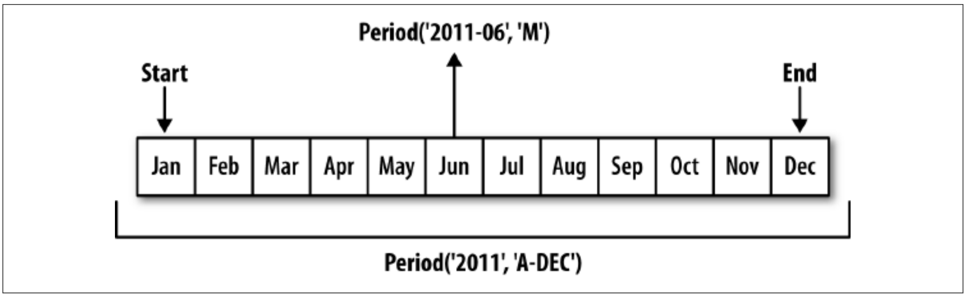

Figure 11-1. Period frequency conversion illustration

When you are converting from high to low frequency, pandas determines the superperiod depending on where the subperiod "belongs." For example, in `A-JUN` frequency, the month `Aug-2007` is actually part of the `2008` period:

```
In [168]: p = pd.Period('Aug-2007', 'M')

In [169]: p.asfreq('A-JUN')
Out[169]: Period('2008', 'A-JUN')
```

Whole `PeriodIndex` objects or time series can be similarly converted with the same semantics:

```
In [170]: rng = pd.period_range('2006', '2009', freq='A-DEC')

In [171]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [172]: ts
Out[172]:
2006    1.607578
2007    0.200381
2008   -0.834068
2009   -0.302988
Freq: A-DEC, dtype: float64

In [173]: ts.asfreq('M', how='start')
Out[173]:
2006-01    1.607578
2007-01    0.200381
2008-01   -0.834068
2009-01   -0.302988
Freq: M, dtype: float64
```

Here, the annual periods are replaced with monthly periods corresponding to the first month falling within each annual period. If we instead wanted the last business day of each year, we can use the `'B'` frequency and indicate that we want the end of the period:

```
In [174]: ts.asfreq('B', how='end')
Out[174]:
2006-12-29    1.607578
2007-12-31    0.200381
2008-12-31   -0.834068
2009-12-31   -0.302988
Freq: B, dtype: float64
```

Quarterly Period Frequencies

Quarterly data is standard in accounting, finance, and other fields. Much quarterly data is reported relative to a *fiscal year end*, typically the last calendar or business day of one of the 12 months of the year. Thus, the period `2012Q4` has a different meaning depending on fiscal year end. pandas supports all 12 possible quarterly frequencies as `Q-JAN` through `Q-DEC`:

```
In [175]: p = pd.Period('2012Q4', freq='Q-JAN')

In [176]: p
Out[176]: Period('2012Q4', 'Q-JAN')
```

In the case of fiscal year ending in January, 2012Q4 runs from November through January, which you can check by converting to daily frequency. See Figure 11-2 for an illustration.

Figure 11-2. Different quarterly frequency conventions

```
In [177]: p.asfreq('D', 'start')
Out[177]: Period('2011-11-01', 'D')

In [178]: p.asfreq('D', 'end')
Out[178]: Period('2012-01-31', 'D')
```

Thus, it's possible to do easy period arithmetic; for example, to get the timestamp at 4 PM on the second-to-last business day of the quarter, you could do:

```
In [179]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')

In [181]: p4pm.to_timestamp()
Out[181]: Timestamp('2012-01-30 16:00:00')
```

You can generate quarterly ranges using `period_range`. Arithmetic is identical, too:

```
In [182]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')

In [183]: ts = pd.Series(np.arange(len(rng)), index=rng)

In [184]: ts
Out[184]:
2011Q3    0
2011Q4    1
2012Q1    2
2012Q2    3
2012Q3    4
2012Q4    5
Freq: Q-JAN, dtype: int64

In [185]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [186]: ts.index = new_rng.to_timestamp()
```

```
In [187]: ts
Out[187]:
2010-10-28 16:00:00    0
2011-01-28 16:00:00    1
2011-04-28 16:00:00    2
2011-07-28 16:00:00    3
2011-10-28 16:00:00    4
2012-01-30 16:00:00    5
dtype: int64
```

Converting Timestamps to Periods (and Back)

Series and DataFrame objects indexed by timestamps can be converted to periods with the `to_period` method:

```
In [188]: rng = pd.date_range('2000-01-01', periods=3, freq='M')

In [189]: ts = pd.Series(np.random.randn(3), index=rng)

In [190]: ts
Out[190]:
2000-01-31    1.663261
2000-02-29   -0.996206
2000-03-31    1.521760
Freq: M, dtype: float64

In [191]: pts = ts.to_period()

In [192]: pts
Out[192]:
2000-01    1.663261
2000-02   -0.996206
2000-03    1.521760
Freq: M, dtype: float64
```

Since periods refer to non-overlapping timespans, a timestamp can only belong to a single period for a given frequency. While the frequency of the new `PeriodIndex` is inferred from the timestamps by default, you can specify any frequency you want. There is also no problem with having duplicate periods in the result:

```
In [193]: rng = pd.date_range('1/29/2000', periods=6, freq='D')

In [194]: ts2 = pd.Series(np.random.randn(6), index=rng)

In [195]: ts2
Out[195]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
```

```
2000-02-03   -0.167933
Freq: D, dtype: float64

In [196]: ts2.to_period('M')
Out[196]:
2000-01    0.244175
2000-01    0.423331
2000-01   -0.654040
2000-02    2.089154
2000-02   -0.060220
2000-02   -0.167933
Freq: M, dtype: float64
```

To convert back to timestamps, use `to_timestamp`:

```
In [197]: pts = ts2.to_period()

In [198]: pts
Out[198]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64

In [199]: pts.to_timestamp(how='end')
Out[199]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64
```

Creating a PeriodIndex from Arrays

Fixed frequency datasets are sometimes stored with timespan information spread across multiple columns. For example, in this macroeconomic dataset, the year and quarter are in different columns:

```
In [200]: data = pd.read_csv('examples/macrodata.csv')

In [201]: data.head(5)
Out[201]:
     year  quarter   realgdp  realcons  realinv  realgovt  realdpi    cpi  \
0  1959.0      1.0  2710.349    1707.4  286.898   470.045   1886.9  28.98
1  1959.0      2.0  2778.801    1733.7  310.859   481.301   1919.7  29.15
2  1959.0      3.0  2775.488    1751.8  289.226   491.260   1916.4  29.35
3  1959.0      4.0  2785.204    1753.7  299.356   484.052   1931.3  29.37
```

```
4  1960.0      1.0  2847.699    1770.5  331.722   462.199   1955.5  29.54
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00      0.00
1  141.7      3.08    5.1  177.830  2.34      0.74
2  140.5      3.82    5.3  178.657  2.74      1.09
3  140.0      4.33    5.6  179.386  0.27      4.06
4  139.6      3.50    5.2  180.007  2.31      1.19

In [202]: data.year
Out[202]:
0      1959.0
1      1959.0
2      1959.0
3      1959.0
4      1960.0
5      1960.0
6      1960.0
7      1960.0
8      1961.0
9      1961.0
        ...
193    2007.0
194    2007.0
195    2007.0
196    2008.0
197    2008.0
198    2008.0
199    2008.0
200    2009.0
201    2009.0
202    2009.0
Name: year, Length: 203, dtype: float64

In [203]: data.quarter
Out[203]:
0      1.0
1      2.0
2      3.0
3      4.0
4      1.0
5      2.0
6      3.0
7      4.0
8      1.0
9      2.0
      ...
193    2.0
194    3.0
195    4.0
196    1.0
197    2.0
198    3.0
```

```
199    4.0
200    1.0
201    2.0
202    3.0
Name: quarter, Length: 203, dtype: float64
```

By passing these arrays to `PeriodIndex` with a frequency, you can combine them to form an index for the DataFrame:

```
In [204]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....:                        freq='Q-DEC')

In [205]: index
Out[205]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203, freq='Q-DEC')

In [206]: data.index = index

In [207]: data.infl
Out[207]:
1959Q1    0.00
1959Q2    2.34
1959Q3    2.74
1959Q4    0.27
1960Q1    2.31
1960Q2    0.14
1960Q3    2.70
1960Q4    1.21
1961Q1   -0.40
1961Q2    1.47
          ...
2007Q2    2.75
2007Q3    3.45
2007Q4    6.38
2008Q1    2.82
2008Q2    8.53
2008Q3   -3.16
2008Q4   -8.79
2009Q1    0.94
2009Q2    3.37
2009Q3    3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

11.6 Resampling and Frequency Conversion

Resampling refers to the process of converting a time series from one frequency to another. Aggregating higher frequency data to lower frequency is called *downsampling*, while converting lower frequency to higher frequency is called *upsampling*. Not all resampling falls into either of these categories; for example, converting `W-WED` (weekly on Wednesday) to `W-FRI` is neither upsampling nor downsampling.

pandas objects are equipped with a `resample` method, which is the workhorse function for all frequency conversion. `resample` has a similar API to `groupby`; you call `resample` to group the data, then call an aggregation function:

```
In [208]: rng = pd.date_range('2000-01-01', periods=100, freq='D')

In [209]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [210]: ts
Out[210]:
2000-01-01    0.631634
2000-01-02   -1.594313
2000-01-03   -1.519937
2000-01-04    1.108752
2000-01-05    1.255853
2000-01-06   -0.024330
2000-01-07   -2.047939
2000-01-08   -0.272657
2000-01-09   -1.692615
2000-01-10    1.423830
                ...
2000-03-31   -0.007852
2000-04-01   -1.638806
2000-04-02    1.401227
2000-04-03    1.758539
2000-04-04    0.628932
2000-04-05   -0.423776
2000-04-06    0.789740
2000-04-07    0.937568
2000-04-08   -2.253294
2000-04-09   -1.772919
Freq: D, Length: 100, dtype: float64

In [211]: ts.resample('M').mean()
Out[211]:
2000-01-31   -0.165893
2000-02-29    0.078606
2000-03-31    0.223811
2000-04-30   -0.063643
Freq: M, dtype: float64

In [212]: ts.resample('M', kind='period').mean()
Out[212]:
```

```
2000-01   -0.165893
2000-02    0.078606
2000-03    0.223811
2000-04   -0.063643
Freq: M, dtype: float64
```

resample is a flexible and high-performance method that can be used to process very large time series. The examples in the following sections illustrate its semantics and use. Table 11-5 summarizes some of its options.

Table 11-5. Resample method arguments

Argument	Description
freq	String or DateOffset indicating desired resampled frequency (e.g., 'M', '5min', or Second(15))
axis	Axis to resample on; default axis=0
fill_method	How to interpolate when upsampling, as in 'ffill' or 'bfill'; by default does no interpolation
closed	In downsampling, which end of each interval is closed (inclusive), 'right' or 'left'
label	In downsampling, how to label the aggregated result, with the 'right' or 'left' bin edge (e.g., the 9:30 to 9:35 five-minute interval could be labeled 9:30 or 9:35)
loffset	Time adjustment to the bin labels, such as '-1s' / Second(-1) to shift the aggregate labels one second earlier
limit	When forward or backward filling, the maximum number of periods to fill
kind	Aggregate to periods ('period') or timestamps ('timestamp'); defaults to the type of index the time series has
convention	When resampling periods, the convention ('start' or 'end') for converting the low-frequency period to high frequency; defaults to 'end'

Downsampling

Aggregating data to a regular, lower frequency is a pretty normal time series task. The data you're aggregating doesn't need to be fixed frequently; the desired frequency defines *bin edges* that are used to slice the time series into pieces to aggregate. For example, to convert to monthly, 'M' or 'BM', you need to chop up the data into one-month intervals. Each interval is said to be *half-open*; a data point can only belong to one interval, and the union of the intervals must make up the whole time frame. There are a couple things to think about when using resample to downsample data:

- Which side of each interval is *closed*
- How to label each aggregated bin, either with the start of the interval or the end

To illustrate, let's look at some one-minute data:

```
In [213]: rng = pd.date_range('2000-01-01', periods=12, freq='T')

In [214]: ts = pd.Series(np.arange(12), index=rng)
```

```
In [215]: ts
Out[215]:
2000-01-01 00:00:00     0
2000-01-01 00:01:00     1
2000-01-01 00:02:00     2
2000-01-01 00:03:00     3
2000-01-01 00:04:00     4
2000-01-01 00:05:00     5
2000-01-01 00:06:00     6
2000-01-01 00:07:00     7
2000-01-01 00:08:00     8
2000-01-01 00:09:00     9
2000-01-01 00:10:00    10
2000-01-01 00:11:00    11
Freq: T, dtype: int64
```

Suppose you wanted to aggregate this data into five-minute chunks or *bars* by taking the sum of each group:

```
In [216]: ts.resample('5min', closed='right').sum()
Out[216]:
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int64
```

The frequency you pass defines bin edges in five-minute increments. By default, the *left* bin edge is inclusive, so the `00:00` value is included in the `00:00` to `00:05` interval.[1] Passing `closed='right'` changes the interval to be closed on the right:

```
In [217]: ts.resample('5min', closed='right').sum()
Out[217]:
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int64
```

The resulting time series is labeled by the timestamps from the left side of each bin. By passing `label='right'` you can label them with the right bin edge:

```
In [218]: ts.resample('5min', closed='right', label='right').sum()
Out[218]:
2000-01-01 00:00:00     0
2000-01-01 00:05:00    15
```

1 The choice of the default values for `closed` and `label` might seem a bit odd to some users. In practice the choice is somewhat arbitrary; for some target frequencies, `closed='left'` is preferable, while for others `closed='right'` makes more sense. The important thing is that you keep in mind exactly how you are segmenting the data.

```
2000-01-01 00:10:00    40
2000-01-01 00:15:00    11
Freq: 5T, dtype: int64
```

See Figure 11-3 for an illustration of minute frequency data being resampled to five-minute frequency.

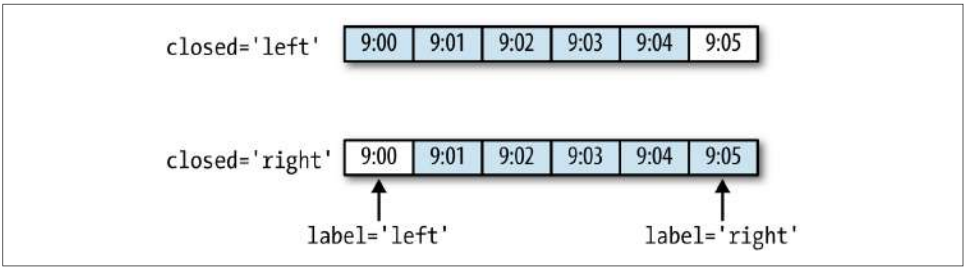

Figure 11-3. Five-minute resampling illustration of closed, label conventions

Lastly, you might want to shift the result index by some amount, say subtracting one second from the right edge to make it more clear which interval the timestamp refers to. To do this, pass a string or date offset to `loffset`:

```
In [219]: ts.resample('5min', closed='right',
   .....:             label='right', loffset='-1s').sum()
Out[219]:
1999-12-31 23:59:59     0
2000-01-01 00:04:59    15
2000-01-01 00:09:59    40
2000-01-01 00:14:59    11
Freq: 5T, dtype: int64
```

You also could have accomplished the effect of `loffset` by calling the `shift` method on the result without the `loffset`.

Open-High-Low-Close (OHLC) resampling

In finance, a popular way to aggregate a time series is to compute four values for each bucket: the first (open), last (close), maximum (high), and minimal (low) values. By using the `ohlc` aggregate function you will obtain a DataFrame having columns containing these four aggregates, which are efficiently computed in a single sweep of the data:

```
In [220]: ts.resample('5min').ohlc()
Out[220]:
                     open  high  low  close
2000-01-01 00:00:00     0     4    0      4
2000-01-01 00:05:00     5     9    5      9
2000-01-01 00:10:00    10    11   10     11
```

Upsampling and Interpolation

When converting from a low frequency to a higher frequency, no aggregation is needed. Let's consider a DataFrame with some weekly data:

```
In [221]: frame = pd.DataFrame(np.random.randn(2, 4),
   .....:                      index=pd.date_range('1/1/2000', periods=2,
   .....:                                          freq='W-WED'),
   .....:                      columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [222]: frame
Out[222]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

When you are using an aggregation function with this data, there is only one value per group, and missing values result in the gaps. We use the `asfreq` method to convert to the higher frequency without any aggregation:

```
In [223]: df_daily = frame.resample('D').asfreq()

In [224]: df_daily
Out[224]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09       NaN       NaN       NaN       NaN
2000-01-10       NaN       NaN       NaN       NaN
2000-01-11       NaN       NaN       NaN       NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

Suppose you wanted to fill forward each weekly value on the non-Wednesdays. The same filling or interpolation methods available in the `fillna` and `reindex` methods are available for resampling:

```
In [225]: frame.resample('D').ffill()
Out[225]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
2000-01-08 -0.896431  0.677263  0.036503  0.087102
2000-01-09 -0.896431  0.677263  0.036503  0.087102
2000-01-10 -0.896431  0.677263  0.036503  0.087102
2000-01-11 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

You can similarly choose to only fill a certain number of periods forward to limit how far to continue using an observed value:

```
In [226]: frame.resample('D').ffill(limit=2)
Out[226]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09       NaN       NaN       NaN       NaN
2000-01-10       NaN       NaN       NaN       NaN
2000-01-11       NaN       NaN       NaN       NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

Notably, the new date index need not overlap with the old one at all:

```
In [227]: frame.resample('W-THU').ffill()
Out[227]:
            Colorado     Texas  New York      Ohio
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-13 -0.046662  0.927238  0.482284 -0.867130
```

Resampling with Periods

Resampling data indexed by periods is similar to timestamps:

```
In [228]: frame = pd.DataFrame(np.random.randn(24, 4),
   .....:                      index=pd.period_range('1-2000', '12-2001',
   .....:                                            freq='M'),
   .....:                      columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [229]: frame[:5]
Out[229]:
         Colorado     Texas  New York      Ohio
2000-01  0.493841 -0.155434  1.397286  1.507055
2000-02 -1.179442  0.443171  1.395676 -0.529658
2000-03  0.787358  0.248845  0.743239  1.267746
2000-04  1.302395 -0.272154 -0.051532 -0.467740
2000-05 -1.040816  0.426419  0.312945 -1.115689

In [230]: annual_frame = frame.resample('A-DEC').mean()

In [231]: annual_frame
Out[231]:
      Colorado     Texas  New York      Ohio
2000  0.556703  0.016631  0.111873 -0.027445
2001  0.046303  0.163344  0.251503 -0.157276
```

Upsampling is more nuanced, as you must make a decision about which end of the timespan in the new frequency to place the values before resampling, just like the `asfreq` method. The `convention` argument defaults to `'start'` but can also be `'end'`:

```
# Q-DEC: Quarterly, year ending in December
In [232]: annual_frame.resample('Q-DEC').ffill()
Out[232]:
```

```
         Colorado     Texas  New York      Ohio
2000Q1  0.556703  0.016631  0.111873 -0.027445
2000Q2  0.556703  0.016631  0.111873 -0.027445
2000Q3  0.556703  0.016631  0.111873 -0.027445
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.046303  0.163344  0.251503 -0.157276
2001Q2  0.046303  0.163344  0.251503 -0.157276
2001Q3  0.046303  0.163344  0.251503 -0.157276
2001Q4  0.046303  0.163344  0.251503 -0.157276

In [233]: annual_frame.resample('Q-DEC', convention='end').ffill()
Out[233]:
         Colorado     Texas  New York      Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
```

Since periods refer to timespans, the rules about upsampling and downsampling are more rigid:

- In downsampling, the target frequency must be a *subperiod* of the source frequency.
- In upsampling, the target frequency must be a *superperiod* of the source frequency.

If these rules are not satisfied, an exception will be raised. This mainly affects the quarterly, annual, and weekly frequencies; for example, the timespans defined by `Q-MAR` only line up with `A-MAR`, `A-JUN`, `A-SEP`, and `A-DEC`:

```
In [234]: annual_frame.resample('Q-MAR').ffill()
Out[234]:
         Colorado     Texas  New York      Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
2002Q1  0.046303  0.163344  0.251503 -0.157276
2002Q2  0.046303  0.163344  0.251503 -0.157276
2002Q3  0.046303  0.163344  0.251503 -0.157276
```

11.7 Moving Window Functions

An important class of array transformations used for time series operations are statistics and other functions evaluated over a sliding window or with exponentially decaying weights. This can be useful for smoothing noisy or gappy data. I call these *moving window functions*, even though it includes functions without a fixed-length window like exponentially weighted moving average. Like other statistical functions, these also automatically exclude missing data.

Before digging in, we can load up some time series data and resample it to business day frequency:

```
In [235]: close_px_all = pd.read_csv('examples/stock_px_2.csv',
   .....:                            parse_dates=True, index_col=0)

In [236]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]

In [237]: close_px = close_px.resample('B').ffill()
```

I now introduce the `rolling` operator, which behaves similarly to `resample` and `groupby`. It can be called on a Series or DataFrame along with a `window` (expressed as a number of periods; see Figure 11-4 for the plot created):

```
In [238]: close_px.AAPL.plot()
Out[238]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f2570cf98>

In [239]: close_px.AAPL.rolling(250).mean().plot()
```

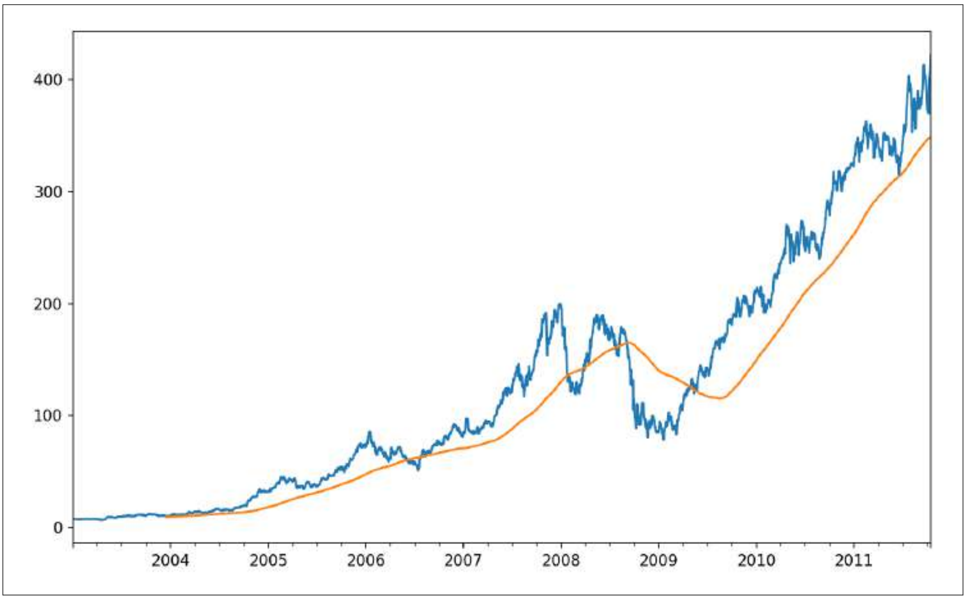

Figure 11-4. Apple Price with 250-day MA

The expression `rolling(250)` is similar in behavior to `groupby`, but instead of grouping it creates an object that enables grouping over a 250-day sliding window. So here we have the 250-day moving window average of Apple's stock price.

By default rolling functions require all of the values in the window to be non-NA. This behavior can be changed to account for missing data and, in particular, the fact that you will have fewer than `window` periods of data at the beginning of the time series (see Figure 11-5):

```
In [241]: appl_std250 = close_px.AAPL.rolling(250, min_periods=10).std()

In [242]: appl_std250[5:12]
Out[242]:
2003-01-09         NaN
2003-01-10         NaN
2003-01-13         NaN
2003-01-14         NaN
2003-01-15    0.077496
2003-01-16    0.074760
2003-01-17    0.112368
Freq: B, Name: AAPL, dtype: float64

In [243]: appl_std250.plot()
```

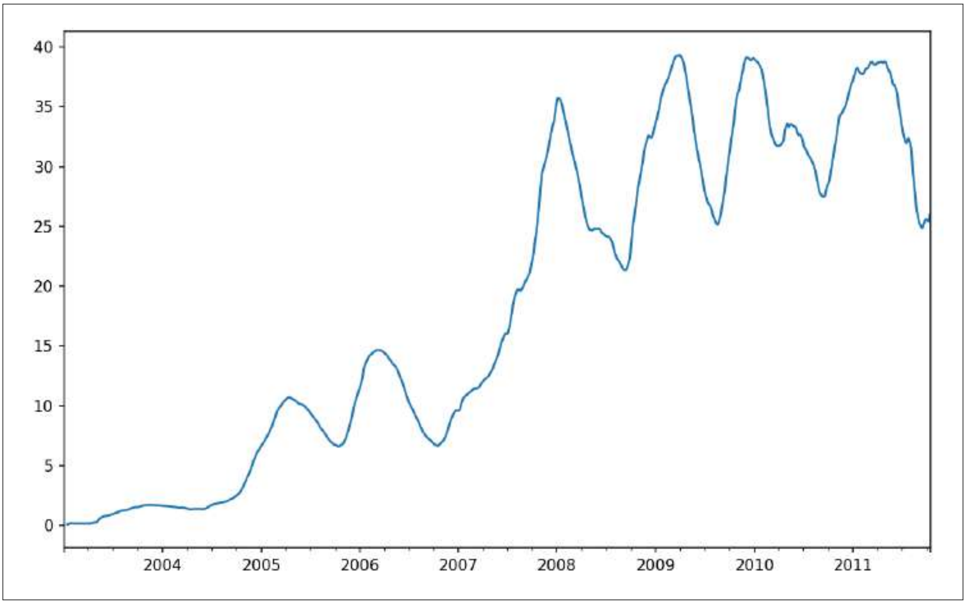

Figure 11-5. Apple 250-day daily return standard deviation

In order to compute an *expanding window mean*, use the `expanding` operator instead of `rolling`. The expanding mean starts the time window from the beginning of the time series and increases the size of the window until it encompasses the whole series. An expanding window mean on the `apple_std250` time series looks like this:

```
In [244]: expanding_mean = appl_std250.expanding().mean()
```

Calling a moving window function on a DataFrame applies the transformation to each column (see Figure 11-6):

```
In [246]: close_px.rolling(60).mean().plot(logy=True)
```

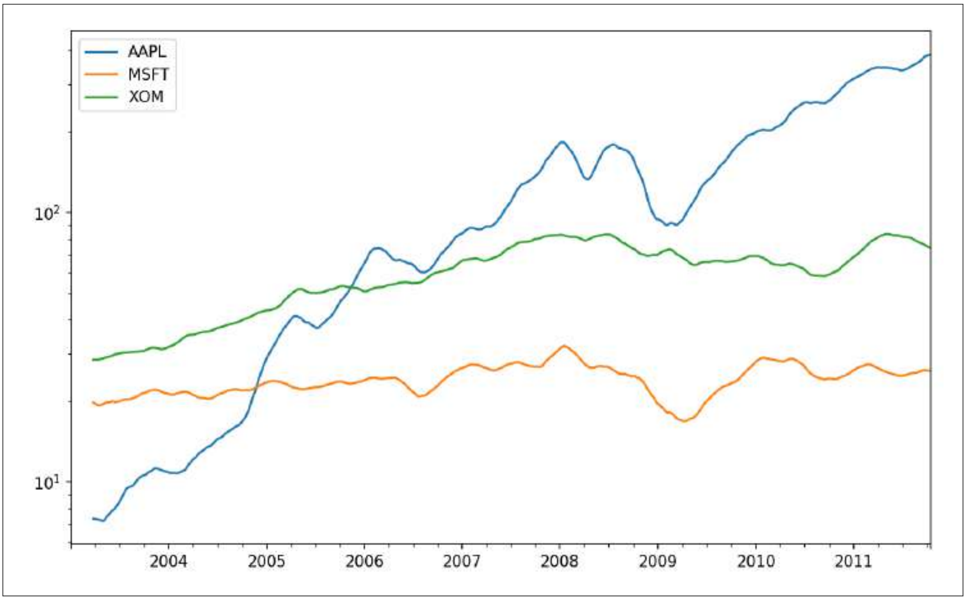

Figure 11-6. Stocks prices 60-day MA (log Y-axis)

The rolling function also accepts a string indicating a fixed-size time offset rather than a set number of periods. Using this notation can be useful for irregular time series. These are the same strings that you can pass to resample. For example, we could compute a 20-day rolling mean like so:

```
In [247]: close_px.rolling('20D').mean()
Out[247]:
                  AAPL       MSFT        XOM
2003-01-02    7.400000  21.110000  29.220000
2003-01-03    7.425000  21.125000  29.230000
2003-01-06    7.433333  21.256667  29.473333
2003-01-07    7.432500  21.425000  29.342500
2003-01-08    7.402000  21.402000  29.240000
2003-01-09    7.391667  21.490000  29.273333
2003-01-10    7.387143  21.558571  29.238571
2003-01-13    7.378750  21.633750  29.197500
2003-01-14    7.370000  21.717778  29.194444
2003-01-15    7.355000  21.757000  29.152000
...                ...        ...        ...
2011-10-03  398.002143  25.890714  72.413571
2011-10-04  396.802143  25.807857  72.427143
2011-10-05  395.751429  25.729286  72.422857
```

```
2011-10-06  394.099286  25.673571  72.375714
2011-10-07  392.479333  25.712000  72.454667
2011-10-10  389.351429  25.602143  72.527857
2011-10-11  388.505000  25.674286  72.835000
2011-10-12  388.531429  25.810000  73.400714
2011-10-13  388.826429  25.961429  73.905000
2011-10-14  391.038000  26.048667  74.185333
[2292 rows x 3 columns]
```

Exponentially Weighted Functions

An alternative to using a static window size with equally weighted observations is to specify a constant *decay factor* to give more weight to more recent observations. There are a couple of ways to specify the decay factor. A popular one is using a *span*, which makes the result comparable to a simple moving window function with window size equal to the span.

Since an exponentially weighted statistic places more weight on more recent observations, it "adapts" faster to changes compared with the equal-weighted version.

pandas has the `ewm` operator to go along with `rolling` and `expanding`. Here's an example comparing a 60-day moving average of Apple's stock price with an EW moving average with `span=60` (see Figure 11-7):

```
In [249]: aapl_px = close_px.AAPL['2006':'2007']

In [250]: ma60 = aapl_px.rolling(30, min_periods=20).mean()

In [251]: ewma60 = aapl_px.ewm(span=30).mean()

In [252]: ma60.plot(style='k--', label='Simple MA')
Out[252]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [253]: ewma60.plot(style='k-', label='EW MA')
Out[253]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [254]: plt.legend()
```

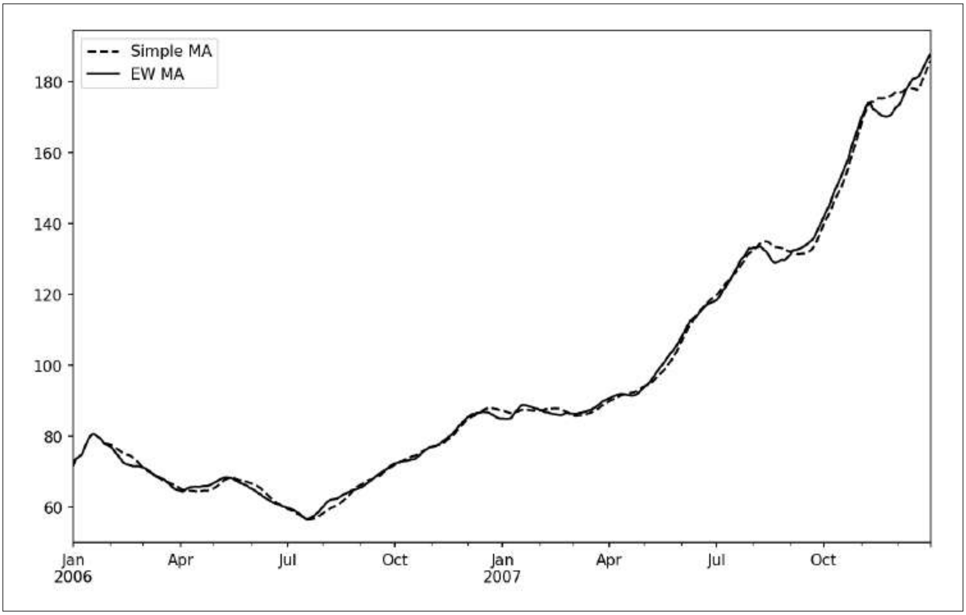

Figure 11-7. Simple moving average versus exponentially weighted

Binary Moving Window Functions

Some statistical operators, like correlation and covariance, need to operate on two time series. As an example, financial analysts are often interested in a stock's correlation to a benchmark index like the S&P 500. To have a look at this, we first compute the percent change for all of our time series of interest:

```
In [256]: spx_px = close_px_all['SPX']

In [257]: spx_rets = spx_px.pct_change()

In [258]: returns = close_px.pct_change()
```

The `corr` aggregation function after we call `rolling` can then compute the rolling correlation with `spx_rets` (see Figure 11-8 for the resulting plot):

```
In [259]: corr = returns.AAPL.rolling(125, min_periods=100).corr(spx_rets)

In [260]: corr.plot()
```

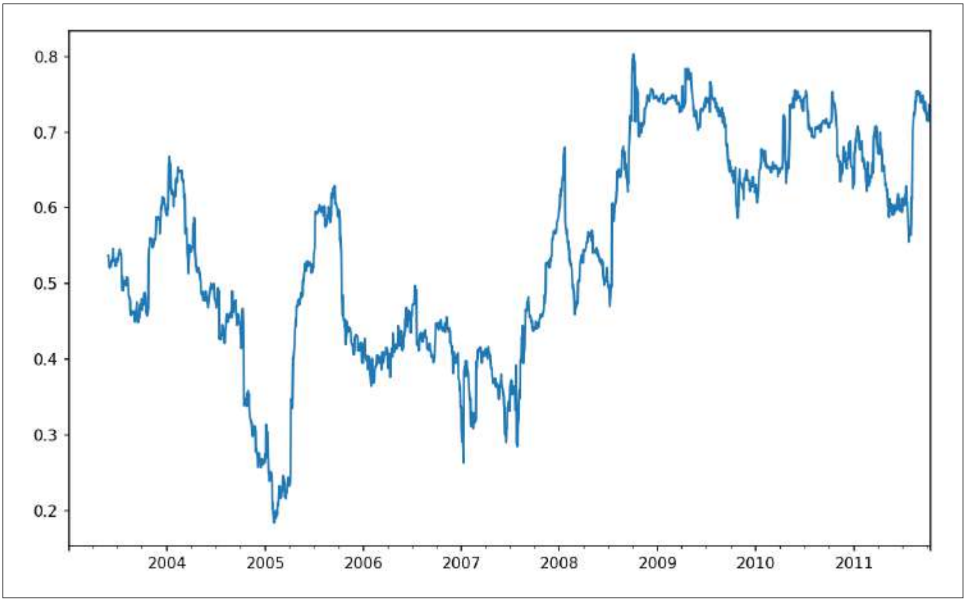

Figure 11-8. Six-month AAPL return correlation to S&P 500

Suppose you wanted to compute the correlation of the S&P 500 index with many stocks at once. Writing a loop and creating a new DataFrame would be easy but might get repetitive, so if you pass a Series and a DataFrame, a function like `rolling_corr` will compute the correlation of the Series (`spx_rets`, in this case) with each column in the DataFrame (see Figure 11-9 for the plot of the result):

```
In [262]: corr = returns.rolling(125, min_periods=100).corr(spx_rets)

In [263]: corr.plot()
```

Figure 11-9. Six-month return correlations to S&P 500

User-Defined Moving Window Functions

The `apply` method on `rolling` and related methods provides a means to apply an array function of your own devising over a moving window. The only requirement is that the function produce a single value (a reduction) from each piece of the array. For example, while we can compute sample quantiles using `rolling(...).quantile(q)`, we might be interested in the percentile rank of a particular value over the sample. The `scipy.stats.percentileofscore` function does just this (see Figure 11-10 for the resulting plot):

```
In [265]: from scipy.stats import percentileofscore

In [266]: score_at_2percent = lambda x: percentileofscore(x, 0.02)

In [267]: result = returns.AAPL.rolling(250).apply(score_at_2percent)

In [268]: result.plot()
```

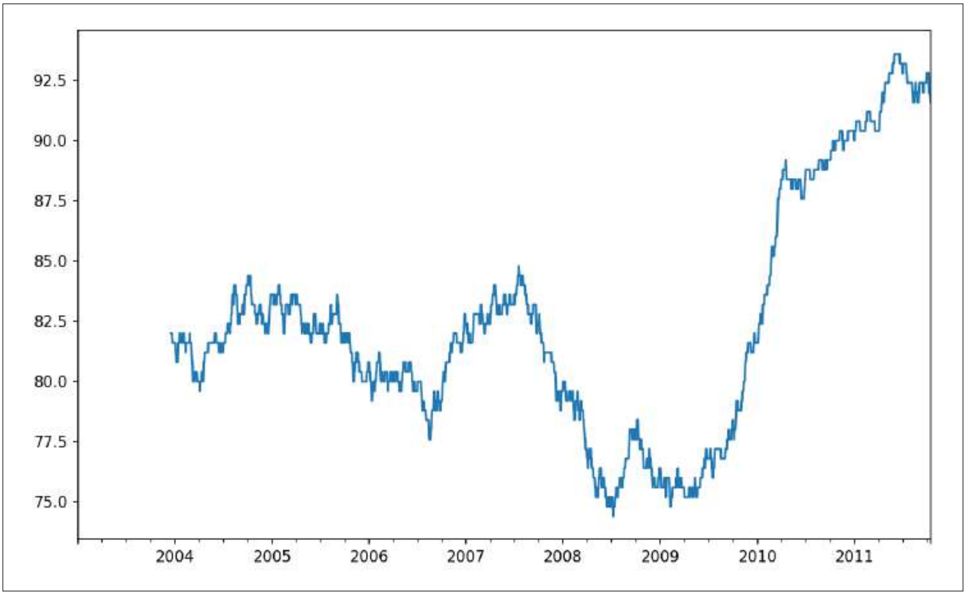

Figure 11-10. Percentile rank of 2% AAPL return over one-year window

If you don't have SciPy installed already, you can install it with conda or pip.

11.8 Conclusion

Time series data calls for different types of analysis and data transformation tools than the other types of data we have explored in previous chapters.

In the following chapters, we will move on to some advanced pandas methods and show how to start using modeling libraries like statsmodels and scikit-learn.