explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a `groupby`.

Planets Data

Here we will use the Planets dataset, available via the Seaborn package (see "Visualization with Seaborn" on page 311). It gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short). It can be downloaded with a simple Seaborn command:

```
In[2]: import seaborn as sns
       planets = sns.load_dataset('planets')
       planets.shape
```

```
Out[2]: (1035, 6)
```

```
In[3]: planets.head()
```

```
Out[3]:    method           number  orbital_period  mass   distance  year
        0  Radial Velocity  1       269.300         7.10   77.40     2006
        1  Radial Velocity  1       874.774         2.21   56.95     2008
        2  Radial Velocity  1       763.000         2.60   19.84     2011
        3  Radial Velocity  1       326.030         19.40  110.62    2007
        4  Radial Velocity  1       516.220         10.50  119.47    2009
```

This has some details on the 1,000+ exoplanets discovered up to 2014.

Simple Aggregation in Pandas

Earlier we explored some of the data aggregations available for NumPy arrays ("Aggregations: Min, Max, and Everything in Between" on page 58). As with a one-dimensional NumPy array, for a Pandas `Series` the aggregates return a single value:

```
In[4]: rng = np.random.RandomState(42)
       ser = pd.Series(rng.rand(5))
       ser
```

```
Out[4]: 0    0.374540
        1    0.950714
        2    0.731994
        3    0.598658
        4    0.156019
        dtype: float64
```

```
In[5]: ser.sum()
```

```
Out[5]: 2.8119254917081569
```

```
In[6]: ser.mean()
```

```
Out[6]: 0.56238509834163142
```

For a `DataFrame`, by default the aggregates return results within each column:

```
In[7]: df = pd.DataFrame({'A': rng.rand(5),
                          'B': rng.rand(5)})
       df
```

```
Out[7]:           A         B
        0  0.155995  0.020584
        1  0.058084  0.969910
        2  0.866176  0.832443
        3  0.601115  0.212339
        4  0.708073  0.181825
```

```
In[8]: df.mean()
```

```
Out[8]: A    0.477888
        B    0.443420
        dtype: float64
```

By specifying the `axis` argument, you can instead aggregate within each row:

```
In[9]: df.mean(axis='columns')
```

```
Out[9]: 0    0.088290
        1    0.513997
        2    0.849309
        3    0.406727
        4    0.444949
        dtype: float64
```

Pandas `Series` and `DataFrame`s include all of the common aggregates mentioned in "Aggregations: Min, Max, and Everything in Between" on page 58; in addition, there is a convenience method `describe()` that computes several common aggregates for each column and returns the result. Let's use this on the Planets data, for now dropping rows with missing values:

```
In[10]: planets.dropna().describe()
```

```
Out[10]:          number  orbital_period        mass    distance         year
         count  498.00000      498.000000  498.000000  498.000000   498.000000
         mean     1.73494      835.778671    2.509320   52.068213  2007.377510
         std      1.17572     1469.128259    3.636274   46.596041     4.167284
         min      1.00000        1.328300    0.003600    1.350000  1989.000000
         25%      1.00000       38.272250    0.212500   24.497500  2005.000000
         50%      1.00000      357.000000    1.245000   39.940000  2009.000000
         75%      2.00000      999.600000    2.867500   59.332500  2011.000000
         max      6.00000    17337.500000   25.000000  354.000000  2014.000000
```

This can be a useful way to begin understanding the overall properties of a dataset. For example, we see in the `year` column that although exoplanets were discovered as far back as 1989, half of all known exoplanets were not discovered until 2010 or after. This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

Table 3-3 summarizes some other built-in Pandas aggregations.

Table 3-3. Listing of Pandas aggregation methods

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

These are all methods of `DataFrame` and `Series` objects.

To go deeper into the data, however, simple aggregates are often not enough. The next level of data summarization is the `groupby` operation, which allows you to quickly and efficiently compute aggregates on subsets of data.

GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called `groupby` operation. The name "group by" comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *split, apply, combine*.

Split, apply, combine

A canonical example of this split-apply-combine operation, where the "apply" is a summation aggregation, is illustrated in Figure 3-1.

Figure 3-1 makes clear what the `GroupBy` accomplishes:

- The *split* step involves breaking up and grouping a `DataFrame` depending on the value of the specified key.
- The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
- The *combine* step merges the results of these operations into an output array.