CHAPTER 3

Manipulating Tabular Data Using Pandas

What Is Pandas?

While NumPy arrays are a much-improved N-dimensional array object version over Python's list, it is insufficient to meet the needs of data science. In the real world, data are often presented in table formats. For example, consider the content of the CSV file shown here:

```
,DateTime,mmol/L
0,2016-06-01 08:00:00,6.1
1,2016-06-01 12:00:00,6.5
2,2016-06-01 18:00:00,6.7
3,2016-06-02 08:00:00,5.0
4,2016-06-02 12:00:00,4.9
5,2016-06-02 18:00:00,5.5
6,2016-06-03 08:00:00,5.6
7,2016-06-03 12:00:00,7.1
8,2016-06-03 18:00:00,5.9
9,2016-06-04 09:00:00,6.6
10,2016-06-04 11:00:00,4.1
11,2016-06-04 17:00:00,5.9
12,2016-06-05 08:00:00,7.6
13,2016-06-05 12:00:00,5.1
14,2016-06-05 18:00:00,6.9
15,2016-06-06 08:00:00,5.0
```

```
16,2016-06-06 12:00:00,6.1
17,2016-06-06 18:00:00,4.9
18,2016-06-07 08:00:00,6.6
19,2016-06-07 12:00:00,4.1
20,2016-06-07 18:00:00,6.9
21,2016-06-08 08:00:00,5.6
22,2016-06-08 12:00:00,8.1
23,2016-06-08 18:00:00,10.9
24,2016-06-09 08:00:00,5.2
25,2016-06-09 12:00:00,7.1
26,2016-06-09 18:00:00,4.9
```

The CSV file contains rows of data that are divided into three columns—index, date and time of recording, and blood glucose readings in mmol/L. To be able to deal with data stored as tables, you need a new data type that is more suited to deal with it—*Pandas*. While Python supports lists and dictionaries for manipulating structured data, it is not well suited for manipulating numerical tables, such as the one stored in the CSV file. *Pandas* is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive.

NOTE **Pandas stands for *Panel Data Analysis*.**

Pandas supports two key data structures: Series and DataFrame. In this chapter, you will learn how to work with Series and DataFrames in Pandas.

Pandas Series

A *Pandas Series* is a one-dimensional NumPy-like array, with each element having an index (0, 1, 2, . . . by default); a Series behaves very much like a dictionary that includes an index. Figure 3.1 shows the structure of a Series in Pandas.

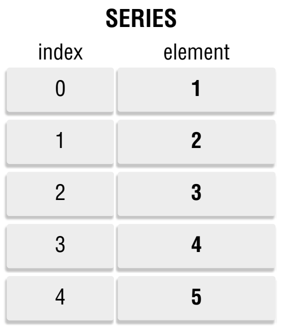

Figure 3.1: A Pandas Series

To create a Series, you first need to import the `pandas` library (the convention is to use `pd` as the alias) and then use the `Series` class:

```
import pandas as pd
series = pd.Series([1,2,3,4,5])
print(series)
```

The preceding code snippet will print the following output:

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

By default, the index of a Series starts from 0.

Creating a Series Using a Specified Index

You can specify an optional index for a Series using the `index` parameter:

```
series = pd.Series([1,2,3,4,5], index=['a','b','c','d','c'])
print(series)
```

The preceding code snippet prints out the following:

```
a    1
b    2
c    3
d    4
c    5
dtype: int64
```

It is worth noting that the index of a Series need not be unique, as the preceding output shows.

Accessing Elements in a Series

Accessing an element in a Series is similar to accessing an element in an array. You can use the position of the element as follows:

```
print(series[2])          # 3
# same as
print(series.iloc[2])     # 3  - based on the position of the index
```

The `iloc` indexer allows you to specify an element via its position.

Alternatively, you can also specify the value of the index of the element you wish to access like this:

```
print(series['d'])        # 4
# same as
print(series.loc['d'])    # 4 - based on the label in the index
```

The `loc` indexer allows you to specify the *label* (value) of an index.

Note that in the preceding two examples, the result is an integer (which is the type of this Series). What happens if we do the following?

```
print(series['c'])        # more than 1 row has the index 'c'
```

In this case, the result would be another Series:

```
c    3
c    5
dtype: int64
```

You can also perform slicing on a Series:

```
print(series[2:])         # returns a Series
print(series.iloc[2:])    # returns a Series
```

The preceding code snippet generates the following output:

```
c    3
d    4
c    5
dtype: int64
```

Specifying a Datetime Range as the Index of a Series

Often, you want to create a timeseries, such as a running sequence of dates in a month. You could use the `date _ range()` function for this purpose:

```
dates1 = pd.date_range('20190525', periods=12)
print(dates1)
```

The preceding code snippet will display the following:

```
DatetimeIndex(['2019-05-25', '2019-05-26', '2019-05-27', '2019-05-28',
               '2019-05-29', '2019-05-30', '2019-05-31', '2019-06-01',
               '2019-06-02', '2019-06-03', '2019-06-04', '2019-06-05'],
              dtype='datetime64[ns]', freq='D')
```

To assign the range of dates as the index of a Series, use the `index` property of the Series like this:

```
series = pd.Series([1,2,3,4,5,6,7,8,9,10,11,12])
series.index = dates1
print(series)
```

You should see the following output:

```
2019-05-25     1
2019-05-26     2
2019-05-27     3
2019-05-28     4
2019-05-29     5
2019-05-30     6
2019-05-31     7
2019-06-01     8
2019-06-02     9
2019-06-03    10
2019-06-04    11
2019-06-05    12
Freq: D, dtype: int64
```

Date Ranges

In the previous section, you saw how to create date ranges using the `date _ range()` function. The `periods` parameter specifies how many dates you want to create, and the default frequency is `D` (for Daily). If you want to change the frequency to month, use the `freq` parameter and set it to `M`:

```
dates2 = pd.date_range('2019-05-01', periods=12, freq='M')
print(dates2)
```

This will print out the following dates:

```
DatetimeIndex(['2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
               '2019-09-30', '2019-10-31', '2019-11-30', '2019-12-31',
               '2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30'],
              dtype='datetime64[ns]', freq='M')
```

Notice that when the frequency is set to month, the day of each date will be the last day of the month. If you want the date to start with the first day of the month, set the `freq` parameter to `MS`:

```
dates2 = pd.date_range('2019-05-01', periods=12, freq='MS')
print(dates2)
```

You should now see that each of the dates starts with the first day of every month:

```
DatetimeIndex(['2019-05-01', '2019-06-01', '2019-07-01', '2019-08-01',
               '2019-09-01', '2019-10-01', '2019-11-01', '2019-12-01',
               '2020-01-01', '2020-02-01', '2020-03-01', '2020-04-01'],
              dtype='datetime64[ns]', freq='MS')
```

TIP For other date frequencies, check out the *Offset Aliases* section of the documentation at:

`http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases`

Notice that Pandas automatically interprets the date you specified. In this case, *2019-05-01* is interpreted as 1st May, 2019. In some regions, developers will specify the date in the *dd-mm-yyyy* format. Thus to represent 5th January, 2019, you would specify it as follows:

```
dates2 = pd.date_range('05-01-2019', periods=12, freq='MS')
print(dates2)
```

Note however that in this case, Pandas will interpret 05 as the month, 01 as the day, and 2019 as the year, as the following output proves:

```
DatetimeIndex(['2019-05-01', '2019-06-01', '2019-07-01', '2019-08-01',
               '2019-09-01', '2019-10-01', '2019-11-01', '2019-12-01',
               '2020-01-01', '2020-02-01', '2020-03-01', '2020-04-01'],
              dtype='datetime64[ns]', freq='MS')
```

In addition to setting dates, you can also set the time:

```
dates3 = pd.date_range('2019/05/17 09:00:00', periods=8, freq='H')
print(dates3)
```

You should see the following output:

```
DatetimeIndex(['2019-05-17 09:00:00', '2019-05-17 10:00:00',
               '2019-05-17 11:00:00', '2019-05-17 12:00:00',
               '2019-05-17 13:00:00', '2019-05-17 14:00:00',
               '2019-05-17 15:00:00', '2019-05-17 16:00:00'],
              dtype='datetime64[ns]', freq='H')
```

TIP If you review each of the code snippets that you have seen in this section, you will see that Pandas allows you to specify the date in different formats, such as *mm-dd-yyyy*, *yyyy-mm-dd*, and *yyyy/mm/dd*, and it will automatically try to make sense of the dates specified. When in doubt, it is always useful to print out the range of dates to confirm.

Pandas DataFrame

A *Pandas DataFrame* is a two-dimensional NumPy-like array. You can think of it as a table. Figure 3.2 shows the structure of a DataFrame in Pandas. It also shows you that an individual column in a DataFrame (together with the index) is a Series.

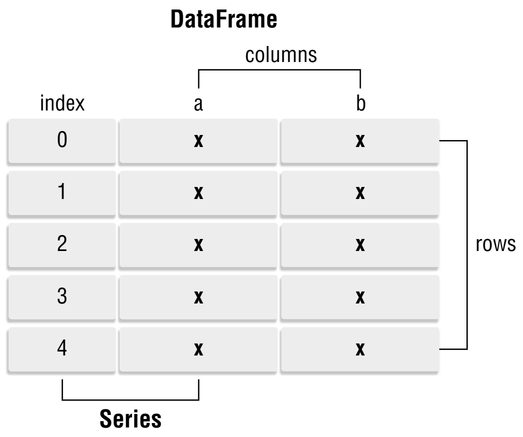

Figure 3.2: A Pandas DataFrame

A DataFrame is very useful in the world of data science and machine learning, as it closely mirrors how data are stored in real-life. Imagine the data stored in a spreadsheet, and you would have a very good visual impression of a DataFrame. A Pandas DataFrame is often used when representing data in machine learning. Hence, for the remaining sections in this chapter, we are going to invest significant time and effort in understanding how it works.

Creating a DataFrame

You can create a Pandas DataFrame using the `DataFrame()` class:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10,4),
                  columns=list('ABCD'))
print(df)
```

In the preceding code snippet, a DataFrame of 10 rows and 4 columns was created, and each cell is filled with a random number using the `randn()` function. Each column has a label: "A", "B", "C", and "D":

```
          A         B         C         D
0  0.187497  1.122150 -0.988277 -1.985934
```

```
1  0.360803 -0.562243 -0.340693 -0.986988
2 -0.040627  0.067333 -0.452978  0.686223
3 -0.279572 -0.702492  0.252265  0.958977
4  0.537438 -1.737568  0.714727 -0.939288
5  0.070011 -0.516443 -1.655689  0.246721
6  0.001268  0.951517  2.107360 -0.108726
7 -0.185258  0.856520 -0.686285  1.104195
8  0.387023  1.706336 -2.452653  0.260466
9 -1.054974  0.556775 -0.945219 -0.030295
```

NOTE Obviously, you will see a different set of numbers in your own DataFrame, as the numbers are generated randomly.

More often than not, a DataFrame is usually loaded from a text file, such as a CSV file. Suppose that you have a CSV file named `data.csv` with the following content:

```
A,B,C,D
0.187497,1.122150,-0.988277,-1.985934
0.360803,-0.562243,-0.340693,-0.986988
-0.040627,0.067333,-0.452978,0.686223
-0.279572,-0.702492,0.252265,0.958977
0.537438,-1.737568,0.714727,-0.939288
0.070011,-0.516443,-1.655689,0.246721
0.001268,0.951517,2.107360,-0.108726
-0.185258,0.856520,-0.686285,1.104195
0.387023,1.706336,-2.452653,0.260466
-1.054974,0.556775,-0.945219,-0.030295
```

You can load the content of the CSV file into a DataFrame using the `read_csv()` function:

```
df = pd.read_csv('data.csv')
```

Specifying the Index in a DataFrame

Notice that the DataFrame printed in the previous section has an index starting from 0. This is similar to that of a Series. Like a Series, you can also set the index for the DataFrame using the `index` property, as in the following code snippet:

```
df = pd.read_csv('data.csv')
days = pd.date_range('20190525', periods=10)
df.index = days
print(df)
```

You should see the following output:

```
                   A         B         C         D
2019-05-25  0.187497  1.122150 -0.988277 -1.985934
2019-05-26  0.360803 -0.562243 -0.340693 -0.986988
2019-05-27 -0.040627  0.067333 -0.452978  0.686223
2019-05-28 -0.279572 -0.702492  0.252265  0.958977
2019-05-29  0.537438 -1.737568  0.714727 -0.939288
2019-05-30  0.070011 -0.516443 -1.655689  0.246721
2019-05-31  0.001268  0.951517  2.107360 -0.108726
2019-06-01 -0.185258  0.856520 -0.686285  1.104195
2019-06-02  0.387023  1.706336 -2.452653  0.260466
2019-06-03 -1.054974  0.556775 -0.945219 -0.030295
```

To get the index of the DataFrame, use the `index` property as follows:

```
print(df.index)
```

You will see the following output:

```
DatetimeIndex(['2019-05-25', '2019-05-26', '2019-05-27', '2019-05-28',
               '2019-05-29', '2019-05-30', '2019-05-31', '2019-06-01',
               '2019-06-02', '2019-06-03'],
              dtype='datetime64[ns]', freq='D')
```

If you want to get the values of the entire DataFrame as a two-dimensional `ndarray`, use the `values` property:

```
print(df.values)
```

You should see the following output:

```
[[ 1.874970e-01  1.122150e+00 -9.882770e-01 -1.985934e+00]
 [ 3.608030e-01 -5.622430e-01 -3.406930e-01 -9.869880e-01]
 [-4.062700e-02  6.733300e-02 -4.529780e-01  6.862230e-01]
 [-2.795720e-01 -7.024920e-01  2.522650e-01  9.589770e-01]
 [ 5.374380e-01 -1.737568e+00  7.147270e-01 -9.392880e-01]
 [ 7.001100e-02 -5.164430e-01 -1.655689e+00  2.467210e-01]
 [ 1.268000e-03  9.515170e-01  2.107360e+00 -1.087260e-01]
 [-1.852580e-01  8.565200e-01 -6.862850e-01  1.104195e+00]
 [ 3.870230e-01  1.706336e+00 -2.452653e+00  2.604660e-01]
 [-1.054974e+00  5.567750e-01 -9.452190e-01 -3.029500e-02]]
```

Generating Descriptive Statistics on the DataFrame

The Pandas DataFrame comes with a few useful functions to provide you with some detailed statistics about the values in the DataFrame. For example, you can use the `describe()` function to get values such as count, mean, standard deviation, minimum and maximum, as well as the various quartiles:

```
print(df.describe())
```

Using the DataFrame that you have used in the previous section, you should see the following values:

```
              A          B          C          D
count 10.000000  10.000000  10.000000  10.000000
mean  -0.001639   0.174188  -0.444744  -0.079465
std    0.451656   1.049677   1.267397   0.971164
min   -1.054974  -1.737568  -2.452653  -1.985934
25%   -0.149100  -0.550793  -0.977513  -0.731647
50%    0.035640   0.312054  -0.569632   0.108213
75%    0.317477   0.927768   0.104026   0.579784
max    0.537438   1.706336   2.107360   1.104195
```

If you simply want to compute the mean in the DataFrame, you can use the `mean()` function, indicating the axis:

```
print(df.mean(0))    # 0 means compute the mean for each columns
```

You should get the following output:

```
A   -0.001639
B    0.174188
C   -0.444744
D   -0.079465
dtype: float64
```

If you want to get the mean for each row, set the axis to 1:

```
print(df.mean(1))   # 1 means compute the mean for each row
```

You should get the following output:

```
2019-05-25   -0.416141
2019-05-26   -0.382280
2019-05-27    0.064988
2019-05-28    0.057294
2019-05-29   -0.356173
2019-05-30   -0.463850
2019-05-31    0.737855
2019-06-01    0.272293
2019-06-02   -0.024707
2019-06-03   -0.368428
Freq: D, dtype: float64
```

Extracting from DataFrames

In Chapter 2, "Extending Python Using NumPy," you learned about NumPy and how slicing allows you to extract part of a NumPy array. Likewise, in Pandas, slicing applies to both Series and DataFrames.

Because extracting rows and columns in DataFrames is one of the most common tasks that you will perform with DataFrames (and potentially can be confusing), let's walk through the various methods one step at a time so that you have time to digest how they work.

Selecting the First and Last Five Rows

Sometimes, the DataFrame might be too lengthy, and you just want to take a glimpse of the first few rows in the DataFrame. For this purpose, you can use the `head()` function:

```
print(df.head())
```

The `head()` function prints out the first five rows in the DataFrame:

```
                   A         B         C         D
2019-05-25  0.187497  1.122150 -0.988277 -1.985934
2019-05-26  0.360803 -0.562243 -0.340693 -0.986988
2019-05-27 -0.040627  0.067333 -0.452978  0.686223
2019-05-28 -0.279572 -0.702492  0.252265  0.958977
2019-05-29  0.537438 -1.737568  0.714727 -0.939288
```

If you want more than five rows (or less than five), you can indicate the number of rows that you want in the `head()` function as follows:

```
print(df.head(8))     # prints out the first 8 rows
```

There is also a `tail()` function:

```
print(df.tail())
```

The `tail()` function prints the last five rows:

```
                   A         B         C         D
2019-05-30  0.070011 -0.516443 -1.655689  0.246721
2019-05-31  0.001268  0.951517  2.107360 -0.108726
2019-06-01 -0.185258  0.856520 -0.686285  1.104195
2019-06-02  0.387023  1.706336 -2.452653  0.260466
2019-06-03 -1.054974  0.556775 -0.945219 -0.030295
```

Like the `head()` function, the `tail()` function allows you to specify the number of rows to print:

```
print(df.tail(8))     # prints out the last 8 rows
```

Selecting a Specific Column in a DataFrame

To obtain one or more columns in a DataFrame, you can specify the column label as follows:

```
print(df['A'])
# same as
print(df.A)
```

This will print out the "A" column together with its index:

```
2019-05-25    0.187497
2019-05-26    0.360803
2019-05-27   -0.040627
2019-05-28   -0.279572
2019-05-29    0.537438
2019-05-30    0.070011
2019-05-31    0.001268
2019-06-01   -0.185258
2019-06-02    0.387023
2019-06-03   -1.054974
Freq: D, Name: A, dtype: float64
```

Essentially, what you get in return is a Series. If you want to retrieve more than one column, pass in a list containing the column labels:

```
print(df[['A', 'B']])
```

You should see the following output:

```
                   A         B
2019-05-25  0.187497  1.122150
2019-05-26  0.360803 -0.562243
2019-05-27 -0.040627  0.067333
2019-05-28 -0.279572 -0.702492
2019-05-29  0.537438 -1.737568
2019-05-30  0.070011 -0.516443
2019-05-31  0.001268  0.951517
2019-06-01 -0.185258  0.856520
2019-06-02  0.387023  1.706336
2019-06-03 -1.054974  0.556775
```

In this case, instead of a Series, you are now getting a DataFrame.

Slicing Based on Row Number

First, let's extract a range of rows in the DataFrame:

```
print(df[2:4])
```

This extracts row numbers 2 through 4 (not including row 4) from the DataFrame, and you should see the following output:

```
                  A         B         C         D
2019-05-27 -0.040627  0.067333 -0.452978  0.686223
2019-05-28 -0.279572 -0.702492  0.252265  0.958977
```

You can also use the `iloc` indexer for extracting rows based on row number:

```
print(df.iloc[2:4])
```

This will produce the same output as the preceding code snippet.

Note that if you wish to extract specific rows (and not a range of rows) using row numbers, you need to use the `iloc` indexer like this:

```
print(df.iloc[[2,4]])
```

This will print the following output:

```
                  A         B         C         D
2019-05-27 -0.040627  0.067333 -0.452978  0.686223
2019-05-29  0.537438 -1.737568  0.714727 -0.939288
```

Without using the `iloc` indexer, the following will not work:

```
print(df[[2,4]])   # error; need to use the iloc indexer
```

The same applies when extracting a single row using a row number; you need to use `iloc`:

```
print(df.iloc[2])  # prints out row number 2
```

Slicing Based on Row and Column Numbers

If you wish to extract specific rows and columns in a DataFrame, you need to use the `iloc` indexer. The following code snippet extracts row numbers 2 to 3, and column numbers 1 to 3:

```
print(df.iloc[2:4, 1:4])
```

You should get the following output:

```
                  B         C         D
2019-05-27  0.067333 -0.452978  0.686223
2019-05-28 -0.702492  0.252265  0.958977
```

You can also extract specific rows and columns using a list as follows:

```
print(df.iloc[[2,4], [1,3]])
```

The preceding statement prints out row numbers 2 and 4, and column numbers 1 and 3:

```
                   B         D
2019-05-27  0.067333  0.686223
2019-05-29 -1.737568 -0.939288
```

TIP **To summarize, if you want to extract a range of rows using slicing, you can simply use the following syntax:** `df[`*`start _ row`*`:`*`end _ row`*`]`**. If you want to extract specific rows or columns, use the** `iloc` **indexer:** `df.iloc[[`*`row _ 1`*`,`*`row _ 2`*`,...,`*`row _ n`*`],[`*`column _ 1`*`,`*`column _ 2`*`,...,`*`column _ n`*`]]`**.**

Slicing Based on Labels

Besides extracting rows and columns using their row and column numbers, you can also extract them by label (value). For example, the following code snippet extracts a range of *rows* using their index values (which is of `DatetimeIndex` type):

```
print(df['20190601':'20190603'])
```

This will print out the following output:

```
                   A         B         C         D
2019-06-01 -0.185258  0.856520 -0.686285  1.104195
2019-06-02  0.387023  1.706336 -2.452653  0.260466
2019-06-03 -1.054974  0.556775 -0.945219 -0.030295
```

You can also use the `loc` indexer as follows:

```
print(df.loc['20190601':'20190603'])
```

Using the `loc` indexer is mandatory if you want to extract the *columns* using their values, as the following example shows:

```
print(df.loc['20190601':'20190603', 'A':'C'])
```

The preceding statement prints out the following:

```
                   A         B         C
2019-06-01 -0.185258  0.856520 -0.686285
2019-06-02  0.387023  1.706336 -2.452653
2019-06-03 -1.054974  0.556775 -0.945219
```

TIP **Unlike slicing by number, where *start*:*end* means extracting row *start* through row *end* but not including *end*, slicing by value will include the *end* row.**

You can also extract specific columns:

```
print(df.loc['20190601':'20190603', ['A','C']])
```

The preceding statement prints out the following:

```
                   A         C
2019-06-01 -0.185258 -0.686285
2019-06-02  0.387023 -2.452653
2019-06-03 -1.054974 -0.945219
```

If you want to extract a specific row, use the `loc` indexer as follows:

```
print(df.loc['20190601'])
```

It will print out the following:

```
A   -0.185258
B    0.856520
C   -0.686285
D    1.104195
Name: 2019-06-01 00:00:00, dtype: float64
```

Oddly, if you want to extract specific rows with `datetime` as the index, you cannot simply pass the date value to the `loc` indexer as follows:

```
print(df.loc[['20190601','20190603']])   # KeyError
```

First, you need to convert the date into a `datetime` format:

```
from datetime import datetime
date1 = datetime(2019, 6, 1, 0, 0, 0)
date2 = datetime(2019, 6, 3, 0, 0, 0)
print(df.loc[[date1,date2]])
```

You will now see the output like this:

```
                   A         B         C         D
2019-06-01 -0.185258  0.856520 -0.686285  1.104195
2019-06-03 -1.054974  0.556775 -0.945219 -0.030295
```

If you want a specific row and specific columns, you can extract them as follows:

```
print(df.loc[date1, ['A','C']])
```

And the output will look like this:

```
A   -0.185258
C   -0.686285
Name: 2019-06-01 00:00:00, dtype: float64
```

In the preceding example, because there is only a single specified date, the result is a Series.

TIP To summarize, if you want to extract a range of rows using their labels, you can simply use the following syntax: `df[start _ label:end _ label]`**. If you want to extract specific rows or columns, use the** `loc` **indexer with the following syntax:** `df.loc[[row _ 1 _ label,row _ 2 _ label,...,row _ n _ label],[column _ 1 _ label,column _ 2 _ label,...,column _ n _ label]]`**.**

Selecting a Single Cell in a DataFrame

If you simply wish to access a single cell in a DataFrame, there is a function that does just that: `at()`. Using the same example as in the previous section, if you want to get the value of a specific cell, you can use the following code snippet:

```
from datetime import datetime
d = datetime(2019, 6, 3, 0, 0, 0)
print(df.at[d,'B'])
```

You should see the following output:

```
0.556775
```

Selecting Based on Cell Value

If you want to select a subset of the DataFrame based on certain values in the cells, you can use the Boolean Indexing method, as described in Chapter 2. The following code snippet prints out all of the rows that have positive values in the A and B columns:

```
print(df[(df.A > 0) & (df.B>0)])
```

You should see the following output:

```
                   A         B         C         D
2019-05-25  0.187497  1.122150 -0.988277 -1.985934
2019-05-31  0.001268  0.951517  2.107360 -0.108726
2019-06-02  0.387023  1.706336 -2.452653  0.260466
```

Transforming DataFrames

If you need to reflect the DataFrame over its main diagonal (converting columns to rows and rows to columns), you can use the `transpose()` function:

```
print(df.transpose())
```

Alternatively, you can just use the `T` property, which is an accessor to the `transpose()` function:

```
print(df.T)
```

In either case, you will see the following output:

```
   2019-05-25  2019-05-26  2019-05-27  2019-05-28  2019-05-29  2019-05-30  \
A    0.187497    0.360803   -0.040627   -0.279572    0.537438    0.070011
B    1.122150   -0.562243    0.067333   -0.702492   -1.737568   -0.516443
C   -0.988277   -0.340693   -0.452978    0.252265    0.714727   -1.655689
D   -1.985934   -0.986988    0.686223    0.958977   -0.939288    0.246721

   2019-05-31  2019-06-01  2019-06-02  2019-06-03
A    0.001268   -0.185258    0.387023   -1.054974
B    0.951517    0.856520    1.706336    0.556775
C    2.107360   -0.686285   -2.452653   -0.945219
D   -0.108726    1.104195    0.260466   -0.030295
```

Checking to See If a Result Is a DataFrame or Series

One of the common problems that you will face with Pandas is knowing if the result that you have obtained is a Series or a DataFrame. To solve this mystery, here is a function that can make your life easier:

```
def checkSeriesOrDataframe(var):
    if isinstance(var, pd.DataFrame):
        return 'Dataframe'
    if isinstance(var, pd.Series):
        return 'Series'
```

Sorting Data in a DataFrame

There are two ways that you can sort the data in a DataFrame:

1. Sort by labels (axis) using the `sort _ index()` function
2. Sort by value using the `sort _ values()` function

Sorting by Index

To sort using the axis, you need to specify if you want to sort by index or column. Setting the `axis` parameter to `0` indicates that you want to sort by index:

```
print(df.sort_index(axis=0, ascending=False))  # axis = 0 means sort by
                                               # index
```

Based on the preceding statement, the DataFrame is now sorted according to the index in descending order:

```
                   A         B         C         D
2019-06-03 -1.054974  0.556775 -0.945219 -0.030295
2019-06-02  0.387023  1.706336 -2.452653  0.260466
```

```
2019-06-01 -0.185258  0.856520 -0.686285  1.104195
2019-05-31  0.001268  0.951517  2.107360 -0.108726
2019-05-30  0.070011 -0.516443 -1.655689  0.246721
2019-05-29  0.537438 -1.737568  0.714727 -0.939288
2019-05-28 -0.279572 -0.702492  0.252265  0.958977
2019-05-27 -0.040627  0.067333 -0.452978  0.686223
2019-05-26  0.360803 -0.562243 -0.340693 -0.986988
2019-05-25  0.187497  1.122150 -0.988277 -1.985934
```

TIP **Note that the** `sort _ index()` **function returns the sorted DataFrame. The original DataFrame is not affected. If you want the original DataFrame to be sorted, use the** `inplace` **parameter and set it to** `True`**. In general, most operations involving DataFrames do not alter the original DataFrame. So** `inplace` **is by default set to** `False`**. When** `inplace` **is set to** `True`**, the function returns** `None` **as the result.**

Setting the `axis` parameter to `1` indicates that you want to sort by column labels:

```
print(df.sort_index(axis=1, ascending=False))  # axis = 1 means sort by
                                               # column
```

The DataFrame is now sorted based on the column labels (in descending order):

```
                   D         C         B         A
2019-05-25 -1.985934 -0.988277  1.122150  0.187497
2019-05-26 -0.986988 -0.340693 -0.562243  0.360803
2019-05-27  0.686223 -0.452978  0.067333 -0.040627
2019-05-28  0.958977  0.252265 -0.702492 -0.279572
2019-05-29 -0.939288  0.714727 -1.737568  0.537438
2019-05-30  0.246721 -1.655689 -0.516443  0.070011
2019-05-31 -0.108726  2.107360  0.951517  0.001268
2019-06-01  1.104195 -0.686285  0.856520 -0.185258
2019-06-02  0.260466 -2.452653  1.706336  0.387023
2019-06-03 -0.030295 -0.945219  0.556775 -1.054974
```

Sorting by Value

To sort by value, use the `sort _ values()` function. The following statement sorts the DataFrame based on the values in column "A":

```
print(df.sort_values('A', axis=0))
```

The output now is now sorted (in ascending order) based on the value of column "A" (the values are highlighted). Notice that the index is now jumbled up:

```
                   A         B         C         D
2019-06-03 -1.054974  0.556775 -0.945219 -0.030295
2019-05-28 -0.279572 -0.702492  0.252265  0.958977
```

```
2019-06-01 -0.185258  0.856520 -0.686285  1.104195
2019-05-27 -0.040627  0.067333 -0.452978  0.686223
2019-05-31  0.001268  0.951517  2.107360 -0.108726
2019-05-30  0.070011 -0.516443 -1.655689  0.246721
2019-05-25  0.187497  1.122150 -0.988277 -1.985934
2019-05-26  0.360803 -0.562243 -0.340693 -0.986988
2019-06-02  0.387023  1.706336 -2.452653  0.260466
2019-05-29  0.537438 -1.737568  0.714727 -0.939288
```

To sort based on a particular index, set the `axis` parameter to `1`:

```
print(df.sort_values('20190601', axis=1))
```

You can see that the DataFrame is now sorted (in ascending order) based on the row whose index is `2019-06-01` (the values are highlighted):

```
                   C         A         B         D
2019-05-25 -0.988277  0.187497  1.122150 -1.985934
2019-05-26 -0.340693  0.360803 -0.562243 -0.986988
2019-05-27 -0.452978 -0.040627  0.067333  0.686223
2019-05-28  0.252265 -0.279572 -0.702492  0.958977
2019-05-29  0.714727  0.537438 -1.737568 -0.939288
2019-05-30 -1.655689  0.070011 -0.516443  0.246721
2019-05-31  2.107360  0.001268  0.951517 -0.108726
2019-06-01 -0.686285 -0.185258  0.856520  1.104195
2019-06-02 -2.452653  0.387023  1.706336  0.260466
2019-06-03 -0.945219 -1.054974  0.556775 -0.030295
```

Applying Functions to a DataFrame

You can also apply functions to values in a DataFrame using the `apply()` function. First, let's define two lambda functions as follows:

```
import math
sq_root = lambda x: math.sqrt(x) if x > 0 else x
sq      = lambda x: x**2
```

The first function, `sq _ root()`, takes the square root of the value `x` if it is a positive number. The second function, `sq()`, takes the square of the value `x`.

It is important to note that objects passed to the `apply()` function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`).

We can now apply the functions to the DataFrame. First, apply the `sq _ root()` function to column "B":

```
print(df.B.apply(sq_root))
```

Since the result of `df.B` is a Series, we can apply the `sq _ root()` function to it and it will return the following results:

```
2019-05-25    1.029231
2019-05-26   -0.562243
2019-05-27    0.509398
2019-05-28   -0.702492
2019-05-29   -1.737568
2019-05-30   -0.516443
2019-05-31    0.987652
2019-06-01    0.962021
2019-06-02    1.142921
2019-06-03    0.863813
Freq: D, Name: B, dtype: float64
```

You can also apply the `sq()` function to `df.B`:

```
print(df.B.apply(sq))
```

You should see the following results:

```
2019-05-25    1.122150
2019-05-26    0.316117
2019-05-27    0.067333
2019-05-28    0.493495
2019-05-29    3.019143
2019-05-30    0.266713
2019-05-31    0.951517
2019-06-01    0.856520
2019-06-02    1.706336
2019-06-03    0.556775
Freq: D, Name: B, dtype: float64
```

If you apply the `sq _ root()` function to the DataFrame as shown here,

```
df.apply(sq_root)    # ValueError
```

you will get the following error:

```
ValueError: ('The truth value of a Series is ambiguous. Use a.empty,
a.bool(), a.item(), a.any() or a.all().', 'occurred at index A')
```

This is because the object passed into the `apply()` function in this case is a DataFrame, not a Series. Interestingly, you can apply the `sq()` function to the DataFrame:

```
df.apply(sq)
```

This will print out the following:

```
                   A         B         C         D
2019-05-25  0.035155  1.259221  0.976691  3.943934
2019-05-26  0.130179  0.316117  0.116072  0.974145
```

```
2019-05-27  0.001651  0.004534  0.205189  0.470902
2019-05-28  0.078161  0.493495  0.063638  0.919637
2019-05-29  0.288840  3.019143  0.510835  0.882262
2019-05-30  0.004902  0.266713  2.741306  0.060871
2019-05-31  0.000002  0.905385  4.440966  0.011821
2019-06-01  0.034321  0.733627  0.470987  1.219247
2019-06-02  0.149787  2.911583  6.015507  0.067843
2019-06-03  1.112970  0.309998  0.893439  0.000918
```

If you want to apply the `sq_root()` function to the entire DataFrame, you can iterate through the columns and apply the function to each column:

```
for column in df:
    df[column] = df[column].apply(sq_root)
print(df)
```

The result will now look like this:

```
                   A         B         C         D
2019-05-25  0.433009  1.059316 -0.988277 -1.985934
2019-05-26  0.600669 -0.562243 -0.340693 -0.986988
2019-05-27 -0.040627  0.259486 -0.452978  0.828386
2019-05-28 -0.279572 -0.702492  0.502260  0.979274
2019-05-29  0.733102 -1.737568  0.845415 -0.939288
2019-05-30  0.264596 -0.516443 -1.655689  0.496710
2019-05-31  0.035609  0.975457  1.451675 -0.108726
2019-06-01 -0.185258  0.925484 -0.686285  1.050807
2019-06-02  0.622112  1.306268 -2.452653  0.510359
2019-06-03 -1.054974  0.746174 -0.945219 -0.030295
```

The `apply()` function can be applied on either axis: *index* (0; apply function to each column) or *column* (1; apply function to each row). For the two particular lambda functions that we have seen thus far, it does not matter which axis you apply it to, and the result would be the same. However, for some functions, the axis that you apply it to does matter. For example, the following statement uses the `sum()` function from NumPy and applies it to the rows of the DataFrame:

```
print(df.apply(np.sum, axis=0))
```

Essentially, you are summing up all of the values in each column. You should see the following:

```
A    1.128665
B    1.753438
C   -4.722444
D   -0.185696
dtype: float64
```

If you set `axis` to `1` as follows,

```
print(df.apply(np.sum, axis=1))
```

you will see the summation applied across each row:

```
2019-05-25   -1.481886
2019-05-26   -1.289255
2019-05-27    0.594267
2019-05-28    0.499470
2019-05-29   -1.098339
2019-05-30   -1.410826
2019-05-31    2.354015
2019-06-01    1.104747
2019-06-02   -0.013915
2019-06-03   -1.284314
Freq: D, dtype: float64
```

Adding and Removing Rows and Columns in a DataFrame

So far, all of the previous sections have involved extracting rows and columns from DataFrames, as well as how to sort DataFrames. In this section, we will focus on how to add and remove columns in DataFrames.

Consider the following code snippet, where a DataFrame is created from a dictionary:

```
import pandas as pd

data = {'name': ['Janet', 'Nad', 'Timothy', 'June', 'Amy'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [6, 13, 14, 1, 7]}

df = pd.DataFrame(data, index =
       ['Singapore', 'China', 'Japan', 'Sweden', 'Norway'])
print(df)
```

The DataFrame looks like this:

```
             name  reports  year
Singapore   Janet        6  2012
China         Nad       13  2012
Japan     Timothy       14  2013
Sweden       June        1  2014
Norway        Amy        7  2014
```

Adding a Column

The following code snippet shows you how to add a new column named "school" to the DataFrame:

```
import numpy as np

schools = np.array(["Cambridge","Oxford","Oxford","Cambridge","Oxford"])
df["school"] = schools
print(df)
```

Printing the DataFrame will look like this:

```
              name  reports  year     school
Singapore    Janet        6  2012  Cambridge
China          Nad       13  2012     Oxford
Japan      Timothy       14  2013     Oxford
Sweden        June        1  2014  Cambridge
Norway         Amy        7  2014     Oxford
```

Removing Rows

To remove one or more rows, use the `drop()` function. The following code snippet removes the two rows whose index value is "China" and "Japan":

```
print(df.drop(['China', 'Japan'])) # drop rows based on value of index
```

The following output proves that the two rows are removed:

```
            name  reports  year     school
Singapore  Janet        6  2012  Cambridge
Sweden      June        1  2014  Cambridge
Norway       Amy        7  2014     Oxford
```

TIP **Like the `sort _ index()` function, by default the `drop()` function does not affect the original DataFrame. Use the `inplace` parameter if you want to modify the original DataFrame.**

If you want to drop a row based on a particular column value, specify the column name and the condition like this:

```
print(df[df.name != 'Nad'])         # drop row based on column value
```

The preceding statement drops the row whose name is "`Nad`":

```
              name  reports  year     school
Singapore    Janet        6  2012  Cambridge
```

```
Japan      Timothy       14  2013     Oxford
Sweden        June        1  2014  Cambridge
Norway         Amy        7  2014     Oxford
```

You can also remove rows based on row number:

```
print(df.drop(df.index[1]))
```

The preceding statement drops row number 1 (the second row):

```
              name  reports  year     school
Singapore    Janet        6  2012  Cambridge
Japan      Timothy       14  2013     Oxford
Sweden        June        1  2014  Cambridge
Norway         Amy        7  2014     Oxford
```

Since `df.index[1]` returns "China", the preceding statement is equivalent to `df.drop['China']`.

If you want to drop multiple rows, specify the row numbers represented as a list:

```
print(df.drop(df.index[[1,2]]))                # remove the second and
third row
```

The preceding statement removes row numbers 1 and 2 (the second and the third row):

```
            name  reports  year     school
Singapore  Janet        6  2012  Cambridge
Sweden      June        1  2014  Cambridge
Norway       Amy        7  2014     Oxford
```

The following removes the second to last row:

```
print(df.drop(df.index[-2]))                   # remove second last row
```

You should see the following output:

```
              name  reports  year     school
Singapore    Janet        6  2012  Cambridge
China          Nad       13  2012     Oxford
Japan      Timothy       14  2013     Oxford
Norway         Amy        7  2014     Oxford
```

Removing Columns

The `drop()` function drops rows by default, but if you want to drop columns instead, set the `axis` parameter to `1` like this:

```
print(df.drop('reports', axis=1))   # drop column
```

The preceding code snippet drops the `reports` column:

```
              name  year     school
Singapore    Janet  2012  Cambridge
China          Nad  2012     Oxford
Japan      Timothy  2013     Oxford
Sweden        June  2014  Cambridge
Norway         Amy  2014     Oxford
```

If you want to drop by column number, specify the column number using the `columns` indexer:

```
print(df.drop(df.columns[1], axis=1))   # drop using columns number
```

This will drop the second column ("`reports`"):

```
              name  year     school
Singapore    Janet  2012  Cambridge
China          Nad  2012     Oxford
Japan      Timothy  2013     Oxford
Sweden        June  2014  Cambridge
Norway         Amy  2014     Oxford
```

You can also drop multiple columns:

```
print(df.drop(df.columns[[1,3]], axis=1))   # drop multiple columns
```

This will drop the second and fourth columns ("`reports`" and "`school`"):

```
              name  year
Singapore    Janet  2012
China          Nad  2012
Japan      Timothy  2013
Sweden        June  2014
Norway         Amy  2014
```

Generating a Crosstab

In statistics, a *crosstab* is used to aggregate and jointly display the distribution of two or more variables. It shows the relationships between these variables. Consider the following example:

```
df = pd.DataFrame(
    {
        "Gender": ['Male','Male','Female','Female','Female'],
        "Team"  : [1,2,3,3,1]
    })
print(df)
```

Here you are creating a DataFrame using a dictionary. When the DataFrame is printed out, you will see the following:

```
   Gender  Team
0    Male     1
1    Male     2
2  Female     3
3  Female     3
4  Female     1
```

This DataFrame shows the gender of each person and the team to which the person belongs. Using a crosstab, you would be able to summarize the data and generate a table to show the distribution of each gender for each team. To do that, you use the `crosstab()` function:

```
print("Displaying the distribution of genders in each team")
print(pd.crosstab(df.Gender, df.Team))
```

You will see the following output:

```
Displaying the distribution of genders in each team

Team    1  2  3
Gender
Female  1  0  2
Male    1  1  0
```

If you want to see the distribution of each team for each gender, you simply reverse the argument:

```
print(pd.crosstab(df.Team, df.Gender))
```

You will see the following output:

```
Gender  Female  Male
Team
1            1     1
2            0     1
3            2     0
```

Summary

In this chapter, you witnessed the use of Pandas to represent tabular data. You learned about the two main Pandas data structures: Series and DataFrame. I attempted to keep things simple and to show you some of the most common operations that you would perform on these data structures. As extracting rows and columns from DataFrames is so common, I have summarized some of these operations in Table 3.1.

Table 3.1: Common DataFrame Operations

DESCRIPTION	CODE EXAMPLES

Extract a range of rows using row numbers	`df[2:4]`
	`df.iloc[2:4]`

Extract a single row using row number	`df.iloc[2]`
Extract a range of rows and range of columns	`df.iloc[2:4, 1:4]`
Extract a range of rows and specific columns using positional values	`df.iloc[2:4, [1,3]]`
Extract specific row(s) and column(s)	`df.iloc[[2,4], [1,3]]`
Extract a range of rows using labels	`df['20190601':'20190603']`
Extract a single row based on its label	`df.loc['20190601']`
Extract specific row(s) using their labels	`df.loc[[date1,date2]]`

Extract specific row(s) and column(s) using their labels	`df.loc[[date1,date2], ['A','C']]`
	`df.loc[[date1,date2], 'A':'C']`

Extract a range of rows and columns using their labels	`df.loc[date1:date2, 'A':'C']`