CHAPTER 4

The pandas Library—An Introduction

This chapter gets into the heart of this book: the pandas library. This fantastic Python library is a perfect tool for anyone who wants to perform data analysis using Python as a programming language.

First you will learn about the fundamental aspects of this library and how to install it on your system, and then you will become familiar with the two data structures called *series* and *dataframes*. During the course of the chapter, you will work with a basic set of functions provided by the pandas library, in order to perform the most common data processing tasks. Getting familiar with these operations is a key goal of the rest of the book. This is why it is very important to repeat this chapter until you feel comfortable with its content.

Furthermore, with a series of examples you will learn some particularly new concepts introduced in the pandas library: indexing data structures. You will learn how to get the most of this feature for data manipulation in this chapter and in the next chapters.

Finally, you will see how to extend the concept of indexing to multiple levels at the same time, through the process called hierarchical indexing.

pandas: The Python Data Analysis Library

pandas is an open source Python library for highly specialized data analysis. It is currently the reference point that all professionals using the Python language need to study for the statistical purposes of analysis and decision making.

This library was designed and developed primarily by Wes McKinney starting in 2008. In 2012, Sien Chang, one of his colleagues, was added to the development. Together they set up one of the most used libraries in the Python community.

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_4

pandas arises from the need to have a specific library to analyze data that provides, in the simplest possible way, all the instruments for data processing, data extraction, and data manipulation.

This Python package is designed on the basis of the NumPy library. This choice, we can say, was critical to the success and the rapid spread of pandas. In fact, this choice not only makes this library compatible with most other modules, but also takes advantage of the high quality of the NumPy module.

Another fundamental choice was to design ad hoc data structures for data analysis. In fact, instead of using existing data structures built into Python or provided by other libraries, two new data structures were developed.

These data structures are designed to work with relational data or labeled data, thus allowing you to manage data with features similar to those designed for SQL relational databases and Excel spreadsheets.

Throughout the book in fact, you will see a series of basic operations for data analysis, which are normally used on database tables and spreadsheets. pandas in fact provides an extended set of functions and methods that allow you to perform these operations efficiently.

So pandas' main purpose is to provide all the building blocks for anyone approaching the data analysis world.

Installation of pandas

The easiest and most general way to install the pandas library is to use a prepackaged solution, i.e., installing it through an Anaconda or Enthought distribution.

Installation from Anaconda

For those who choose to use the Anaconda distribution, managing the installation is very simple. First you have to see if the pandas module is installed and, if so, which version. To do this, type the following command from the terminal:

```
conda list pandas
```

Since I have the module installed on my PC (Windows), I get the following result:

```
# packages in environment at C:\Users\Fabio\Anaconda:
#
pandas                    0.20.3              py36hce827b7_2
```

If you do not have pandas installed, you will need to install it. Enter the following command:

```
conda install pandas
```

Anaconda will immediately check all dependencies, managing the installation of other modules, without you having to worry too much.

```
Solving environment: done
## Package Plan ##

Environment location: C:\Users\Fabio\Anaconda3
added / updated specs:

   - pandas
```

The following new packages will be installed:

```
    Pandas: 0.22.0-py36h6538335_0
Proceed ([y]/n)?
Press the y key on your keyboard to continue the installation.
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

If you want to upgrade your package to a newer version, the command to do so is very simple and intuitive:

```
conda update pandas
```

The system will check the version of pandas and the version of all the modules on which it depends and then suggest any updates. It will then ask if you want to proceed to the update.

Installation from PyPI

pandas can also be installed by PyPI using this command:

```
pip install pandas
```

Installation on Linux

If you're working on a Linux distribution, and you choose not to use any of these prepackaged distributions, you can install the pandas module like any other package.

On Debian and Ubuntu distributions, use this command:

```
sudo apt-get install python-pandas
```

While on OpenSuse and Fedora, enter the following command:

```
zypper in python-pandas
```

Installation from Source

If you want to compile your pandas module from the source code, you can find what you need on GitHub at `https://github.com/pandas-dev/pandas`:

```
git clone git://github.com/pydata/pandas.git
cd pandas
python setup.py install
```

Make sure you have installed Cython at compile time. For more information, read the documentation available on the Web, including the official page (`http://pandas.pydata.org/pandas-docs/stable/install.html`).

A Module Repository for Windows

If you are working on Windows and prefer to manage your packages in order to always have the most current modules, there is also a resource on the Internet where you can download many third-party modules—Christoph Gohlke's Python Extension Packages for Windows repository (`www.lfd.uci.edu/~gohlke/pythonlibs/`). Each module is supplied with the format archival WHL (wheel) in both 32-bit and 64-bit. To install each module, you have to use the pip application (see PyPI in Chapter 2).

```
pip install SomePackege-1.0.whl
```

For example, for pandas you can find and download the following package:

```
pip install pandas-0.22.0-cp36-cp36m-win_amd64.whl
```

When choosing the module, be careful to choose the correct version for your version of Python and the architecture on which you're working. Furthermore, while NumPy does not require the installation of other packages, on the contrary, pandas has many dependencies. So make sure you get them all. The installation order is not important.

The disadvantage of this approach is that you need to install the packages individually without a package manager that can help manage versioning and interdependencies between the various packages. The advantage is greater mastery of the modules and their versions, so you have the most current modules possible without depending on the choices of the distributions.

Testing Your pandas Installation

The pandas library can run a check after it's installed to verify the internal controls (the official documentation states that the test provides a 97% coverage of all the code inside).

First, make sure you have installed the `nose` module in your Python distribution (see the "Nose Module" sidebar). If you did, you can start the test by entering the following command:

```
nosetests pandas
```

The test will take several minutes and in the end it will show a list of any problems encountered.

NOSE MODULE

This module is designed for testing Python code during the development phases of a project or a Python module in particular. This module extends the capabilities of the `unittest` module. The Python module involved in testing the code, however, making its coding much simpler and easier.

I suggest you read this article at `http://pythontesting.net/framework/nose/nose-introduction/` for more information.

Getting Started with pandas

The best way to get started with pandas is to open a Python shell and type commands one by one. This way, you have the opportunity to become familiar with the individual functions and data structures that are explained in this chapter.

Furthermore, the data and functions defined in the various examples remain valid throughout the chapter, which means you don't have to to define them each time. You are invited, at the end of each example, to repeat the various commands, modify them if appropriate, and control how the values in the data structures vary during operation. This approach is great for getting familiar with the different topics covered in this chapter, leaving you the opportunity to interact freely with what you are reading.

Note This chapter assumes that you have some familiarity with Python and NumPy in general. If you have any difficulty, read Chapters 2 and 3 of this book.

First, open a session on the Python shell and then import the pandas library. The general practice for importing the pandas module is as follows:

```
>>> import pandas as pd
>>> import numpy as np
```

Thus, in this chapter and throughout the book, every time you see `pd` and `np`, you'll make reference to an object or method referring to these two libraries, even though you will often be tempted to import the pandas module in this way:

```
>>> from pandas import *
```

Thus, you no longer have to reference a function, object, or method with `pd`; this approach is not considered good practice by the Python community in general.

Introduction to pandas Data Structures

The heart of pandas is the two primary data structures on which all transactions, which are generally made during the analysis of data, are centralized:

- Series
- Dataframes

The *series*, as you will see, constitutes the data structure designed to accommodate a sequence of one-dimensional data, while the *dataframe*, a more complex data structure, is designed to contain cases with several dimensions.

Although these data structures are not the universal solution to all problems, they do provide a valid and robust tool for most applications. In fact, they remain very simple to understand and use. In addition, many cases of more complex data structures can still be traced to these simple two cases.

However, their peculiarities are based on a particular feature—integration in their structure of index objects and labels. You will see that this feature causes these data structures to be easily manipulated.

The Series

The *series* is the object of the pandas library designed to represent one-dimensional data structures, similar to an array but with some additional features. Its internal structure is simple (see Figure 4-1) and is composed of two arrays associated with each other. The main array holds the data (data of any NumPy type) to which each element is associated with a label, contained within the other array, called the *index*.

Series	
index	value
0	12
1	-4
2	7
3	9

Figure 4-1. *The structure of the series object*

Declaring a Series

To create the series specified in Figure 4-1, you simply call the `Series()` constructor and pass as an argument an array containing the values to be included in it.

```
>>> s = pd.Series([12,-4,7,9])
>>> s
0    12
1    -4
2     7
3     9
dtype: int64
```

As you can see from the output of the series, on the left there are the values in the index, which is a series of labels, and on the right are the corresponding values.

If you do not specify any index during the definition of the series, by default, pandas will assign numerical values increasing from 0 as labels. In this case, the labels correspond to the indexes (position in the array) of the elements in the series object.

Often, however, it is preferable to create a series using meaningful labels in order to distinguish and identify each item regardless of the order in which they were inserted into the series.

In this case it will be necessary, during the constructor call, to include the `index` option and assign an array of strings containing the labels.

```
>>> s = pd.Series([12,-4,7,9], index=['a','b','c','d'])
>>> s
a    12
b    -4
c     7
d     9
dtype: int64
```

If you want to individually see the two arrays that make up this data structure, you can call the two attributes of the series as follows: `index` and `values`.

```
>>> s.values
array([12, -4,  7,  9], dtype=int64)
>>> s.index
Index([u'a', u'b', u'c', u'd'], dtype='object')
```

Selecting the Internal Elements

You can select individual elements as ordinary numpy arrays, specifying the key.

```
>>> s[2]
7
```

Or you can specify the label corresponding to the position of the index.

```
>>> s['b']
-4
```

In the same way you select multiple items in a numpy array, you can specify the following:

```
>>> s[0:2]
a    12
b    -4
dtype: int64
```

In this case, you can use the corresponding labels, but specify the list of labels in an array.

```
>>> s[['b','c']]
b   -4
c    7
dtype: int64
```

Assigning Values to the Elements

Now that you understand how to select individual elements, you also know how to assign new values to them. In fact, you can select the value by index or by label.

```
>>> s[1] = 0
>>> s
a    12
b     0
c     7
d     9
dtype: int64
```

```
>>> s['b'] = 1
>>> s
a    12
b     1
c     7
d     9
dtype: int64
```

Defining a Series from NumPy Arrays and Other Series

You can define a new series starting with NumPy arrays or with an existing series.

```
>>> arr = np.array([1,2,3,4])
>>> s3 = pd.Series(arr)
>>> s3
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s4 = pd.Series(s)
>>> s4
a    12
b     4
c     7
d     9
dtype: int64
```

Always keep in mind that the values contained in the NumPy array or in the original series are not copied, but are passed by reference. That is, the object is inserted dynamically within the new series object. If it changes, for example its internal element varies in value, then those changes will also be present in the new series object.

```
>>> s3
0    1
1    2
2    3
3    4
```

```
dtype: int64
>>> arr[2] = -2
>>> s3
0    1
1    2
2   -2
3    4
dtype: int64
```

As you can see in this example, by changing the third element of the `arr` array, we also modified the corresponding element in the `s3` series.

Filtering Values

Thanks to the choice of the NumPy library as the base of the pandas library and, as a result, for its data structures, many operations that are applicable to NumPy arrays are extended to the series. One of these is filtering values contained in the data structure through conditions.

For example, if you need to know which elements in the series are greater than 8, you write the following:

```
>>> s[s > 8]
a    12
d     9
dtype: int64
```

Operations and Mathematical Functions

Other operations such as operators (+, -, *, and /) and mathematical functions that are applicable to NumPy array can be extended to series.

You can simply write the arithmetic expression for the operators.

```
>>> s / 2
a    6.0
b   -2.0
c    3.5
d    4.5
dtype: float64
```

However, with the NumPy mathematical functions, you must specify the function referenced with `np` and the instance of the series passed as an argument.

```
>>> np.log(s)
a    2.484907
b    0.000000
c    1.945910
d    2.197225
dtype: float64
```

Evaluating Vales

There are often duplicate values in a series. Then you may need to have more information about the samples, including existence of any duplicates and whether a certain value is present in the series.

In this regard, you can declare a series in which there are many duplicate values.

```
>>> serd = pd.Series([1,0,2,1,2,3], index=['white','white','blue','green','
green','yellow'])
>>> serd
white     1
white     0
blue      2
green     1
green     2
yellow    3
dtype: int64
```

To know all the values contained in the series, excluding duplicates, you can use the `unique()` function. The return value is an array containing the unique values in the series, although not necessarily in order.

```
>>> serd.unique()
array([1, 0, 2, 3], dtype=int64)
```

A function that's similar to `unique()` is `value_counts()`, which not only returns unique values but also calculates the occurrences within a series.

```
>>> serd.value_counts()
2    2
1    2
3    1
0    1
dtype: int64
```

Finally, `isin()` evaluates the membership, that is, the given a list of values. This function tells you if the values are contained in the data structure. Boolean values that are returned can be very useful when filtering data in a series or in a column of a dataframe.

```
>>> serd.isin([0,3])
white     False
white      True
blue      False
green     False
green     False
yellow     True
dtype: bool
>>> serd[serd.isin([0,3])]
white     0
yellow    3
dtype: int64
```

NaN Values

As you can see in the previous case, we tried to run the logarithm of a negative number and received `NaN` as a result. This specific value `NaN` (Not a Number) is used in pandas data structures to indicate the presence of an empty field or something that's not definable numerically.

Generally, these `NaN` values are a problem and must be managed in some way, especially during data analysis. These data are often generated when extracting data from a questionable source or when the source is missing data. Furthermore, as you have just seen, the `NaN` values can also be generated in special cases, such as calculations of logarithms of negative values, or exceptions during execution of some calculation or function. In later chapters, you see how to apply different strategies to address the problem of `NaN` values.

Despite their problematic nature, however, pandas allows you to explicitly define NaNs and add them to a data structure, such as a series. Within the array containing the values, you enter np.NaN wherever you want to define a missing value.

```
>>> s2 = pd.Series([5,-3,np.NaN,14])
>>> s2
0     5.0
1    -3.0
2   NaN
3    14.0
dtype: float64
```

The isnull() and notnull() functions are very useful to identify the indexes without a value.

```
>>> s2.isnull()
0    False
1    False
2     True
3    False
dtype: bool
>>> s2.notnull()
0     True
1     True
2    False
3     True
dtype: bool
```

In fact, these functions return two series with Boolean values that contain the True and False values, depending on whether the item is a NaN value or less. The isnull() function returns True at NaN values in the series; inversely, the notnull() function returns True if they are not NaN. These functions are often placed inside filters to make a condition.

```
>>> s2[s2.notnull()]
0     5.0
1    -3.0
3    14.0
```

```
dtype: float64
>>> s2[s2.isnull()]
2   NaN
dtype: float64
```

Series as Dictionaries

An alternative way to think of a series is to think of it as an object `dict` (dictionary). This similarity is also exploited during the definition of an object series. In fact, you can create a series from a previously defined `dict`.

```
>>> mydict = {'red': 2000, 'blue': 1000, 'yellow': 500,
 'orange': 1000}
>>> myseries = pd.Series(mydict)
>>> myseries
red       2000
blue      1000
yellow     500
orange    1000
dtype: int64
```

As you can see from this example, the array of the index is filled with the keys while the data are filled with the corresponding values. You can also define the array indexes separately. In this case, controlling correspondence between the keys of the `dict` and labels array of indexes will run. If there is a mismatch, pandas will add the `NaN` value.

```
>>> colors = ['red','yellow','orange','blue','green']
>>> myseries = pd.Series(mydict, index=colors)
>>> myseries
red       2000.0
yellow     500.0
orange    1000.0
blue      1000.0
green      NaN
dtype: float64
```

Operations Between Series

We have seen how to perform arithmetic operations between series and scalar values. The same thing is possible by performing operations between two series, but in this case even the labels come into play.

In fact, one of the great potentials of this type of data structures is that series can align data addressed differently between them by identifying their corresponding labels.

In the following example, you add two series having only some elements in common with the label.

```
>>> mydict2 = {'red':400,'yellow':1000,'black':700}
>>> myseries2 = pd.Series(mydict2)
>>> myseries + myseries2
black      NaN
blue       NaN
green      NaN
orange     NaN
red       2400.0
yellow    1500.0
dtype: float64
```

You get a new object series in which only the items with the same label are added. All other labels present in one of the two series are still added to the result but have a `NaN` value.

The DataFrame

The *dataframe* is a tabular data structure very similar to a spreadsheet. This data structure is designed to extend series to multiple dimensions. In fact, the dataframe consists of an ordered collection of columns (see Figure 4-2), each of which can contain a value of a different type (numeric, string, Boolean, etc.).

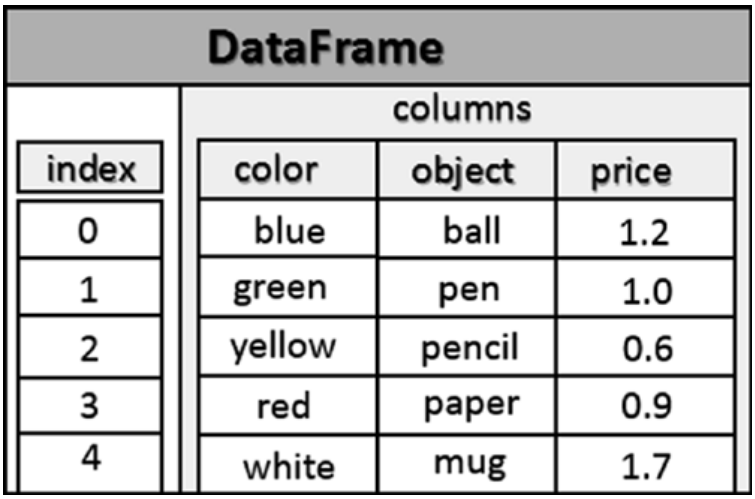


Figure 4-2. *The dataframe structure*

Unlike series, which have an index array containing labels associated with each element, the dataframe has two index arrays. The first index array, associated with the lines, has very similar functions to the index array in series. In fact, each label is associated with all the values in the row. The second array contains a series of labels, each associated with a particular column.

A dataframe may also be understood as a `dict` of series, where the keys are the column names and the values are the series that will form the columns of the dataframe. Furthermore, all elements in each series are mapped according to an array of labels, called the *index*.

Defining a Dataframe

The most common way to create a new dataframe is precisely to pass a `dict` object to the `DataFrame()` constructor. This `dict` object contains a key for each column that you want to define, with an array of values for each of them.

```
>>> data = {'color' : ['blue','green','yellow','red','white'],
                      'object' : ['ball','pen','pencil','paper','mug'],
                      'price' : [1.2,1.0,0.6,0.9,1.7]}
>>> frame = pd.DataFrame(data)
>>> frame
    color  object price
0    blue    ball   1.2
1   green     pen   1.0
```

```
2  yellow  pencil   0.6
3     red   paper   0.9
4   white     mug   1.7
```

If the `dict` object from which you want to create a dataframe contains more data than you are interested in, you can make a selection. In the constructor of the dataframe, you can specify a sequence of columns using the `columns` option. The columns will be created in the order of the sequence regardless of how they are contained in the `dict` object.

```
>>> frame2 = pd.DataFrame(data, columns=['object','price'])
>>> frame2
   object price
0    ball   1.2
1     pen   1.0
2  pencil   0.6
3   paper   0.9
4     mug   1.7
```

Even for dataframe objects, if the labels are not explicitly specified in the Index array, pandas automatically assigns a numeric sequence starting from 0. Instead, if you want to assign labels to the indexes of a dataframe, you have to use the `index` option and assign it an array containing the labels.

```
>>> frame2 = pd.DataFrame(data, index=['one','two','three','four','five'])
>>> frame2
        color  object  price
one      blue    ball    1.2
two     green     pen    1.0
three  yellow  pencil    0.6
four      red   paper    0.9
five    white     mug    1.7
```

Now that we have introduced the two new options called `index` and `columns`, it is easy to imagine an alternative way to define a dataframe. Instead of using a `dict` object, you can define three arguments in the constructor, in the following order—a data matrix, an array containing the labels assigned to the `index` option, and an array containing the names of the columns assigned to the `columns` option.

In many examples, as you will see from now on in this book, to create a matrix of values quickly and easily, you can use `np.arange(16).reshape((4,4))`, which generates a 4x4 matrix of numbers increasing from 0 to 15.

```
>>> frame3 = pd.DataFrame(np.arange(16).reshape((4,4)),
...                   index=['red','blue','yellow','white'],
...                   columns=['ball','pen','pencil','paper'])
>>> frame3
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

Selecting Elements

If you want to know the name of all the columns of a dataframe, you can is specify the `columns` attribute on the instance of the dataframe object.

```
>>> frame.columns
Index(['colors', 'object', 'price'], dtype='object')
```

Similarly, to get the list of indexes, you should specify the `index` attribute.

```
>>> frame.index
RangeIndex(start=0, stop=5, step=1)
```

You can also get the entire set of data contained within the data structure using the `values` attribute.

```
>>> frame.values
array([['blue', 'ball', 1.2],
       ['green', 'pen', 1.0],
       ['yellow', 'pencil', 0.6],
       ['red', 'paper', 0.9],
       ['white', 'mug', 1.7]], dtype=object)
```

Or, if you are interested in selecting only the contents of a column, you can write the name of the column.

```
>>> frame['price']
0    1.2
1    1.0
2    0.6
3    0.9
4    1.7
Name: price, dtype: float64
```

As you can see, the return value is a series object. Another way to do this is to use the column name as an attribute of the instance of the dataframe.

```
>>> frame.price
0    1.2
1    1.0
2    0.6
3    0.9
4    1.7
Name: price, dtype: float64
```

For rows within a dataframe, it is possible to use the `loc` attribute with the index value of the row that you want to extract.

```
>>> frame.loc[2]
color     yellow
object    pencil
price        0.6
Name: 2, dtype: object
```

The object returned is again a series in which the names of the columns have become the label of the array index, and the values have become the data of series.

To select multiple rows, you specify an array with the sequence of rows to insert:

```
>>> frame.loc[[2,4]]
    color  object  price
2  yellow  pencil    0.6
4   white     mug    1.7
```

If you need to extract a portion of a DataFrame, selecting the lines that you want to extract, you can use the reference numbers of the indexes. In fact, you can consider a row as a portion of a dataframe that has the index of the row as the source (in the next 0) value and the line above the one we want as a second value (in the next one).

```
>>> frame[0:1]
  color object  price
0  blue   ball    1.2
```

As you can see, the return value is an object dataframe containing a single row. If you want more than one line, you must extend the selection range.

```
>>> frame[1:3]
    color  object  price
1   green     pen    1.0
2  yellow  pencil    0.6
```

Finally, if what you want to achieve is a single value within a dataframe, you first use the name of the column and then the index or the label of the row.

```
>>> frame['object'][3]
'paper'
```

Assigning Values

Once you understand how to access the various elements that make up a dataframe, you follow the same logic to add or change the values in it.

For example, you have already seen that within the dataframe structure, an array of indexes is specified by the `index` attribute, and the row containing the name of the columns is specified with the `columns` attribute. Well, you can also assign a label, using the `name` attribute, to these two substructures to identify them.

```
>>> frame.index.name = 'id'
>>> frame.columns.name = 'item'
>>> frame
```

```
item   color  object  price
id
0       blue    ball    1.2
1      green     pen    1.0
2     yellow  pencil    0.6
3        red   paper    0.9
4      white     mug    1.7
```

One of the best features of the data structures of pandas is their high flexibility. In fact, you can always intervene at any level to change the internal data structure. For example, a very common operation is to add a new column.

You can do this by simply assigning a value to the instance of the dataframe and specifying a new column name.

```
>>> frame['new'] = 12
>>> frame
   colors  object price  new
0    blue    ball   1.2   12
1   green     pen   1.0   12
2  yellow  pencil   0.6   12
3     red   paper   0.9   12
4   white     mug   1.7   12
```

As you can see from this result, there is a new column called `new` with the value within 12 replicated for each of its elements.

If, however, you want to update the contents of a column, you have to use an array.

```
>>> frame['new'] = [3.0,1.3,2.2,0.8,1.1]
>>> frame
    color  object  price  new
0    blue    ball    1.2  3.0
1   green     pen    1.0  1.3
2  yellow  pencil    0.6  2.2
3     red   paper    0.9  0.8
4   white     mug    1.7  1.1
```

You can follow a similar approach if you want to update an entire column, for example, by using the `np.arange()` function to update the values of a column with a predetermined sequence.

The columns of a dataframe can also be created by assigning a series to one of them, for example by specifying a series containing an increasing series of values through the use of `np.arange()`.

```
>>> ser = pd.Series(np.arange(5))
>>> ser
0    0
1    1
2    2
3    3
4    4
dtype: int64
>>> frame['new'] = ser
>>> frame
    color  object  price  new
0    blue    ball    1.2    0
1   green     pen    1.0    1
2  yellow  pencil    0.6    2
3     red   paper    0.9    3
4   white     mug    1.7    4
```

Finally, to change a single value, you simply select the item and give it the new value.

```
>>> frame['price'][2] = 3.3
```

Membership of a Value

You have already seen the `isin()` function applied to the series to determine the membership of a set of values. Well, this feature is also applicable to dataframe objects.

```
>>> frame.isin([1.0,'pen'])
   color object  price  new
0  False  False  False  False
1  False   True   True  True
2  False  False  False  False
3  False  False  False  False
4  False  False  False  False
```

You get a dataframe containing Boolean values, where `True` indicates values that meet the membership. If you pass the value returned as a condition, then you'll get a new dataframe containing only the values that satisfy the condition.

```
>>> frame[frame.isin([1.0,'pen'])]
  color object  price  new
0   NaN    NaN    NaN  NaN
1   NaN    pen    1.0  1.0
2   NaN    NaN    NaN  NaN
3   NaN    NaN    NaN  NaN
4   NaN    NaN    NaN  NaN
```

Deleting a Column

If you want to delete an entire column and all its contents, use the `del` command.

```
>>> del frame['new']
>>> frame
   colors  object price
0    blue    ball   1.2
1   green     pen   1.0
2  yellow  pencil   3.3
3     red   paper   0.9
4   white     mug   1.7
```

Filtering

Even when a dataframe, you can apply the filtering through the application of certain conditions. For example, say you want to get all values smaller than a certain number, for example 1.2.

```
>>> frame[frame < 1.2]
>>> frame
   colors  object price
0    blue    ball   NaN
1   green     pen   1.0
2  yellow  pencil   NaN
3     red   paper   0.9
4   white     mug   NaN
```

You will get a dataframe containing values less than 1.2, keeping their original position. All others will be replaced with NaN.

DataFrame from Nested dict

A very common data structure used in Python is a nested dict, as follows:

```
nestdict = { 'red': { 2012: 22, 2013: 33 },
                    'white': { 2011: 13, 2012: 22, 2013: 16},
                    'blue': {2011: 17, 2012: 27, 2013: 18}}
```

This data structure, when it is passed directly as an argument to the DataFrame() constructor, will be interpreted by pandas to treat external keys as column names and internal keys as labels for the indexes.

During the interpretation of the nested structure, it is possible that not all fields will find a successful match. pandas compensates for this inconsistency by adding the NaN value to missing values.

```
>>> nestdict = {'red':{2012: 22, 2013: 33},
...             'white':{2011: 13, 2012: 22, 2013: 16},
...             'blue': {2011: 17, 2012: 27, 2013: 18}}
>>> frame2 = pd.DataFrame(nestdict)
>>> frame2
      blue    red  white
2011    17    NaN     13
2012    27   22.0     22
2013    18   33.0     16
```

Transposition of a Dataframe

An operation that you might need when you're dealing with tabular data structures is transposition (that is, columns become rows and rows become columns). pandas allows you to do this in a very simple way. You can get the transposition of the dataframe by adding the T attribute to its application.

```
>>> frame2.T
       2011  2012  2013
blue   17.0  27.0  18.0
red     NaN  22.0  33.0
white  13.0  22.0  16.0
```

The Index Objects

Now that you know what the series and the dataframe are and how they are structured, you can likely perceive the peculiarities of these data structures. Indeed, the majority of their excellent characteristics are due to the presence of an `Index` object that's integrated in these data structures.

The `Index` objects are responsible for the labels on the axes and other metadata as the name of the axes. You have already seen how an array containing labels is converted into an `Index` object and that you need to specify the `index` option in the constructor.

```
>>> ser = pd.Series([5,0,3,8,4], index=['red','blue','yellow','white','green'])
>>> ser.index
Index(['red', 'blue', 'yellow', 'white', 'green'], dtype='object')
```

Unlike all the other elements in the pandas data structures (series and dataframe), the `Index` objects are immutable. Once declared, they cannot be changed. This ensures their secure sharing between the various data structures.

Each `Index` object has a number of methods and properties that are useful when you need to know the values they contain.

Methods on Index

There are some specific methods for indexes available to get some information about indexes from a data structure. For example, `idmin()` and `idmax()` are two functions that return, respectively, the index with the lowest value and the index with the highest value.

```
>>> ser.idxmin()
'blue'
>>> ser.idxmax()
'white'
```

Index with Duplicate Labels

So far, you have met all cases in which indexes within a single data structure have a unique label. Although many functions require this condition to run, this condition is not mandatory on the data structures of pandas.

Define by way of example, a series with some duplicate labels.

```
>>> serd = pd.Series(range(6), index=['white','white','blue','green',
'green','yellow'])
>>> serd
white     0
white     1
blue      2
green     3
green     4
yellow    5
dtype: int64
```

Regarding the selection of elements in a data structure, if there are more values in correspondence of the same label, you will get a series in place of a single element.

```
>>> serd['white']
white    0
white    1
dtype: int64
```

The same logic applies to the dataframe, with duplicate indexes that will return the dataframe.

With small data structures, it is easy to identify any duplicate indexes, but if the structure becomes gradually larger, this starts to become difficult. In this respect, pandas provides you with the `is_unique` attribute belonging to the `Index` objects. This attribute will tell you if there are indexes with duplicate labels inside the structure data (both series and dataframe).

```
>>> serd.index.is_unique
False
>>> frame.index.is_unique
True
```

Other Functionalities on Indexes

Compared to data structures commonly used with Python, you saw that pandas, as well as taking advantage of the high-performance quality offered by NumPy arrays, has chosen to integrate indexes in them.

This choice has proven somewhat successful. In fact, despite the enormous flexibility given by the dynamic structures that already exist, using the internal reference to the structure, such as that offered by the labels, allows developers who must perform operations to carry them out in a simpler and more direct way.

This section analyzes in detail a number of basic features that take advantage of this mechanism.

- Reindexing
- Dropping
- Alignment

Reindexing

It was previously stated that once it's declared in a data structure, the `Index` object cannot be changed. This is true, but by executing a reindexing, you can also overcome this problem.

In fact it is possible to obtain a new data structure from an existing one where indexing rules can be defined again.

```
>>> ser = pd.Series([2,5,7,4], index=['one','two','three','four'])
>>> ser
one      2
two      5
three    7
four     4
dtype: int64
```

In order to reindex this series, pandas provides you with the `reindex()` function. This function creates a new `series` object with the values of the previous series rearranged according to the new sequence of labels.

During reindexing, it is possible to change the order of the sequence of indexes, delete some of them, or add new ones. In the case of a new label, pandas adds `NaN` as the corresponding value.

```
>>> ser.reindex(['three','four','five','one'])
three     7.0
four      4.0
five      NaN
one       2.0
dtype: float64
```

As you can see from the value returned, the order of the labels has been completely rearranged. The value corresponding to the label `two` has been dropped and a new label called `five` is present in the series.

However, to measure the reindexing process, defining the list of the labels can be awkward, especially with a large dataframe. So you could use some method that allows you to fill in or interpolate values automatically.

To better understand the functioning of this mode of automatic reindexing, define the following series.

```
>>> ser3 = pd.Series([1,5,6,3],index=[0,3,5,6])
>>> ser3
0    1
3    5
5    6
6    3
dtype: int64
```

As you can see in this example, the index column is not a perfect sequence of numbers; in fact there are some missing values (1, 2, and 4). A common need would be to perform interpolation in order to obtain the complete sequence of numbers. To achieve this, you will use reindexing with the `method` option set to `ffill`. Moreover, you need to set a range of values for indexes. In this case, to specify a set of values between 0 and 5, you can use `range(6)` as an argument.

```
>>> ser3.reindex(range(6),method='ffill')
0    1
1    1
```

```
2    1
3    5
4    5
5    6
dtype: int64
```

As you can see from the result, the indexes that were not present in the original series were added. By interpolation, those with the lowest index in the original series have been assigned as values. In fact, the indexes 1 and 2 have the value 1, which belongs to index 0.

If you want this index value to be assigned during the interpolation, you have to use the `bfill` method.

```
>>> ser3.reindex(range(6),method='bfill')
0    1
1    5
2    5
3    5
4    6
5    6
dtype: int64
```

In this case, the value assigned to the indexes 1 and 2 is the value 5, which belongs to index 3.

Extending the concepts of reindexing with series to the dataframe, you can have a rearrangement not only for indexes (rows), but also with regard to the columns, or even both. As previously mentioned, adding a new column or index is possible, but since there are missing values in the original data structure, pandas adds `NaN` values to them.

```
>>> frame.reindex(range(5), method='ffill',columns=['colors','price','new',
'object'])
   colors price  new     object
0    blue   1.2  blue    ball
1   green   1.0  green   pen
2  yellow   3.3  yellow  pencil
3     red   0.9  red     paper
4   white   1.7  white   mug
```

Dropping

Another operation that is connected to Index objects is dropping. Deleting a row or a column becomes simple, due to the labels used to indicate the indexes and column names.

Also in this case, pandas provides a specific function for this operation, called drop(). This method will return a new object without the items that you want to delete.

For example, take the case where we want to remove a single item from a series. To do this, define a generic series of four elements with four distinct labels.

```
>>> ser = pd.Series(np.arange(4.), index=['red','blue','yellow','white'])
>>> ser
red       0.0
blue      1.0
yellow    2.0
white     3.0
dtype: float64
```

Now say, for example, that you want to delete the item corresponding to the label yellow. Simply specify the label as an argument of the function drop() to delete it.

```
>>> ser.drop('yellow')
red      0.0
blue     1.0
white    3.0
dtype: float64
```

To remove more items, just pass an array with the corresponding labels.

```
>>> ser.drop(['blue','white'])
red       0.0
yellow    2.0
dtype: float64
```

Regarding the dataframe instead, the values can be deleted by referring to the labels of both axes. Declare the following frame by way of example.

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
```

```
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

To delete rows, you just pass the indexes of the rows.

```
>>> frame.drop(['blue','yellow'])
       ball  pen  pencil  paper
red       0    1       2      3
white    12   13      14     15
```

To delete columns, you always need to specify the indexes of the columns, but you must specify the axis from which to delete the elements, and this can be done using the `axis` option. So to refer to the column names, you should specify `axis = 1`.

```
>>> frame.drop(['pen','pencil'],axis=1)
        ball  paper
red        0      3
blue       4      7
yellow     8     11
white     12     15
```

Arithmetic and Data Alignment

Perhaps the most powerful feature involving the indexes in a data structure, is that pandas can align indexes coming from two different data structures. This is especially true when you are performing an arithmetic operation on them. In fact, during these operations, not only can the indexes between the two structures be in a different order, but they also can be present in only one of the two structures.

As you can see from the examples that follow, pandas proves to be very powerful in aligning indexes during these operations. For example, you can start considering two series in which they are defined, respectively, two arrays of labels not perfectly matching each other.

```
>>> s1 = pd.Series([3,2,5,1],['white','yellow','green','blue'])
>>> s2 = pd.Series([1,4,7,2,1],['white','yellow','black','blue','brown'])
```

Now among the various arithmetic operations, consider the simple sum. As you can see from the two series just declared, some labels are present in both, while other labels are present only in one of the two. When the labels are present in both operators, their values will be added, while in the opposite case, they will also be shown in the result (new series), but with the value NaN.

```
>>> s1 + s2
black    NaN
blue     3.0
brown    NaN
green    NaN
white    4.0
yellow   6.0
dtype: float64
```

In the case of the dataframe, although it may appear more complex, the alignment follows the same principle, but is carried out both for the rows and for the columns.

```
>>> frame1 = pd.DataFrame(np.arange(16).reshape((4,4)),
...                   index=['red','blue','yellow','white'],
...                   columns=['ball','pen','pencil','paper'])
>>> frame2 = pd.DataFrame(np.arange(12).reshape((4,3)),
...                   index=['blue','green','white','yellow'],
...                   columns=['mug','pen','ball'])
>>> frame1
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame2
        mug  pen  ball
blue      0    1     2
green     3    4     5
white     6    7     8
yellow    9   10    11
```

```
>>> frame1 + frame2
        ball  mug  paper  pen  pencil
blue     6.0  NaN    NaN  6.0     NaN
green    NaN  NaN    NaN  NaN     NaN
red      NaN  NaN    NaN  NaN     NaN
white    20.0 NaN    NaN  20.0    NaN
yellow   19.0 NaN    NaN  19.0    NaN
```

Operations Between Data Structures

Now that you are familiar with the data structures such as series and dataframe and you have seen how various elementary operations can be performed on them, it's time to go to operations involving two or more of these structures.

For example, in the previous section, you saw how the arithmetic operators apply between two of these objects. Now in this section you will deepen more the topic of operations that can be performed between two data structures.

Flexible Arithmetic Methods

You've just seen how to use mathematical operators directly on the pandas data structures. The same operations can also be performed using appropriate methods, called *flexible arithmetic methods*.

- `add()`
- `sub()`
- `div()`
- `mul()`

In order to call these functions, you need to use a specification different than what you're used to dealing with mathematical operators. For example, instead of writing a sum between two dataframes, such as `frame1 + frame2`, you have to use the following format:

```
>>> frame1.add(frame2)
         ball  mug  paper   pen  pencil
blue      6.0  NaN    NaN   6.0     NaN
green     NaN  NaN    NaN   NaN     NaN
```

```
red       NaN  NaN    NaN   NaN    NaN
white    20.0  NaN    NaN  20.0    NaN
yellow   19.0  NaN    NaN  19.0    NaN
```

As you can see, the results are the same as what you'd get using the addition operator +. You can also note that if the indexes and column names differ greatly from one series to another, you'll find yourself with a new dataframe full of NaN values. You'll see later in this chapter how to handle this kind of data.

Operations Between DataFrame and Series

Coming back to the arithmetic operators, pandas allows you to make transactions between different structures. For example, between a dataframe and a series. For example, you can define these two structures in the following way.

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> ser = pd.Series(np.arange(4), index=['ball','pen','pencil','paper'])
>>> ser
ball      0
pen       1
pencil    2
paper     3
dtype: int64
```

The two newly defined data structures have been created specifically so that the indexes of series match the names of the columns of the dataframe. This way, you can apply a direct operation.

```
>>> frame - ser
        ball  pen  pencil  paper
red        0    0       0      0
blue       4    4       4      4
yellow     8    8       8      8
white     12   12      12     12
```

As you can see, the elements of the series are subtracted from the values of the dataframe corresponding to the same index on the column. The value is subtracted for all values of the column, regardless of their index.

If an index is not present in one of the two data structures, the result will be a new column with that index only that all its elements will be NaN.

```
>>> ser['mug'] = 9
>>> ser
ball      0
pen       1
pencil    2
paper     3
mug       9
dtype: int64
>>> frame - ser
        ball  mug  paper  pen  pencil
red        0  NaN      0    0       0
blue       4  NaN      4    4       4
yellow     8  NaN      8    8       8
white     12  NaN     12   12      12
```

Function Application and Mapping

This section covers the pandas library functions.

Functions by Element

The pandas library is built on the foundations of NumPy and then extends many of its features by adapting them to new data structures as series and dataframe. Among these are the *universal functions*, called `ufunc`. This class of functions operates by element in the data structure.

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                   index=['red','blue','yellow','white'],
...                   columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

For example, you could calculate the square root of each value in the dataframe using the NumPy `np.sqrt()`.

```
>>> np.sqrt(frame)
            ball       pen    pencil     paper
red     0.000000  1.000000  1.414214  1.732051
blue    2.000000  2.236068  2.449490  2.645751
yellow  2.828427  3.000000  3.162278  3.316625
white   3.464102  3.605551  3.741657  3.872983
```

Functions by Row or Column

The application of the functions is not limited to the `ufunc` functions, but also includes those defined by the user. The important point is that they operate on a one-dimensional array, giving a single number as a result. For example, you can define a lambda function that calculates the range covered by the elements in an array.

```
>>> f = lambda x: x.max() - x.min()
```

It is possible to define the function this way as well:

```
>>> def f(x):
...     return x.max() - x.min()
...
```

Using the `apply()` function, you can apply the function just defined on the dataframe.

```
>>> frame.apply(f)
ball      12
pen       12
pencil    12
paper     12
dtype: int64
```

The result this time is one value for the column, but if you prefer to apply the function by row instead of by column, you have to set the `axis` option to 1.

```
>>> frame.apply(f, axis=1)
red       3
blue      3
yellow    3
white     3
dtype: int64
```

It is not mandatory that the method `apply()` return a scalar value. It can also return a series. A useful case would be to extend the application to many functions simultaneously. In this case, we will have two or more values for each feature applied. This can be done by defining a function in the following manner:

```
>>> def f(x):
...      return pd.Series([x.min(), x.max()], index=['min','max'])
...
```

Then, you apply the function as before. But in this case as an object returned you get a dataframe instead of a series, in which there will be as many rows as the values returned by the function.

```
>>> frame.apply(f)
     ball  pen  pencil  paper
min     0    1       2      3
max    12   13      14     15
```

Statistics Functions

Most of the statistical functions for arrays are still valid for dataframe, so using the `apply()` function is no longer necessary. For example, functions such as `sum()` and `mean()` can calculate the sum and the average, respectively, of the elements contained within a dataframe.

```
>>> frame.sum()
ball      24
pen       28
pencil    32
paper     36
dtype: int64
>>> frame.mean()
ball      6.0
pen       7.0
pencil    8.0
paper     9.0
dtype: float64
```

There is also a function called `describe()` that allows you to obtain summary statistics at once.

```
>>> frame.describe()
            ball        pen     pencil      paper
count   4.000000   4.000000   4.000000   4.000000
mean    6.000000   7.000000   8.000000   9.000000
std     5.163978   5.163978   5.163978   5.163978
min     0.000000   1.000000   2.000000   3.000000
25%     3.000000   4.000000   5.000000   6.000000
50%     6.000000   7.000000   8.000000   9.000000
75%     9.000000  10.000000  11.000000  12.000000
max    12.000000  13.000000  14.000000  15.000000
```

Sorting and Ranking

Another fundamental operation that uses indexing is sorting. Sorting the data is often a necessity and it is very important to be able to do it easily. pandas provides the `sort_index()` function, which returns a new object that's identical to the start, but in which the elements are ordered.

Let's start by seeing how you can sort items in a series. The operation is quite trivial since the list of indexes to be ordered is only one.

```
>>> ser = pd.Series([5,0,3,8,4],
...     index=['red','blue','yellow','white','green'])
>>> ser
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64
>>> ser.sort_index()
blue      0
green     4
red       5
white     8
yellow    3
dtype: int64
```

As you can see, the items were sorted in ascending alphabetical order based on their labels (from A to Z). This is the default behavior, but you can set the opposite order by setting the `ascending` option to `False`.

```
>>> ser.sort_index(ascending=False)
yellow    3
white     8
red       5
green     4
blue      0
dtype: int64
```

With the dataframe, the sorting can be performed independently on each of its two axes. So if you want to order by row following the indexes, you just continue to use the `sort_index()` function without arguments as you've seen before, or if you prefer to order by columns, you need to set the `axis` options to 1.

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame.sort_index()
        ball  pen  pencil  paper
blue       4    5       6      7
red        0    1       2      3
white     12   13      14     15
yellow     8    9      10     11
>>> frame.sort_index(axis=1)
        ball  paper  pen  pencil
red        0      3    1       2
blue       4      7    5       6
yellow     8     11    9      10
white     12     15   13      14
```

So far, you have learned how to sort the values according to the indexes. But very often you may need to sort the values contained in the data structure. In this case, you have to differentiate depending on whether you have to sort the values of a series or a dataframe.

If you want to order the series, you need to use the `sort_values()` function.

```
>>> ser.sort_values()
blue      0
yellow    3
green     4
```

```
red       5
white     8
dtype: int64
```

If you need to order the values in a dataframe, use the `sort_values()` function seen previously but with the by option. Then you have to specify the name of the column on which to sort.

```
>>> frame.sort_values(by='pen')
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

If the sorting criteria will be based on two or more columns, you can assign an array containing the names of the columns to the by option.

```
>>> frame.sort_values(by=['pen','pencil'])
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

The ranking is an operation closely related to sorting. It mainly consists of assigning a rank (that is, a value that starts at 0 and then increase gradually) to each element of the series. The rank will be assigned starting from the lowest value to the highest.

```
>>> ser.rank()
red       4.0
blue      1.0
yellow    2.0
white     5.0
green     3.0
dtype: float64
```

The rank can also be assigned in the order in which the data are already in the data structure (without a sorting operation). In this case, you just add the `method` option with the `first` value assigned.

```
>>> ser.rank(method='first')
red       4.0
blue      1.0
yellow    2.0
white     5.0
green     3.0
dtype: float64
```

By default, even the ranking follows an ascending sort. To reverse this criteria, set the `ascending` option to `False`.

```
>>> ser.rank(ascending=False)
red       2.0
blue      5.0
yellow    4.0
white     1.0
green     3.0
dtype: float64
```

Correlation and Covariance

Two important statistical calculations are correlation and covariance, expressed in pandas by the `corr()` and `cov()` functions. These kind of calculations normally involve two series.

```
>>> seq2 = pd.Series([3,4,3,4,5,4,3,2],['2006','2007','2008',
'2009','2010','2011','2012','2013'])
>>> seq = pd.Series([1,2,3,4,4,3,2,1],['2006','2007','2008',
'2009','2010','2011','2012','2013'])
>>> seq.corr(seq2)
0.7745966692414835
>>> seq.cov(seq2)
0.8571428571428571
```

Covariance and correlation can also be applied to a single dataframe. In this case, they return their corresponding matrices in the form of two new dataframe objects.

```
>>> frame2 = pd.DataFrame([[1,4,3,6],[4,5,6,1],[3,3,1,5],[4,1,6,4]],
...                       index=['red','blue','yellow','white'],
...                       columns=['ball','pen','pencil','paper'])
>>> frame2
        ball  pen  pencil  paper
red        1    4       3      6
blue       4    5       6      1
yellow     3    3       1      5
white      4    1       6      4
>>> frame2.corr()
            ball       pen    pencil     paper
ball    1.000000 -0.276026  0.577350 -0.763763
pen    -0.276026  1.000000 -0.079682 -0.361403
pencil  0.577350 -0.079682  1.000000 -0.692935
paper  -0.763763 -0.361403 -0.692935  1.000000
>>> frame2.cov()
            ball       pen    pencil     paper
ball    2.000000 -0.666667  2.000000 -2.333333
pen    -0.666667  2.916667 -0.333333 -1.333333
pencil  2.000000 -0.333333  6.000000 -3.666667
paper  -2.333333 -1.333333 -3.666667  4.666667
```

Using the `corrwith()` method, you can calculate the pairwise correlations between the columns or rows of a dataframe with a series or another `DataFrame()`.

```
>>> ser = pd.Series([0,1,2,3,9],
...                 index=['red','blue','yellow','white','green'])
>>> ser
red       0
blue      1
yellow    2
white     3
green     9
dtype: int64
```

```
>>> frame2.corrwith(ser)
ball      0.730297
pen      -0.831522
pencil    0.210819
paper    -0.119523
dtype: float64
>>> frame2.corrwith(frame)
ball      0.730297
pen      -0.831522
pencil    0.210819
paper    -0.119523
dtype: float64
```

"Not a Number" Data

In the previous sections, you saw how easily missing data can be formed. They are recognizable in the data structures by the `NaN` (Not a Number) value. So, having values that are not defined in a data structure is quite common in data analysis.

However, pandas is designed to better manage this eventuality. In fact, in this section, you will learn how to treat these values so that many issues can be obviated. For example, in the pandas library, calculating descriptive statistics excludes `NaN` values implicitly.

Assigning a NaN Value

If you need to specifically assign a `NaN` value to an element in a data structure, you can use the `np.NaN` (or `np.nan`) value of the NumPy library.

```
>>> ser = pd.Series([0,1,2,np.NaN,9],
...                 index=['red','blue','yellow','white','green'])
>>> ser
red      0.0
blue     1.0
yellow   2.0
white    NaN
green    9.0
```

```
dtype: float64
>>> ser['white'] = None
>>> ser
red      0.0
blue     1.0
yellow   2.0
white    NaN
green    9.0
dtype: float64
```

Filtering Out NaN Values

There are various ways to eliminate the NaN values during data analysis. Eliminating them by hand, element by element, can be very tedious and risky, and you're never sure that you eliminated all the NaN values. This is where the dropna() function comes to your aid.

```
>>> ser.dropna()
red     0.0
blue    1.0
yellow  2.0
green   9.0
dtype: float64
```

You can also directly perform the filtering function by placing notnull() in the selection condition.

```
>>> ser[ser.notnull()]
red     0.0
blue    1.0
yellow  2.0
green   9.0
dtype: float64
```

If you're dealing with a dataframe, it gets a little more complex. If you use the dropna() function on this type of object, and there is only one NaN value on a column or row, it will eliminate it.

```
>>> frame3 = pd.DataFrame([[6,np.nan,6],[np.nan,np.nan,np.nan],[2,np.nan,5]],
...                       index = ['blue','green','red'],
...                       columns = ['ball','mug','pen'])
>>> frame3
       ball  mug  pen
blue    6.0  NaN  6.0
green   NaN  NaN  NaN
red     2.0  NaN  5.0
>>> frame3.dropna()
Empty DataFrame
Columns: [ball, mug, pen]
Index: []
```

Therefore, to avoid having entire rows and columns disappear completely, you should specify the `how` option, assigning a value of `all` to it. This tells the `dropna()` function to delete only the rows or columns in which *all* elements are `NaN`.

```
>>> frame3.dropna(how='all')
      ball  mug  pen
blue   6.0  NaN  6.0
red    2.0  NaN  5.0
```

Filling in NaN Occurrences

Rather than filter `NaN` values within data structures, with the risk of discarding them along with values that could be relevant in the context of data analysis, you can replace them with other numbers. For most purposes, the `fillna()` function is a great choice. This method takes one argument, the value with which to replace any `NaN`. It can be the same for all cases.

```
>>> frame3.fillna(0)
       ball  mug  pen
blue    6.0  0.0  6.0
green   0.0  0.0  0.0
red     2.0  0.0  5.0
```

Or you can replace NaN with different values depending on the column, specifying one by one the indexes and the associated values.

```
>>> frame3.fillna({'ball':1,'mug':0,'pen':99})
       ball  mug   pen
blue    6.0  0.0   6.0
green   1.0  0.0  99.0
red     2.0  0.0   5.0
```

Hierarchical Indexing and Leveling

Hierarchical indexing is a very important feature of pandas, as it allows you to have multiple levels of indexes on a single axis. It gives you a way to work with data in multiple dimensions while continuing to work in a two-dimensional structure.

Let's start with a simple example, creating a series containing two arrays of indexes, that is, creating a structure with two levels.

```
>>> mser = pd.Series(np.random.rand(8),
...        index=[['white','white','white','blue','blue','red','red',
           'red'],
...               ['up','down','right','up','down','up','down','left']])
>>> mser
white  up       0.461689
       down     0.643121
       right    0.956163
blue   up       0.728021
       down     0.813079
red    up       0.536433
       down     0.606161
       left     0.996686
dtype: float64
```

```
>>> mser.index
Pd.MultiIndex(levels=[['blue', 'red', 'white'], ['down',
'left', 'right', 'up']],
...        labels=[[2, 2, 2, 0, 0, 1, 1, 1],
           [3, 0, 2, 3, 0, 3, 0, 1]])
```

Through the specification of hierarchical indexing, selecting subsets of values is in a certain way simplified.

In fact, you can select the values for a given value of the first index, and you do it in the classic way:

```
>>> mser['white']
up       0.461689
down     0.643121
right    0.956163
dtype: float64
```

Or you can select values for a given value of the second index, in the following manner:

```
>>> mser[:,'up']
white    0.461689
blue     0.728021
red      0.536433
dtype: float64
```

Intuitively, if you want to select a specific value, you specify both indexes.

```
>>> mser['white','up']
0.46168915430531676
```

Hierarchical indexing plays a critical role in reshaping data and group-based operations such as a pivot-table. For example, the data could be rearranged and used in a dataframe with a special function called `unstack()`. This function converts the series with a hierarchical index to a simple dataframe, where the second set of indexes is converted into a new set of columns.

```
>>> mser.unstack()
          down      left     right        up
blue   0.813079       NaN       NaN  0.728021
red    0.606161  0.996686       NaN  0.536433
white  0.643121       NaN  0.956163  0.461689
```

If what you want is to perform the reverse operation, which is to convert a dataframe to a series, you use the **stack()** function.

```
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame.stack()
red     ball       0
        pen        1
        pencil     2
        paper      3
blue    ball       4
        pen        5
        pencil     6
        paper      7
yellow  ball       8
        pen        9
        pencil    10
        paper     11
white   ball      12
        pen       13
        pencil    14
        paper     15
dtype: int64
```

With dataframe, it is possible to define a hierarchical index both for the rows and for the columns. At the time the dataframe is declared, you have to define an array of arrays for the `index` and `columns` options.

```
>>> mframe = pd.DataFrame(np.random.randn(16).reshape(4,4),
...      index=[['white','white','red','red'], ['up','down','up','down']],
...      columns=[['pen','pen','paper','paper'],[1,2,1,2]])
```

```
>>> mframe
                 pen                paper
                   1         2         1         2
white up   -1.964055  1.312100 -0.914750 -0.941930
      down -1.886825  1.700858 -1.060846 -0.197669
red   up   -1.561761  1.225509 -0.244772  0.345843
      down  2.668155  0.528971 -1.633708  0.921735
```

Reordering and Sorting Levels

Occasionally, you might need to rearrange the order of the levels on an axis or sort for values at a specific level.

The `swaplevel()` function accepts as arguments the names assigned to the two levels that you want to interchange and returns a new object with the two levels interchanged between them, while leaving the data unmodified.

```
>>> mframe.columns.names = ['objects','id']
>>> mframe.index.names = ['colors','status']
>>> mframe
objects             pen                paper
id                    1         2         1         2
colors status
white  up     -1.964055  1.312100 -0.914750 -0.941930
       down   -1.886825  1.700858 -1.060846 -0.197669
red    up     -1.561761  1.225509 -0.244772  0.345843
       down    2.668155  0.528971 -1.633708  0.921735

>>> mframe.swaplevel('colors','status')
objects             pen                paper
id                    1         2         1         2
status colors
up     white  -1.964055  1.312100 -0.914750 -0.941930
down   white  -1.886825  1.700858 -1.060846 -0.197669
up     red    -1.561761  1.225509 -0.244772  0.345843
down   red     2.668155  0.528971 -1.633708  0.921735
```

Instead, the **sort_index()** function orders the data considering only those of a certain level by specifying it as parameter

```
>>> mframe.sort_index(level='colors')
objects             pen               paper
id                    1         2         1         2
colors status
red    down    2.668155  0.528971 -1.633708  0.921735
       up     -1.561761  1.225509 -0.244772  0.345843
white  down   -1.886825  1.700858 -1.060846 -0.197669
       up     -1.964055  1.312100 -0.914750 -0.941930
```

Summary Statistic by Level

Many descriptive statistics and summary statistics performed on a dataframe or on a series have a `level` option, with which you can determine at what level the descriptive and summary statistics should be determined.

For example, if you create a statistic at row level, you have to simply specify the `level` option with the level name.

```
 >>> mframe.sum(level='colors')
objects       pen               paper
id              1         2         1         2
colors
red      1.106394  1.754480 -1.878480  1.267578
white   -3.850881  3.012959 -1.975596 -1.139599
```

If you want to create a statistic for a given level of the column, for example, the `id`, you must specify the second axis as an argument through the `axis` option set to 1.

```
>>> mframe.sum(level='id', axis=1)
id                    1         2
colors status
white  up     -2.878806  0.370170
       down   -2.947672  1.503189
red    up     -1.806532  1.571352
       down    1.034447  1.450706
```

Conclusions

This chapter introduced the pandas library. You learned how to install it and saw a general overview of its characteristics.

You learned about the two basic structures data, called the series and dataframe, along with their operation and their main characteristics. Especially, you discovered the importance of indexing within these structures and how best to perform operations on them. Finally, you looked at the possibility of extending the complexity of these structures by creating hierarchies of indexes, thus distributing the data contained in them into different sublevels.

In the next chapter, you learn how to capture data from external sources such as files, and inversely, how to write the analysis results on them.

CHAPTER 5

pandas: Reading and Writing Data

In the previous chapter, you became familiar with the pandas library and with the basic functionalities that it provides for data analysis. You saw that dataframe and series are the heart of this library. These are the material on which to perform all data manipulations, calculations, and analysis.

In this chapter, you will see all of the tools provided by pandas for reading data stored in many types of media (such as files and databases). In parallel, you will also see how to write data structures directly on these formats, without worrying too much about the technologies used.

This chapter focuses on a series of I/O API functions that pandas provides to read and write data directly as dataframe objects. We start by looking at text files, then move gradually to more complex binary formats.

At the end of the chapter, you'll also learn how to interface with all common databases, both SQL and NoSQL, including examples that show how to store data in a dataframe. At the same time, you learn how to read data contained in a database and retrieve them as a dataframe.

I/O API Tools

pandas is a library specialized for data analysis, so you expect that it is mainly focused on calculation and data processing. The processes of writing and reading data from/to external files can be considered part of data processing. In fact, you will see how, even at this stage, you can perform some operations in order to prepare the incoming data for manipulation.

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_5

Thus, this step is very important for data analysis and therefore a specific tool for this purpose must be present in the library pandas—a set of functions called I/O API. These functions are divided into two main categories: *readers* and *writers*.

Readers	Writers
`read_csv`	`to_csv`
`read_excel`	`to_excel`
`read_hdf`	`to_hdf`
`read_sql`	`to_sql`
`read_json`	`to_json`
`read_html`	`to_html`
`read_stata`	`to_stata`
`read_clipboard`	`to_clipboard`
`read_pickle`	`to_pickle`
`read_msgpack`	`to_msgpack` (experimental)
`read_gbq`	`to_gbq` (experimental)

CSV and Textual Files

Everyone has become accustomed over the years to writing and reading files in text form. In particular, data are generally reported in tabular form. If the values in a row are separated by commas, you have the CSV (comma-separated values) format, which is perhaps the best-known and most popular format.

Other forms of tabular data can be separated by spaces or tabs and are typically contained in text files of various types (generally with the `.txt` extension).

This type of file is the most common source of data and is easier to transcribe and interpret. In this regard, pandas provides a set of functions specific for this type of file.

- `read_csv`
- `read_table`
- `to_csv`

Reading Data in CSV or Text Files

From experience, the most common operation of a person approaching data analysis is to read the data contained in a CSV file, or at least in a text file.

But before you start dealing with files, you need to import the following libraries.

```
>>> import numpy as np
>>> import pandas as pd
```

In order to see how pandas handles this kind of data, we'll start by creating a small CSV file in the working directory, as shown in Listing 5-1, and save it as `ch05_01.csv`.

Listing 5-1. ch05_01.csv

```
white,red,blue,green,animal
1,5,2,3,cat
2,7,8,5,dog
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse
```

Since this file is comma-delimited, you can use the `read_csv()` function to read its content and convert it to a dataframe object.

```
>>> csvframe = pd.read_csv('ch05_01.csv')
>>> csvframe
   white  red  blue  green animal
0      1    5     2      3    cat
1      2    7     8      5    dog
2      3    3     6      7  horse
3      2    2     8      3   duck
4      4    4     2      1  mouse
```

As you can see, reading the data in a CSV file is rather trivial. CSV files are tabulated data in which the values on the same column are separated by commas. Since CSV files are considered text files, you can also use the `read_table()` function, but specify the delimiter.

```
>>> pd.read_table('ch05_01.csv',sep=',')
   white  red  blue  green animal
0      1    5     2      3    cat
1      2    7     8      5    dog
2      3    3     6      7  horse
3      2    2     8      3   duck
4      4    4     2      1  mouse
```

In this example, you can see that in the CSV file, headers that identify all the columns are in the first row. But this is not a general case; it often happens that the tabulated data begin directly in the first line (see Listing 5-2).

Listing 5-2. ch05_02.csv

```
1,5,2,3,cat
2,7,8,5,dog
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse
```

```
>>> pd.read_csv('ch05_02.csv')
   1  5  2  3    cat
0  2  7  8  5    dog
1  3  3  6  7  horse
2  2  2  8  3   duck
3  4  4  2  1  mouse
```

In this case, you could make sure that it is pandas that assigns the default names to the columns by setting the `header` option to `None`.

```
>>> pd.read_csv('ch05_02.csv', header=None)
   0  1  2  3      4
0  1  5  2  3    cat
1  2  7  8  5    dog
2  3  3  6  7  horse
3  2  2  8  3   duck
4  4  4  2  1  mouse
```

In addition, you can specify the names directly by assigning a list of labels to the `names` option.

```
>>> pd.read_csv('ch05_02.csv', names=['white','red','blue','green','animal'])
   white  red  blue  green animal
0      1    5     2      3    cat
1      2    7     8      5    dog
2      3    3     6      7  horse
3      2    2     8      3   duck
4      4    4     2      1  mouse
```

In more complex cases, in which you want to create a dataframe with a hierarchical structure by reading a CSV file, you can extend the functionality of the `read_csv()` function by adding the `index_col` option, assigning all the columns to be converted into indexes.

To better understand this possibility, create a new CSV file with two columns to be used as indexes of the hierarchy. Then, save it in the working directory as `ch05_03.csv` (see Listing 5-3).

Listing 5-3. ch05_03.csv

```
color,status,item1,item2,item3
black,up,3,4,6
black,down,2,6,7
white,up,5,5,5
white,down,3,3,2
white,left,1,2,1
red,up,2,2,2
red,down,1,1,4
```

```
>>> pd.read_csv('ch05_03.csv', index_col=['color','status'])
              item1  item2  item3
color status
black up          3      4      6
      down        2      6      7
white up          5      5      5
      down        3      3      2
      left        1      2      1
red   up          2      2      2
      down        1      1      4
```

Using RegExp to Parse TXT Files

In other cases, it is possible that the files on which to parse the data do not show separators well defined as a comma or a semicolon. In these cases, the regular expressions come to our aid. In fact, you can specify a regexp within the `read_table()` function using the `sep` option.

To better understand regexp and understand how you can apply it as criteria for value separation, let's start with a simple case. For example, suppose that your TXT file has values that are separated by spaces or tabs in an unpredictable order. In this case, you have to use the regexp, because that's the only way to take into account both separator types. You can do that using the wildcard `/s*`. `/s` stands for the space or tab character (if you want to indicate a tab, you use `/t`), while the asterisk indicates that there may be multiple characters (see Table 5-1 for other common wildcards). That is, the values may be separated by more spaces or more tabs.

Table 5-1. *Metacharacters*

`.`	Single character, except newline
`\d`	Digit
`\D`	Non-digit character
`\s`	Whitespace character
`\S`	Non-whitespace character
`\n`	New line character
`\t`	Tab character
`\uxxxx`	Unicode character specified by the hexadecimal number `xxxx`

Take for example an extreme case in which we have the values separated by tabs or spaces in a random order (see Listing 5-4).

Listing 5-4. ch05_04.txt

```
white red blue green
    1   5    2     3
    2   7    8     5
    3   3    6     7
```

```
>>> pd.read_table('ch05_04.txt',sep='\s+', engine='python')
   white  red  blue  green
0      1    5     2      3
1      2    7     8      5
2      3    3     6      7
```

As you can see, the result is a perfect dataframe in which the values are perfectly ordered.

Now you will see an example that may seem strange or unusual, but it is not as rare as it may seem. This example can be very helpful in understanding the high potential of a regexp. In fact, you might typically think of separators as special characters like commas, spaces, tabs, etc., but in reality you can consider separator characters like alphanumeric characters, or for example, integers such as 0.

In this example, you need to extract the numeric part from a TXT file, in which there is a sequence of characters with numerical values and the literal characters are completely fused.

Remember to set the `header` option to `None` whenever the column headings are not present in the TXT file (see Listing 5-5).

Listing 5-5. ch05_05.txt

```
000END123AAA122
001END124BBB321
002END125CCC333
```

```
>>> pd.read_table('ch05_05.txt', sep='\D+', header=None, engine='python')
   0    1    2
0  0  123  122
1  1  124  321
2  2  125  333
```

Another fairly common event is to exclude lines from parsing. In fact you do not always want to include headers or unnecessary comments contained in a file (see Listing 5-6). With the `skiprows` option, you can exclude all the lines you want, just assigning an array containing the line numbers to not consider in parsing.

Pay attention when you are using this option. If you want to exclude the first five lines, you have to write `skiprows = 5`, but if you want to rule out the fifth line, you have to write `skiprows = [5]`.

Listing 5-6. ch05_06.txt

```
########### LOG FILE ############
This file has been generated by automatic system
white,red,blue,green,animal
12-Feb-2015: Counting of animals inside the house
1,5,2,3,cat
2,7,8,5,dog
13-Feb-2015: Counting of animals outside the house
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse
```

```
>>> pd.read_table('ch05_06.txt',sep=',',skiprows=[0,1,3,6])
   white  red  blue  green animal
0      1    5     2      3    cat
1      2    7     8      5    dog
2      3    3     6      7  horse
3      2    2     8      3   duck
4      4    4     2      1  mouse
```

Reading TXT Files Into Parts

When large files are processed, or when you are only interested in portions of these files, you often need to read the file into portions (chunks). This is both to apply any iterations and because we are not interested in parsing the entire file.

If, for example, you wanted to read only a portion of the file, you can explicitly specify the number of lines on which to parse. Thanks to the `nrows` and `skiprows` options, you can select the starting line `n` (`n = SkipRows`) and the lines to be read after it (`nrows = i`).

```
>>> pd.read_csv('ch05_02.csv',skiprows=[2],nrows=3,header=None)
   0  1  2  3     4
0  1  5  2  3   cat
1  2  7  8  5   dog
2  2  2  8  3  duck
```

Another interesting and fairly common operation is to split into portions that part of the text on which you want to parse. Then, for each portion a specific operation may be carried out, in order to obtain an iteration, portion by portion.

For example, you want to add the values in a column every three rows and then insert these sums in a series. This example is trivial and impractical but is very simple to understand, so once you have learned the underlying mechanism, you will be able to apply it in more complex cases.

```
>>> out = pd.Series()
>>> i = 0
>>> pieces = pd.read_csv('ch05_01.csv',chunksize=3)
>>> for piece in pieces:
...    out.set_value(i,piece['white'].sum())
...    i = i + 1
...
0    6
dtype: int64
0    6
1    6
dtype: int64
>>> out
0    6
1    6
dtype: int64
```

Writing Data in CSV

In addition to reading the data contained in a file, it's also common to write a data file produced by a calculation, or in general the data contained in a data structure.

For example, you might want to write the data contained in a dataframe to a CSV file. To do this writing process, you will use the `to_csv()` function, which accepts as an argument the name of the file you generate (see Listing 5-7).

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
                index = ['red', 'blue', 'yellow', 'white'],
                columns = ['ball', 'pen', 'pencil', 'paper'])

>>> frame.to_csv('ch05_07.csv')
```

If you open the new file called `ch05_07.csv` generated by the pandas library, you will see data as in Listing 5-7.

Listing 5-7. ch05_07.csv

```
,ball,pen,pencil,paper
0,1,2,3
4,5,6,7
8,9,10,11
12,13,14,15
```

As you can see from the previous example, when you write a dataframe to a file, indexes and columns are marked on the file by default. This default behavior can be changed by setting the two options `index` and `header` to `False` (see Listing 5-8).

```
>>> frame.to_csv('ch05_07b.csv', index=False, header=False)
```

Listing 5-8. ch05_07b.csv

```
1,2,3
5,6,7
9,10,11
13,14,15
```

One point to remember when writing files is that NaN values present in a data structure are shown as empty fields in the file (see Listing 5-9).

```
>>> frame3 = pd.DataFrame([[6,np.nan,np.nan,6,np.nan],
...           [np.nan,np.nan,np.nan,np.nan,np.nan],
...           [np.nan,np.nan,np.nan,np.nan,np.nan],
...           [20,np.nan,np.nan,20.0,np.nan],
...           [19,np.nan,np.nan,19.0,np.nan]
...           ],
...                  index=['blue','green','red','white','yellow'],
                     columns=['ball','mug','paper','pen','pencil'])
>>> frame3
        ball  mug  paper  pen  pencil
blue     6.0  NaN    NaN  6.0     NaN
green    NaN  NaN    NaN  NaN     NaN
red      NaN  NaN    NaN  NaN     NaN
white   20.0  NaN    NaN 20.0     NaN
yellow  19.0  NaN    NaN 19.0     NaN
>>> frame3.to_csv('ch05_08.csv')
```

Listing 5-9. ch05_08.csv

```
,ball,mug,paper,pen,pencil
blue,6.0,,,6.0,
green,,,,,
red,,,,,
white,20.0,,,20.0,
yellow,19.0,,,19.0,
```

However, you can replace this empty field with a value to your liking using the na_rep option in the to_csv() function. Common values may be NULL, 0, or the same NaN (see Listing 5-10).

```
>>> frame3.to_csv('ch05_09.csv', na_rep ='NaN')
```

Listing 5-10. ch05_09.csv

```
,ball,mug,paper,pen,pencil
blue,6.0,NaN,NaN,6.0,NaN
green,NaN,NaN,NaN,NaN,NaN
red,NaN,NaN,NaN,NaN,NaN
white,20.0,NaN,NaN,20.0,NaN
yellow,19.0,NaN,NaN,19.0,NaN
```

Note In the cases specified, dataframe has always been the subject of discussion since these are the data structures that are written to the file. But all these functions and options are also valid with regard to the series.

Reading and Writing HTML Files

pandas provides the corresponding pair of I/O API functions for the HTML format.

- `read_html()`
- `to_html()`

These two functions can be very useful. You will appreciate the ability to convert complex data structures such as dataframes directly into HTML tables without having to hack a long listing in HTML, especially if you're dealing with the Web.

The inverse operation can be very useful, because now the major source of data is just the web world. In fact, a lot of data on the Internet does not always have the form "ready to use," that is packaged in some TXT or CSV file. Very often, however, the data are reported as part of the text of web pages. So also having available a function for reading could prove to be really useful.

This activity is so widespread that it is currently identified as *web scraping*. This process is becoming a fundamental part of the set of processes that will be integrated in the first part of data analysis: data mining and data preparation.

Note Many websites have now adopted the HTML5 format, to avoid any issues of missing modules and error messages. I strongly recommend you install the module `html5lib`. Anaconda specified:

```
conda install html5lib
```

Writing Data in HTML

Now you learn how to convert a dataframe into an HTML table. The internal structure of the dataframe is automatically converted into nested tags <TH>, <TR>, and <TD> retaining any internal hierarchies. You do not need to know HTML to use this kind of function.

Because the data structures as the dataframe can be quite complex and large, it's great to have a function like this when you need to develop web pages.

To better understand this potential, here's an example. You can start by defining a simple dataframe.

Thanks to the `to_html()` function, you can directly convert the dataframe into an HTML table.

```
>>> frame = pd.DataFrame(np.arange(4).reshape(2,2))
```

Since the I/O API functions are defined in the pandas data structures, you can call the `to_html()` function directly on the instance of the dataframe.

```
>>> print(frame.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
```

```
    <tr>
      <th>0</th>
      <td> 0</td>
      <td> 1</td>
    </tr>
    <tr>
      <th>1</th>
      <td> 2</td>
      <td> 3</td>
    </tr>
  </tbody>
</table>
```

As you can see, the whole structure formed by the HTML tags needed to create an HTML table was generated correctly in order to respect the internal structure of the dataframe.

In the next example, you'll see how the table appears automatically generated within an HTML file. In this regard, we create a dataframe a bit more complex than the previous one, where there are the labels of the indexes and column names.

```
>>> frame = pd.DataFrame( np.random.random((4,4)),
...                       index = ['white','black','red','blue'],
...                       columns = ['up','down','right','left'])
>>> frame
             up      down     right      left
white  0.292434  0.457176  0.905139  0.737622
black  0.794233  0.949371  0.540191  0.367835
red    0.204529  0.981573  0.118329  0.761552
blue   0.628790  0.585922  0.039153  0.461598
```

Now you focus on writing an HTML page through the generation of a string. This is a simple and trivial example, but it is very useful to understand and to test the functionality of pandas directly on the web browser.

First of all we create a string that contains the code of the HTML page.

```
>>> s = ['<HTML>']
>>> s.append('<HEAD><TITLE>My DataFrame</TITLE></HEAD>')
>>> s.append('<BODY>')
>>> s.append(frame.to_html())
>>> s.append('</BODY></HTML>')
>>> html = ''.join(s)
```

Now that all the listing of the HTML page is contained within the `html` variable, you can write directly on the file that will be called `myFrame.html`:

```
>>> html_file = open('myFrame.html','w')
>>> html_file.write(html)
>>> html_file.close()
```

Now in your working directory will be a new HTML file, `myFrame.html`. Double-click it to open it directly from the browser. An HTML table will appear in the upper left, as shown in Figure 5-1.

	up	down	right	left
white	0.292434	0.457176	0.905139	0.737622
black	0.794233	0.949371	0.540191	0.367835
red	0.204529	0.981573	0.118329	0.761552
blue	0.628790	0.585922	0.039153	0.461598

Figure 5-1. *The dataframe is shown as an HTML table in the web page*

Reading Data from an HTML File

As you just saw, pandas can easily generate HTML tables starting from the dataframe. The opposite process is also possible; the function `read_html ()` will perform a parsing an HTML page looking for an HTML table. If found, it will convert that table into an object dataframe ready to be used in our data analysis.

More precisely, the `read_html()` function returns a list of dataframes even if there is only one table. The source that will be parsed can be different types. For example, you may have to read an HTML file in any directory. For example you can parse the HTML file you created in the previous example:

```
>>> web_frames = pd.read_html('myFrame.html')
>>> web_frames[0]
  Unnamed: 0        up      down     right      left
0      white  0.292434  0.457176  0.905139  0.737622
1      black  0.794233  0.949371  0.540191  0.367835
2        red  0.204529  0.981573  0.118329  0.761552
3       blue  0.628790  0.585922  0.039153  0.461598
```

As you can see, all of the tags that have nothing to do with HTML table are not considered absolutely. Furthermore `web_frames` is a list of dataframes, although in your case, the dataframe that you are extracting is only one. However, you can select the item in the list that you want to use, calling it in the classic way. In this case, the item is unique and therefore the index will be 0.

However, the mode most commonly used regarding the `read_html()` function is that of a direct parsing of an URL on the Web. In this way the web pages in the network are directly parsed with the extraction of the tables in them.

For example, now you will call a web page where there is an HTML table that shows a ranking list with some names and scores.

```
>>> ranking = pd.read_html('https://www.meccanismocomplesso.org/en/
meccanismo-complesso-sito-2/classifica-punteggio/')
>>> ranking[0]
    Member        points  levels  Unnamed: 3
0        1   BrunoOrsini    1075         NaN
1        2     Berserker     700         NaN
2        3  albertosallu     275         NaN
3        4          Mr.Y     180         NaN
4        5           Jon     170         NaN
5        6  michele sisi     120         NaN
6        7  STEFANO GUST     120         NaN
7        8  Davide Alois     105         NaN
8        9  Cecilia Lala     105         NaN
...
```

The same operation can be run on any web page that has one or more tables.

Reading Data from XML

In the list of I/O API functions, there is no specific tool regarding the XML (Extensible Markup Language) format. In fact, although it is not listed, this format is very important, because many structured data are available in XML format. This presents no problem, since Python has many other libraries (besides pandas) that manage the reading and writing of data in XML format.

One of these libraries is the `lxml` library, which stands out for its excellent performance during the parsing of very large files. In this section you learn how to use this module for parsing XML files and how to integrate it with pandas to finally get the dataframe containing the requested data. For more information about this library, I highly recommend visiting the official website of lxml at `http://lxml.de/index.html`.

Take for example the XML file shown in Listing 5-11. Write down and save it with the name `books.xml` directly in your working directory.

Listing 5-11. books.xml

```
<?xml version="1.0"?>
<Catalog>
   <Book id="ISBN9872122367564">
      <Author>Ross, Mark</Author>
      <Title>XML Cookbook</Title>
      <Genre>Computer</Genre>
      <Price>23.56</Price>
      <PublishDate>2014-22-01</PublishDate>
   </Book>
   <Book id="ISBN9872122367564">
      <Author>Bracket, Barbara</Author>
      <Title>XML for Dummies</Title>
      <Genre>Computer</Genre>
      <Price>35.95</Price>
      <PublishDate>2014-12-16</PublishDate>
   </Book>
</Catalog>
```

In this example, you will take the data structure described in the XML file to convert it directly into a dataframe. The first thing to do is use the sub-module `objectify` of the `lxml` library, importing it in the following way.

```
>>> from lxml import objectify
```

Now you can do the parser of the XML file with just the `parse()` function.

```
>>> xml = objectify.parse('books.xml')
>>> xml
<lxml.etree._ElementTree object at 0x00000000009734E08>
```

You got an object tree, which is an internal data structure of the `lxml` module.

Look in more detail at this type of object. To navigate in this tree structure, so as to select element by element, you must first define the root. You can do this with the `getroot()` function.

```
>>> root = xml.getroot()
```

Now that the root of the structure has been defined, you can access the various nodes of the tree, each corresponding to the tag contained in the original XML file. The items will have the same name as the corresponding tags. So to select them, simply write the various separate tags with points, reflecting in a certain way the hierarchy of nodes in the tree.

```
>>> root.Book.Author
'Ross, Mark'
>>> root.Book.PublishDate
'2014-22-01'
```

In this way you access nodes individually, but you can access various elements at the same time using `getchildren()`. With this function, you'll get all the child nodes of the reference element.

```
>>> root.getchildren()
[<Element Book at 0x9c66688>, <Element Book at 0x9c66e08>]
```

With the `tag` attribute you get the name of the tag corresponding to the child node.

```
>>> [child.tag for child in root.Book.getchildren()]
['Author', 'Title', 'Genre', 'Price', 'PublishDate']
```

While with the `text` attribute you get the value contained between the corresponding tags.

```
>>> [child.text for child in root.Book.getchildren()]
['Ross, Mark', 'XML Cookbook', 'Computer', '23.56', '2014-22-01']
```

However, regardless of the ability to move through the `lxml.etree` tree structure, what you need is to convert it into a dataframe. Define the following function, which has the task of analyzing the contents of an eTree to fill a dataframe line by line.

```
>>> def etree2df(root):
...     column_names = []
...     for i in range(0,len(root.getchildren()[0].getchildren())):
...        column_names.append(root.getchildren()[0].getchildren()[i].tag)
...     xml:frame = pd.DataFrame(columns=column_names)
...     for j in range(0, len(root.getchildren())):
...        obj = root.getchildren()[j].getchildren()
...        texts = []
...        for k in range(0, len(column_names)):
...           texts.append(obj[k].text)
...        row = dict(zip(column_names, texts))
...        row_s = pd.Series(row)
...        row_s.name = j
...        xml:frame = xml:frame.append(row_s)
...     return xml:frame
...
>>> etree2df(root)
             Author            Title     Genre  Price PublishDate
0        Ross, Mark     XML Cookbook  Computer  23.56  2014-22-01
1  Bracket, Barbara  XML for Dummies  Computer  35.95  2014-12-16
```

Reading and Writing Data on Microsoft Excel Files

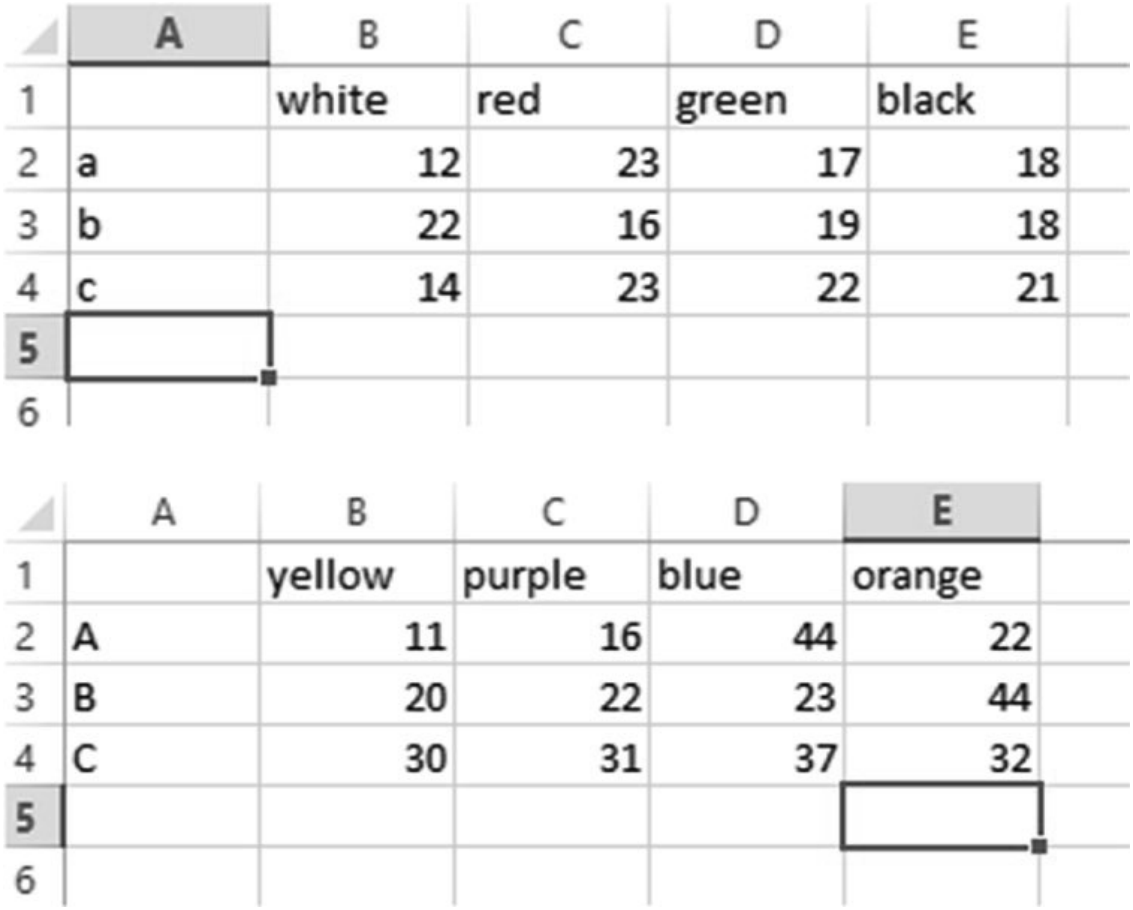
In the previous section, you saw how the data can be easily read from CSV files. It is not uncommon, however, that there are data collected in tabular form in an Excel spreadsheet.

pandas provides specific functions for this type of format. You have seen that the I/O API provides two functions to this purpose:

- `to_excel()`
- `read_excel()`

The `read_excel()` function can read Excel 2003 (.xls) files and Excel 2007 (.xlsx) files. This is possible thanks to the integration of the internal module `xlrd`.

First, open an Excel file and enter the data as shown in Figure 5-2. Copy the data in sheet1 and sheet2. Then save it as `ch05_data.xlsx`.



	A	B	C	D	E
1		white	red	green	black
2	a	12	23	17	18
3	b	22	16	19	18
4	c	14	23	22	21
5					
6					

	A	B	C	D	E
1		yellow	purple	blue	orange
2	A	11	16	44	22
3	B	20	22	23	44
4	C	30	31	37	32
5					
6					

Figure 5-2. *The two datasets in sheet1 and sheet2 of an Excel file*

To read the data contained in the XLS file and convert it into a dataframe, you only have to use the `read_excel()` function.

```
>>> pd.read_excel('ch05_data.xlsx')
   white  red  green  black
a     12   23     17     18
b     22   16     19     18
c     14   23     22     21
```

As you can see, by default, the returned dataframe is composed of the data tabulated in the first spreadsheets. If, however, you need to load the data in the second spreadsheet, you must then specify the name of the sheet or the number of the sheet (index) just as the second argument.

```
>>> pd.read_excel('ch05_data.xlsx','Sheet2')
   yellow  purple  blue  orange
A      11      16    44      22
B      20      22    23      44
C      30      31    37      32
>>> pd.read_excel('ch05_data.xlsx',1)
   yellow  purple  blue  orange
A      11      16    44      22
B      20      22    23      44
C      30      31    37      32
```

The same applies for writing. To convert a dataframe into a spreadsheet on Excel, you have to write the following.

```
>>> frame = pd.DataFrame(np.random.random((4,4)),
...                      index = ['exp1','exp2','exp3','exp4'],
...                      columns = ['Jan2015','Fab2015','Mar2015','Apr2005'])
>>> frame
       Jan2015   Fab2015   Mar2015   Apr2005
exp1  0.030083  0.065339  0.960494  0.510847
exp2  0.531885  0.706945  0.964943  0.085642
exp3  0.981325  0.868894  0.947871  0.387600
exp4  0.832527  0.357885  0.538138  0.357990
>>> frame.to_excel('data2.xlsx')
```

In the working directory, you will find a new Excel file containing the data, as shown in Figure 5-3.

	A	B	C	D	E
1		Jan2015	Fab2015	Mar2015	Apr2005
2	exp1	0,030083	0,065339	0,960494	0,510847
3	exp2	0,531885	0,706945	0,964943	0,085642
4	exp3	0,981325	0,868894	0,947871	0,3876
5	exp4	0,832527	0,357885	0,538138	0,35799
6					

Figure 5-3. *The dataframe in the Excel file*

JSON Data

JSON (JavaScript Object Notation) has become one of the most common standard formats, especially for the transmission of data on the Web. So it is normal to work with this data format if you want to use data on the Web.

The special feature of this format is its great flexibility, although its structure is far from being the one to which you are well accustomed, i.e., tabular.

In this section you will see how to use the `read_json()` and `to_json()` functions to stay within the I/O API functions discussed in this chapter. But in the second part you will see another example in which you will have to deal with structured data in JSON format much more related to real cases.

In my opinion, a useful online application for checking the JSON format is JSONViewer, available at `http://jsonviewer.stack.hu/`. This web application, once you enter or copy data in JSON format, allows you to see if the format you entered is valid. Moreover it displays the tree structure so that you can better understand its structure (see Figure 5-4).

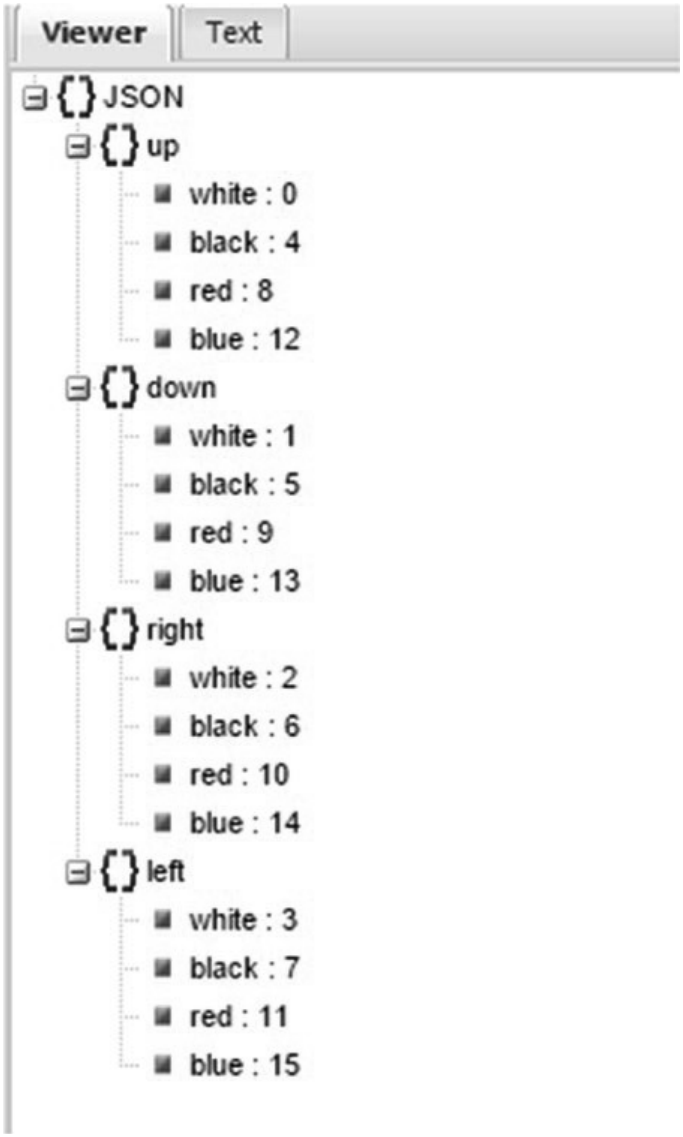


Figure 5-4. *JSONViewer*

Let's begin with the more useful case, that is, when you have a dataframe and you need to convert it into a JSON file. So, define a dataframe and then call the `to_json()` function on it, passing as an argument the name of the file that you want to create.

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4),
...                      index=['white','black','red','blue'],
...                      columns=['up','down','right','left'])
>>> frame.to_json('frame.json')
```

In the working directory, you will find a new JSON file (see Listing 5-12) containing the dataframe data translated into JSON format.

Listing 5-12. frame.json

```
{"up":{"white":0,"black":4,"red":8,"blue":12},"down":{"white":1,"black":5,
"red":9,"blue":13},"right":{"white":2,"black":6,"red":10,"blue":14},
"left":{"white":3,"black":7,"red":11,"blue":15}}
```

The converse is possible, using the `read_json()` with the name of the file passed as an argument.

```
>>> pd.read_json('frame.json')
       down  left  right  up
black     5     7      6   4
blue     13    15     14  12
red       9    11     10   8
white     1     3      2   0
```

The example you have seen is a fairly simple case in which the JSON data were in tabular form (since the file `frame.json` comes from a dataframe). Generally, however, the JSON files do not have a tabular structure. Thus, you will need to somehow convert the structure `dict` file into tabular form. This process is called *normalization*.

The library pandas provides a function, called `json_normalize()`, that is able to convert a `dict` or a list in a table. First you have to import the function:

```
>>> from pandas.io.json import json_normalize
```

Then you write a JSON file as described in Listing 5-13 with any text editor. Save it in the working directory as `books.json`.

Listing 5-13. books.json

```
[{"writer": "Mark Ross",
 "nationality": "USA",
 "books": [
         {"title": "XML Cookbook", "price": 23.56},
         {"title": "Python Fundamentals", "price": 50.70},
         {"title": "The NumPy library", "price": 12.30}
             ]
},
```

```
{"writer": "Barbara Bracket",
 "nationality": "UK",
 "books": [
         {"title": "Java Enterprise", "price": 28.60},
         {"title": "HTML5", "price": 31.35},
         {"title": "Python for Dummies", "price": 28.00}
             ]
}]
```

As you can see, the file structure is no longer tabular, but more complex. Then the approach with the `read_json()` function is no longer valid. As you learn from this example, you can still get the data in tabular form from this structure. First you have to load the contents of the JSON file and convert it into a string.

```
>>> import json
>>> file = open('books.json','r')
>>> text = file.read()
>>> text = json.loads(text)
```

Now you are ready to apply the `json_normalize()` function. From a quick look at the contents of the data within the JSON file, for example, you might want to extract a table that contains all the books. Then write the `books` key as the second argument.

```
>>> json_normalize(text,'books')
   price                title
0  23.56          XML Cookbook
1  50.70  Python Fundamentals
2  12.30    The NumPy library
3  28.60      Java Enterprise
4  31.35                HTML5
5  28.00   Python for Dummies
```

The function will read the contents of all the elements that have `books` as the key. All properties will be converted into nested column names while the corresponding values will fill the dataframe. For the indexes, the function assigns a sequence of increasing numbers.

However, you get a dataframe containing only some internal information. It would be useful to add the values of other keys on the same level. In this case you can add other columns by inserting a key list as the third argument of the function.

```
>>> json_normalize(text,'books',['nationality','writer'])
   price                title nationality           writer
0  23.56          XML Cookbook         USA        Mark Ross
1  50.70  Python Fundamentals          USA        Mark Ross
2  12.30    The NumPy library          USA        Mark Ross
3  28.60       Java Enterprise          UK  Barbara Bracket
4  31.35                 HTML5          UK  Barbara Bracket
5  28.00   Python for Dummies           UK  Barbara Bracket
```

Now as a result you get a dataframe from a starting tree structure.

The Format HDF5

So far you have seen how to write and read data in text format. When your data analysis involves large amounts of data, it is preferable to use them in binary format. There are several tools in Python to handle binary data. A library that is having some success in this area is the `HDF5` library.

The HDF term stands for *hierarchical data format*, and in fact this library is concerned with reading and writing HDF5 files containing a structure with nodes and the possibility to store multiple datasets.

This library, fully developed in C, however, has also interfaces with other types of languages like Python, MATLAB, and Java. It is very efficient, especially when using this format to save huge amounts of data. Compared to other formats that work more simply in binary, HDF5 supports compression in real time, thereby taking advantage of repetitive patterns in the data structure to compress the file size.

At present, the possible choices in Python are `PyTables` and `h5py`. These two forms differ in several aspects and therefore their choice depends very much on the needs of those who use it.

`h5py` provides a direct interface with the high-level APIs HDF5, while `PyTables` makes abstract many of the details of HDF5 to provide more flexible data containers, indexed tables, querying capabilities, and other media on the calculations.

pandas has a class-like dict called HDFStore, using PyTables to store pandas objects. So before working with the format HDF5, you must import the HDFStore class:

```
>>> from pandas.io.pytables import HDFStore
```

Now you're ready to store the data of a dataframe within an .h5 file. First, create a dataframe.

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4),
...                      index=['white','black','red','blue'],
...                      columns=['up','down','right','left'])
```

Now create a file HDF5 calling it mydata.h5, then enter the data inside of the dataframe.

```
>>> store = HDFStore('mydata.h5')
>>> store['obj1'] = frame
From here, you can guess how you can store multiple data structures within
the same HDF5 file, specifying for each of them a label.
>>> frame
       up  down  right  left
white   0   0.5      1   1.5
black   2   2.5      3   3.5
red     4   4.5      5   5.5
blue    6   6.5      7   7.5
>>> store['obj2'] = frame
```

So with this type of format, you can store multiple data structures in a single file, represented by the store variable.

```
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1            frame        (shape->[4,4])
```

Even the reverse process is very simple. Taking account of having an HDF5 file containing various data structures, objects inside can be called in the following way:

```
>>> store['obj2']
       up  down  right  left
white   0   0.5      1   1.5
black   2   2.5      3   3.5
red     4   4.5      5   5.5
blue    6   6.5      7   7.5
```

Pickle—Python Object Serialization

The `pickle` module implements a powerful algorithm for serialization and deserialization of a data structure implemented in Python. Pickling is the process in which the hierarchy of an object is converted into a stream of bytes.

This allows an object to be transmitted and stored, and then to be rebuilt by the receiver itself retaining all the original features.

In Python, the picking operation is carried out by the `pickle` module, but currently there is a module called `cPickle` which is the result of an enormous amount of work optimizing the `pickle` module (written in C). This module can be in fact in many cases even 1,000 times faster than the `pickle` module. However, regardless of which module you do use, the interfaces of the two modules are almost the same.

Before moving to explicitly mention the I/O functions of pandas that operate on this format, let's look in more detail at the `cPickle` module and see how to use it.

Serialize a Python Object with cPickle

The data format used by the `pickle` (or `cPickle`) module is specific to Python. By default, an ASCII representation is used to represent it, in order to be readable from the human point of view. Then, by opening a file with a text editor, you may be able to understand its contents. To use this module, you must first import it:

```
>>> import pickle
```

Then create an object sufficiently complex to have an internal data structure, for example a `dict` object.

```
>>> data = { 'color': ['white','red'], 'value': [5, 7]}
```

Now you will perform a serialization of the `data` object through the `dumps()` function of the `cPickle` module.

```
>>> pickled_data = pickle.dumps(data)
```

Now, to see how it serialized the `dict` object, you need to look at the contents of the `pickled_data` variable.

```
>>> print(pickled_data)
(dp1
S'color'
p2
(lp3
S'white'
p4
aS'red'
p5
asS'value'
p6
(lp7
I5
aI7
as.
```

Once you have serialized data, they can easily be written on a file or sent over a socket, pipe, etc.

After being transmitted, it is possible to reconstruct the serialized object (deserialization) with the `loads()` function of the `cPickle` module.

```
>>> nframe = pickle.loads(pickled_data)
>>> nframe
{'color': ['white', 'red'], 'value': [5, 7]}
```

Pickling with pandas

When it comes to pickling (and unpickling) with the pandas library, everything is much easier. There is no need to import the `cPickle` module in the Python session and the whole operation is performed implicitly.

Also, the serialization format used by pandas is not completely in ASCII.

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4), index =
['up','down','left','right'])
>>> frame.to_pickle('frame.pkl')
```

There is a new file called `frame.pkl` in your working directory that contains all the information about the `frame` dataframe.

To open a PKL file and read the contents, simply use this command:

```
>>> pd.read_pickle('frame.pkl')
        0   1   2   3
up      0   1   2   3
down    4   5   6   7
left    8   9  10  11
right  12  13  14  15
```

As you can see, all the implications on the operation of pickling and unpickling are completely hidden from the pandas user, making the job as easy and understandable as possible, for those who must deal specifically with data analysis.

Note When you use this format make sure that the file you open is safe. Indeed, the pickle format was not designed to be protected against erroneous and maliciously constructed data.

Interacting with Databases

In many applications, the data rarely come from text files, given that this is certainly not the most efficient way to store data.

The data are often stored in an SQL-based relational database, and also in many alternative NoSQL databases that have become very popular in recent times.

Loading data from SQL in a dataframe is sufficiently simple and pandas has some functions to simplify the process.

The pandas.io.sql module provides a unified interface independent of the DB, called sqlalchemy. This interface simplifies the connection mode, since regardless of the DB, the commands will always be the same. To make a connection you use the create_engine() function. With this feature you can configure all the properties necessary to use the driver, as a user, password, port, and database instance.

Here is a list of examples for the various types of databases:

```
>>> from sqlalchemy import create_engine
```

For PostgreSQL:

```
>>> engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')
For MySQL
>>> engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')
For Oracle
>>> engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')
For MSSQL
>>> engine = create_engine('mssql+pyodbc://mydsn')
For SQLite
>>> engine = create_engine('sqlite:///foo.db')
```

Loading and Writing Data with SQLite3

As a first example, you will use a SQLite database using the driver's built-in Python sqlite3. SQLite3 is a tool that implements a DBMS SQL in a very simple and lightweight way, so it can be incorporated in any application implemented with the Python language. In fact, this practical software allows you to create an embedded database in a single file.

This makes it the perfect tool for anyone who wants to have the functions of a database without having to install a real database. SQLite3 could be the right choice for anyone who wants to practice before going on to a real database, or for anyone who needs to use the functions of a database to collect data, but remaining within a single program, without having to interface with a database.

Create a dataframe that you will use to create a new table on the SQLite3 database.

```
>>> frame = pd.DataFrame( np.arange(20).reshape(4,5),
...                       columns=['white','red','blue','black','green'])
>>> frame
   white  red  blue  black  green
0      0    1     2      3      4
1      5    6     7      8      9
2     10   11    12     13     14
3     15   16    17     18     19
```

Now it's time to implement the connection to the SQLite3 database.

```
>>> engine = create_engine('sqlite:///foo.db')
```

Convert the dataframe in a table within the database.

```
>>> frame.to_sql('colors',engine)
```

Instead, to read the database, you have to use the `read_sql()` function with the name of the table and the `engine`.

```
>>> pd.read_sql('colors',engine)
   index  white  red  blue  black  green
0      0      0    1     2      3      4
1      1      5    6     7      8      9
2      2     10   11    12     13     14
3      3     15   16    17     18     19
```

As you can see, even in this case, the writing operation on the database has become very simple thanks to the I/O APIs available in the pandas library.

Now you'll see instead the same operations, but not using the I/O API. This can be useful to get an idea of how pandas proves to be an effective tool for reading and writing data to a database.

First, you must establish a connection to the DB and create a table by defining the corrected data types, so as to accommodate the data to be loaded.

```
>>> import sqlite3
>>> query = """
... CREATE TABLE test
```

```
... (a VARCHAR(20), b VARCHAR(20),
...  c REAL,        d INTEGER
... );"""
>>> con = sqlite3.connect(':memory:')
>>> con.execute(query)
<sqlite3.Cursor object at 0x0000000009E7D730>
>>> con.commit()
```

Now you can enter data using the SQL `INSERT` statement.

```
>>> data = [('white','up',1,3),
...         ('black','down',2,8),
...         ('green','up',4,4),
...         ('red','down',5,5)]
>>> stmt = "INSERT INTO test VALUES(?,?,?,?)"
>>> con.executemany(stmt, data)
<sqlite3.Cursor object at 0x0000000009E7D8F0>
>>> con.commit()
```

Now that you've seen how to load the data on a table, it is time to see how to query the database to get the data you just recorded. This is possible using an SQL `SELECT` statement.

```
>>> cursor = con.execute('select * from test')
>>> cursor
<sqlite3.Cursor object at 0x0000000009E7D730>
>>> rows = cursor.fetchall()
>>> rows
[(u'white', u'up', 1.0, 3), (u'black', u'down', 2.0, 8), (u'green', u'up',
4.0, 4), (u'red', 5.0, 5)]
```

You can pass the list of tuples to the constructor of the dataframe, and if you need the name of the columns, you can find them within the `description` attribute of the cursor.

```
>>> cursor.description
(('a', None, None, None, None, None, None), ('b', None, None, None, None,
None, None), ('c
```

```
one, None, None, None, None), ('d', None, None, None, None, None, None))
>>> pd.DataFrame(rows, columns=zip(*cursor.description)[0])
       a     b  c  d
0  white    up  1  3
1  black  down  2  8
2  green    up  4  4
3    red  down  5  5
```

As you can see, this approach is quite laborious.

Loading and Writing Data with PostgreSQL

From pandas 0.14, the PostgreSQL database is also supported. So double-check if the version on your PC corresponds to this version or greater.

```
>>> pd.__version__
>>> '0.22.0'
```

To run this example, you must have installed on your system a PostgreSQL database. In my case I created a database called `postgres`, with `postgres` as the user and `password` as the password. Replace these values with the values corresponding to your system.

The first thing to do is install the `psycopg2` library, which is designed to manage and handle the connection with the databases.

With Anaconda:

```
conda install psycopg2
```

Or if you are using PyPi:

```
pip install psycopg2
```

Now you can establish a connection with the database:

```
>>> import psycopg2
>>> engine = create_engine('postgresql://postgres:password@localhost:5432/
postgres')
```

Note In this example, depending on how you installed the package on Windows, often you get the following error message:

```
from psycopg2._psycopg import BINARY, NUMBER, STRING,
DATETIME, ROWID

ImportError: DLL load failed: The specified module could not
be found.
```

This probably means you don't have the PostgreSQL DLLs (`libpq.dll` in particular) in your `PATH`. Add one of the `postgres\x.x\bin` directories to your `PATH` and you should be able to connect from Python to your PostgreSQL installations.

Create a dataframe object:

```
>>> frame = pd.DataFrame(np.random.random((4,4)),
              index=['exp1','exp2','exp3','exp4'],
              columns=['feb','mar','apr','may']);
```

Now we see how easily you can transfer this data to a table. With `to_sql()` you will record the data in a table called `dataframe`.

```
>>> frame.to_sql('dataframe',engine)
```

pgAdmin III is a graphical application for managing PostgreSQL databases. It's a very useful tool and is present on Linux and Windows. With this application, it is easy to see the table dataframe you just created (see Figure 5-5).

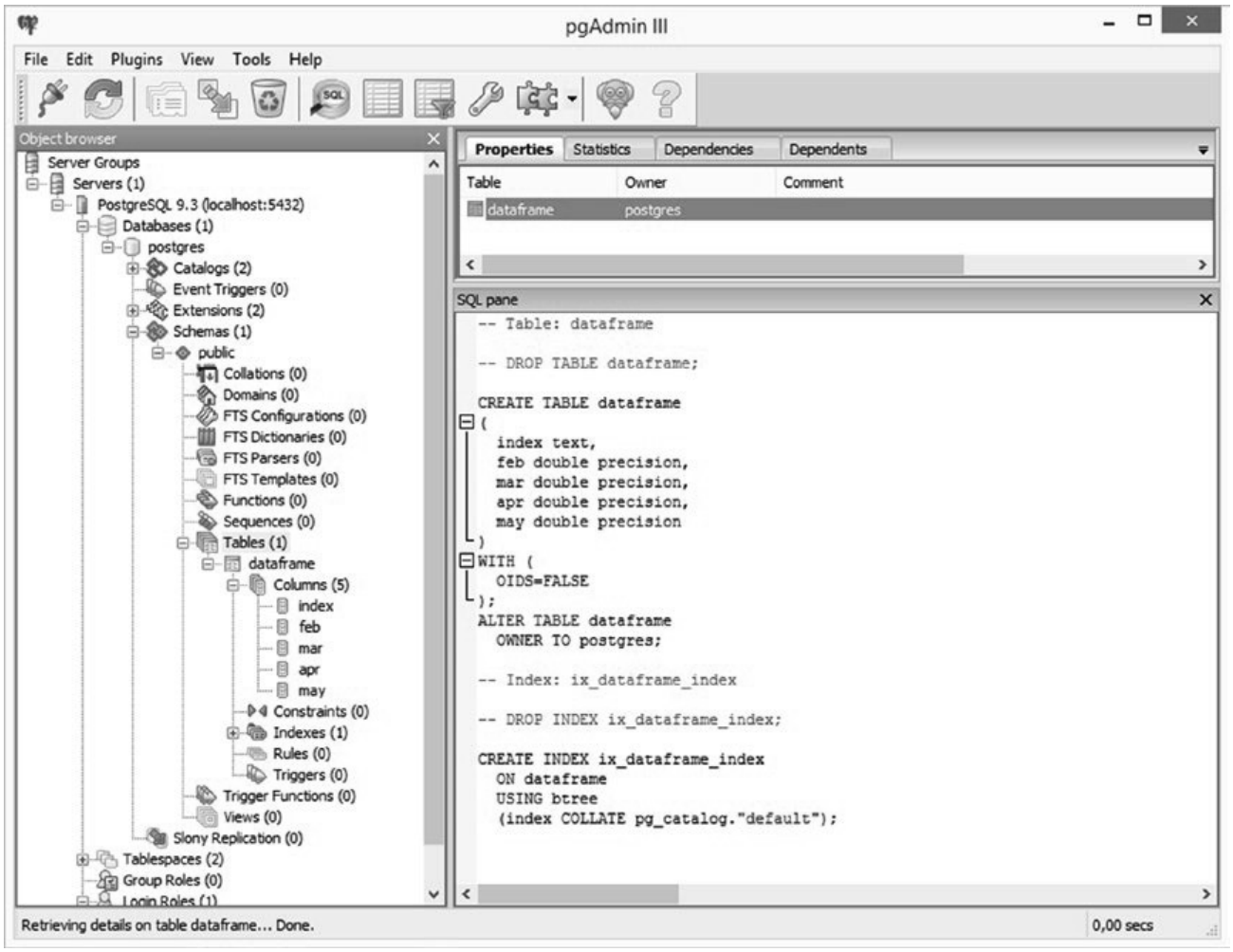


Figure 5-5. *The pgAdmin III application is a perfect graphical DB manager for PostgreSQL*

If you know the SQL language well, a more classic way to see the new created table and its contents is using a `psql` session.

```
>>> psql -U postgres
```

In my case, I am connected to the `postgres` user; it may be different in your case. Once you're connected to the database, perform an SQL query on the newly created table.

```
postgres=# SELECT * FROM DATAFRAME;
index|       feb       |       mar       |       apr       |       may
-----+-----------------+-----------------+-----------------+----------------
exp1 |0.757871296789076|0.422582915331819|0.979085739226726|0.332288515791064
exp2 |0.124353978978927|0.273461421503087|0.049433776453223|0.0271413946693556
exp3 |0.538089036334938|0.097041417119426|0.905979807772598|0.123448718583967
exp4 |0.736585422687497|0.982331931474687|0.958014824504186|0.448063967996436
(4 righe)
```

Even the conversion of a table in a dataframe is a trivial operation. Even here there is a `read_sql_table()` function that reads directly on the database and returns a dataframe.

```
>>> pd.read_sql_table('dataframe',engine)
  index       feb       mar       apr       may
0  exp1  0.757871  0.422583  0.979086  0.332289
1  exp2  0.124354  0.273461  0.049434  0.027141
2  exp3  0.538089  0.097041  0.905980  0.123449
3  exp4  0.736585  0.982332  0.958015  0.448064
```

But when you want to read data in a database, the conversion of a whole and single table into a dataframe is not the most useful operation. In fact, those who work with relational databases prefer to use the SQL language to choose what data and in what form to export the data by inserting an SQL query.

The text of an SQL query can be integrated in the `read_sql_query()` function.

```
>>> pd.read_sql_query('SELECT index,apr,may FROM DATAFRAME WHERE apr >
0.5',engine)
  index       apr       may
0  exp1  0.979086  0.332289
1  exp3  0.905980  0.123449
2  exp4  0.958015  0.448064
```

Reading and Writing Data with a NoSQL Database: MongoDB

Among all the NoSQL databases (BerkeleyDB, Tokyo Cabinet, and MongoDB), MongoDB is becoming the most widespread. Given its diffusion in many systems, it seems appropriate to consider the possibility of reading and writing data produced with the pandas library during data analysis.

First, if you have MongoDB installed on your PC, you can start the service to point to a given directory.

```
mongod --dbpath C:\MongoDB_data
```

Now that the service is listening on port 27017, you can connect to this database using the official driver for MongoDB: pymongo.

```
>>> import pymongo
>>> client = MongoClient('localhost',27017)
```

A single instance of MongoDB is able to support multiple databases at the same time. So now you need to point to a specific database.

```
>>> db = client.mydatabase
>>> db
Database(MongoClient('localhost', 27017), u'mycollection')
In order to refer to this object, you can also use
>>> client['mydatabase']
Database(MongoClient('localhost', 27017), u'mydatabase')
```

Now that you have defined the database, you have to define the collection. The collection is a group of documents stored in MongoDB and can be considered the equivalent of the tables in an SQL database.

```
>>> collection = db.mycollection
>>> db['mycollection']
Collection(Database(MongoClient('localhost', 27017), u'mydatabase'),
u'mycollection')
>>> collection
Collection(Database(MongoClient('localhost', 27017), u'mydatabase'),
u'mycollection')
```

```
Now it is the time to load the data in the collection. Create a DataFrame.
>>> frame = pd.DataFrame( np.arange(20).reshape(4,5),
...                       columns=['white','red','blue','black','green'])
>>> frame
   white  red  blue  black  green
0      0    1     2      3      4
1      5    6     7      8      9
2     10   11    12     13     14
3     15   16    17     18     19
```

Before being added to a collection, it must be converted into a JSON format. The conversion process is not as direct as you might imagine; this is because you need to set the data to be recorded on DB in order to be re-extract as DataFrame as fairly and as simply as possible.

```
>>> import json
>>> record = json.loads(frame.T.to_json()).values()
>>> record
[{u'blue': 7, u'green': 9, u'white': 5, u'black': 8, u'red': 6},
{u'blue': 2, u'green': 4, u'white':
 0, u'black': 3, u'red': 1}, {u'blue': 17, u'green': 19, u'white': 15,
 u'black': 18, u'red': 16}, {u
'blue': 12, u'green': 14, u'white': 10, u'black': 13, u'red': 11}]
Now you are finally ready to insert a document in the collection,
and you can do this with the insert() function.
>>> collection.mydocument.insert(record)
[ObjectId('54fc3afb9bfbee47f4260357'), ObjectId('54fc3afb9bfbee47f4260358'),
ObjectId('54fc3afb9bfbee47f4260359'), ObjectId('54fc3afb9bfbee47f426035a')]
```

As you can see, you have an object for each line recorded. Now that the data has been loaded into the document within the MongoDB database, you can execute the reverse process, i.e., reading data in a document and then converting them to a dataframe.

```
>>> cursor = collection['mydocument'].find()
>>> dataframe = (list(cursor))
>>> del dataframe['_id']
>>> dataframe
   black  blue  green  red  white
0      8     7      9    6      5
1      3     2      4    1      0
2     18    17     19   16     15
3     13    12     14   11     10
```

You have removed the column containing the ID numbers for the internal reference of MongoDB.

Conclusions

In this chapter, you saw how to use the features of the I/O API of the pandas library in order to read and write data to files and databases while preserving the structure of the dataframes. In particular, several modes of writing and reading data according to the type of format were illustrated.

In the last part of the chapter, you saw how to interface to the most popular models of databases to record and/or read data into it directly as a dataframe ready to be processed with the pandas tools.

In the next chapter, you'll see the most advanced features of the library pandas. Complex instruments like the GroupBy and other forms of data processing are discussed in detail.