CHAPTER 5

pandas: Reading and Writing Data

In the previous chapter, you got familiar with the pandas library and with all the basic functionalities that it provides for the data analysis. You have seen that DataFrame and Series are the heart of this library. These are the material on which to perform all manipulations of data, 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 is focused on a series of I/O API functions that pandas provides to facilitate as much as possible the reading and writing data process directly as DataFrame objects on all of the most commonly used formats. You start to see the 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, with examples showing how to store the data in a DataFrame directly in them. At the same time, you will see how to read the data contained in a database and retrieve them already 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. Moreover, even the process of writing and reading data from/to external files can be considered a part of the data processing. In fact, you will see how, even at this stage, you can perform some operations in order to prepare the incoming data to further manipulations.

Thus, this part 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, completely symmetrical to each other: **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

(*continued*)

Readers	Writers
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 write and read files in text form. In particular, data are generally reported in tabular form. If the values in a row are separated by a comma, you have the CSV (comma-separated values) format, which is perhaps the best-known and most popular format.

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

So this type of file is the most common source of data and actually even 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 common experience, the most common operation for a person approaching data analysis is to read the data contained in a CSV file, or at least in a text file.

In order to see how pandas handle this kind of data, start by creating a small CSV file in the working directory as shown in Listing 5-1 and save it as **myCSV_01.csv**.

Listing 5-1. myCSV_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 at the same time in a DataFrame object.

```
>>> csvframe = read_csv('myCSV_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 the reading of the data in a CSV file is a rather trivial. CSV files are tabulated data in which the values on the same column are separated by commas. But since CSV files are considered text files, you can also use the **read_table()** function, but specifying the delimiter.

```
>>> 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 the example you just saw, you can notice that in the CSV file, headers to 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 from the first line (see Listing 5-2).

Listing 5-2. myCSV_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
```

```
>>> 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, then you could make sure that it is precisely pandas to assign default names to the columns by using the **header** option set to None.

```
>>> 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, there is also the possibility to specify the names directly assigning a list of labels to the **names** option.

```
>>> 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 it.

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 **myCSV_03.csv** (Listing 5-3).

Listing 5-3. myCSV_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
```

```
>>> 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 for Parsing 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 the use of a regexp and how you can apply it as a criterion for separation of values, you can start from a simple case. For example, suppose that your file, such as a TXT file, has values separated by spaces or tabs in an unpredictable order. In this case, you have to use the regexp because only with it you will take into account as a separator both cases. You can do that using the wildcard /**s***. /s stands for space or tab character (if you wanted to indicate only the tab, you would have used /t), while the pound indicates that these characters may be multiple (see Table 5-1 for other wildcards most commonly used). 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 a case a little extreme, in which we have the values separated from each other by tab or space in a totally random order (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
```

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

As we can see the result we got is a perfect data frame in which the values are perfectly ordered.

Now you will see an example that may seem strange, or unusual, but actually it is not so rare as it may seem. This example can be very helpful to understand the high potential of a regexp. In fact, usually you think of the separators as special characters like commas, spaces, tabs, etc. but in reality you could consider separator characters as alphanumeric characters, or for example, as 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 literal characters are completely fused.

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

Listing 5-5. ch05_05.txt

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

```
>>> read_table('ch05_05.txt',sep='\D*',header=None)
   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 within 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, then you have to write skiprows = 5, but if we 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
```

```
>>> 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 or Partially

When large files are processed, or when you're 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 doing the parsing of the entire file.

So if for example you want 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).

```
>>> 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 we 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 of a column every three rows of the file and then insert these sums within a series. This example is trivial and impractical but is very simple to understand, so that once you have learned the underlying mechanism you will be able to apply it in much more complex cases.

```
>>> out = Series()
>>> i = 0
>>> pieces = 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 within a file, the writing of a data file produced by a calculation, or in general the data contained in a data structure, is a common and necessary operation.

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

```
>>> frame2
ball  pen  pencil  paper
0       1       2      3
4       5       6      7
8       9      10     11
12     13      14     15
>>> frame2.to_csv('ch05_07.csv')
```

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 make the writing of a data frame to a file, by default both indexes and columns are marked on the file. This default behavior can be changed by placing the two options **index** and **header** set to False (Listing 5-8).

```
>>> frame2.to_csv('ch05_07b.csv', index=False, header=False)
```

Listing 5-8. ch05_08.csv

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

One thing to take into account when making the writing of files is that NaN values present in a data structure are shown as empty fields in the file (Listing 5-9).

```
>>> frame3
        ball  mug  paper  pen  pencil
blue       6  NaN    NaN    6     NaN
green    NaN  NaN    NaN  NaN     NaN
red      NaN  NaN    NaN  NaN     NaN
white     20  NaN    NaN   20     NaN
yellow    19  NaN    NaN   19     NaN
>>> frame3.to_csv('ch05_08.csv')
```

Listing 5-9. ch05_09.csv

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

But 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 (Listing 5-10).

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

Listing 5-10. ch05_10.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, data frame has always been the subject of discussion since usually 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

Also with regard to the HTML format pandas provides the corresponding pair of I/O API functions.

- read_html()
- to_html()

To have these two functions can be very useful. You will appreciate the ability to convert complex data structures such as DataFrame directly in HTML tables without having to hack a long listing in HTML, especially if you're dealing with the world 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 the 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 recommend strongly to install the module html5lib. Anaconda specified:

```
conda install html5lib
```

Writing Data in HTML

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

Because sometimes the data structures as the DataFrame can be quite complex and large, to have a function like this is a great resource for anyone who needs 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 have the ability to directly convert the DataFrame in an HTML table.

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

Since the I/O API functions are defined within 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 data frame.

In the next example you'll see how the table appears automatically generated within an HTML file. In this regard, we create a data frame 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.

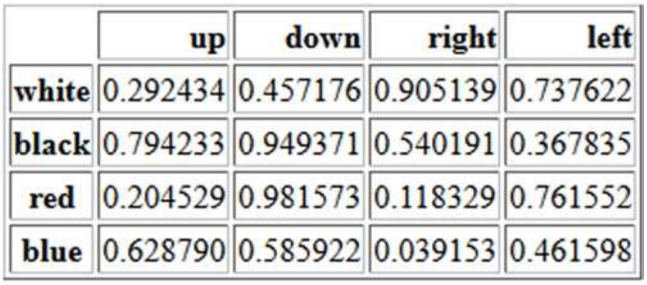
First of all we create a string that contains the code of the HTML page.

```
>>> 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 variable html, 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 data frame. 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 DataFrame even if there is only one table. As regards the source to be subjected to parsing, this can be of 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 we 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 within 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('http://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 will be shown 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: `http://lxml.de/index.html`.

Take for example an XML file as 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">
      272103_1_EnRoss, Mark</Author>
      <Title>XML Cookbook</Title>
      <Genre>Computer</Genre>
      <Price>23.56</Price>
      <PublishDate>2014-22-01</PublishDate>
   </Book>
   <Book id="ISBN9872122367564">
      272103_1_EnBracket, 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. To do so 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 0x0000000009734E08>
You got an object tree, which is an internal data structure of the module lxml.
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 within 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 data frame. Define the following function, which has the task of analyzing the entire contents of a 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

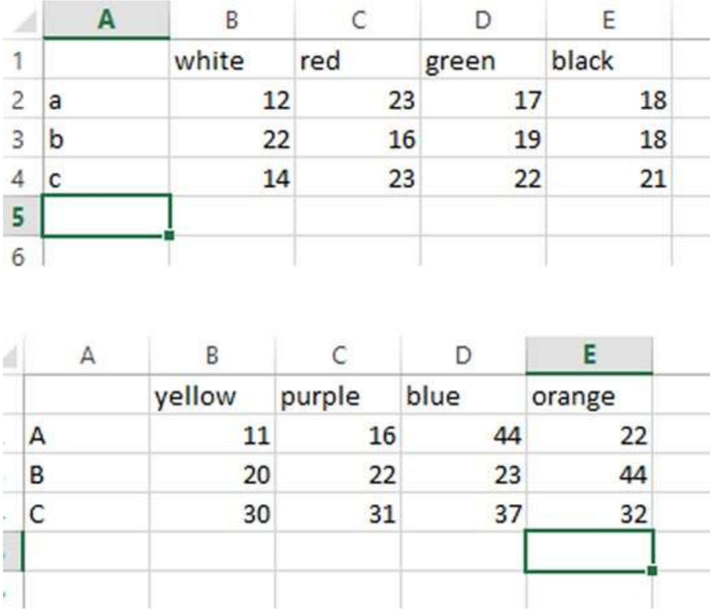
In the previous section, you saw how the data can be easily read from CSV files. It is not uncommon, however, that there are data collected in tabular form in the Excel spreadsheet.

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

- to_excel()
- read_excel()

As regards reading Excel files, the **read_excel()** function is able to read both 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 data in sheet1 and sheet2. Then save it as **data.xls**.



	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
	yellow	purple	blue	orange
A	11	16	44	22
B	20	22	23	44
C	30	31	37	32

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

To read the data contained within the XLS file and obtain the conversion into a data frame, you only have to use the **read_excel()** function.

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

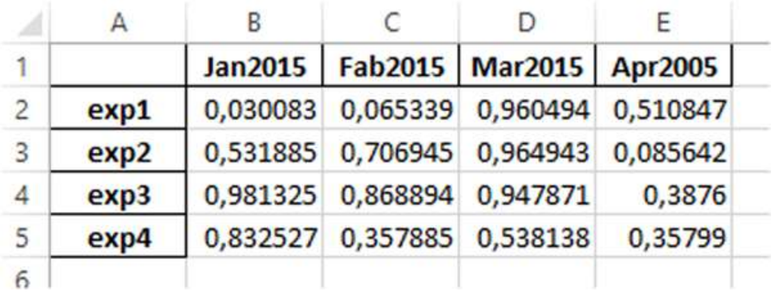
As you can see, by default, the returned DataFrame is composed of the data tabulated in the first spreadsheets. If, however, you'd need to load the data in the second spreadsheet, and then specify the name of the sheet or the number of the sheet (index) just as the second argument.

```
>>> pd.read_excel('data.xls','Sheet2')
   yellow  purple  blue  orange
A      11      16    44      22
B      20      22    23      44
C      30      31    37      32
>>> pd.read_excel('data.xls',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. So to convert a data frame in a spreadsheet on Excel you have to write as follows.

```
>>> 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*

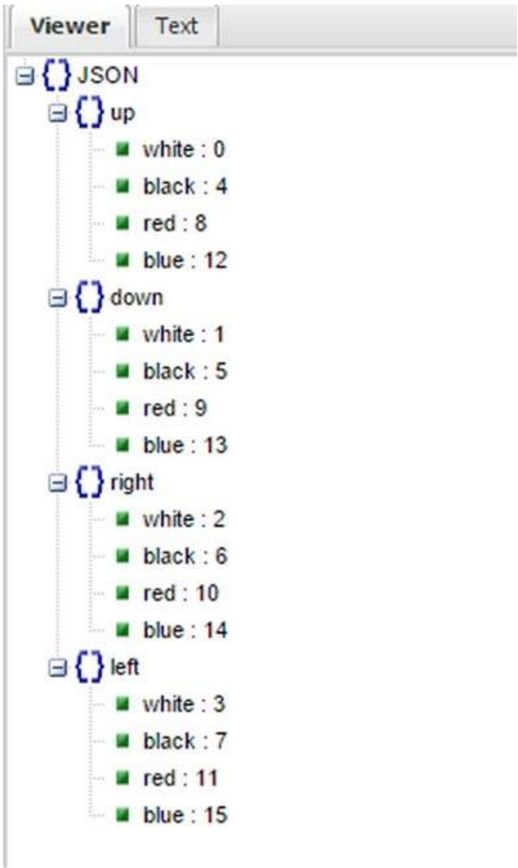
JSON Data

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

The special feature of this format is its great flexibility, though 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 entered or copied data in JSON format, allows you to see if the format you entered is invalid. Moreover it displays the tree structure so that you can better understand its structure (as shown in Figure 5-4).



```
{} 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
```

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 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 in tabular form. You can refer this process as **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 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.

```
>>> 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 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 key. All properties will be converted into nested column names while the corresponding values will fill the DataFrame. As regards 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(text2,'books',['writer','nationality'])
   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 got 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 the 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 the reading and writing of 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. Thus, its extensive use is one of the reasons for its rapid spread. But not only for this, in fact; another reason is its efficiency, 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 within the data structure to compress the file size.

At present, the possible choices in Python are two: **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 class HDFStore.

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

Now you're ready to store the data of a data frame within a file .h5. 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 inside the data 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.
>>> frame2
       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'] = frame2
```

So with this type of format, you can store multiple data structures within 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 de-serialization 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 cPickle module and how to use it.

Serialize a Python Object with cPickle

The data format used by the pickle module (or cPickle) 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 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 cPickle as 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 was serialized the dict object, you need to look at the content 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

As regards the operation of pickling (and unpickling) with the pandas library, everything remains much facilitated. No need to import the cPickle module in the Python session and also 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')
```

Now in your working directory there is a new file called **frame.pkl** containing all the information about the **frame** DataFrame.

To open a PKL file and read the contents, simply use the 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, allowing it to make the job as easy and understandable as possible, for those who must deal specifically with the data analysis.

■ **Note** When you make use of 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. For making 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 within any application implemented with 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 for the collection of data, but remaining within a single program, without worry to interface with a database.

Create a data frame 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 make a reading of 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 even as regards the reading and writing data to a database.

First, you must establish a connection to the DB and create a table by defining data types corrected, 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 through 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 through 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', None, 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 may well see this approach is quite laborious.

Loading and Writing Data with PostgreSQL

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

```
>>> pd.__version__
>>> '0.15.2'
```

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

Now establish a connection with the database:

```
>>> 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 the **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 both Linux and Windows. With this application is easy to see the table dataframe just created (see Figure 5-5).

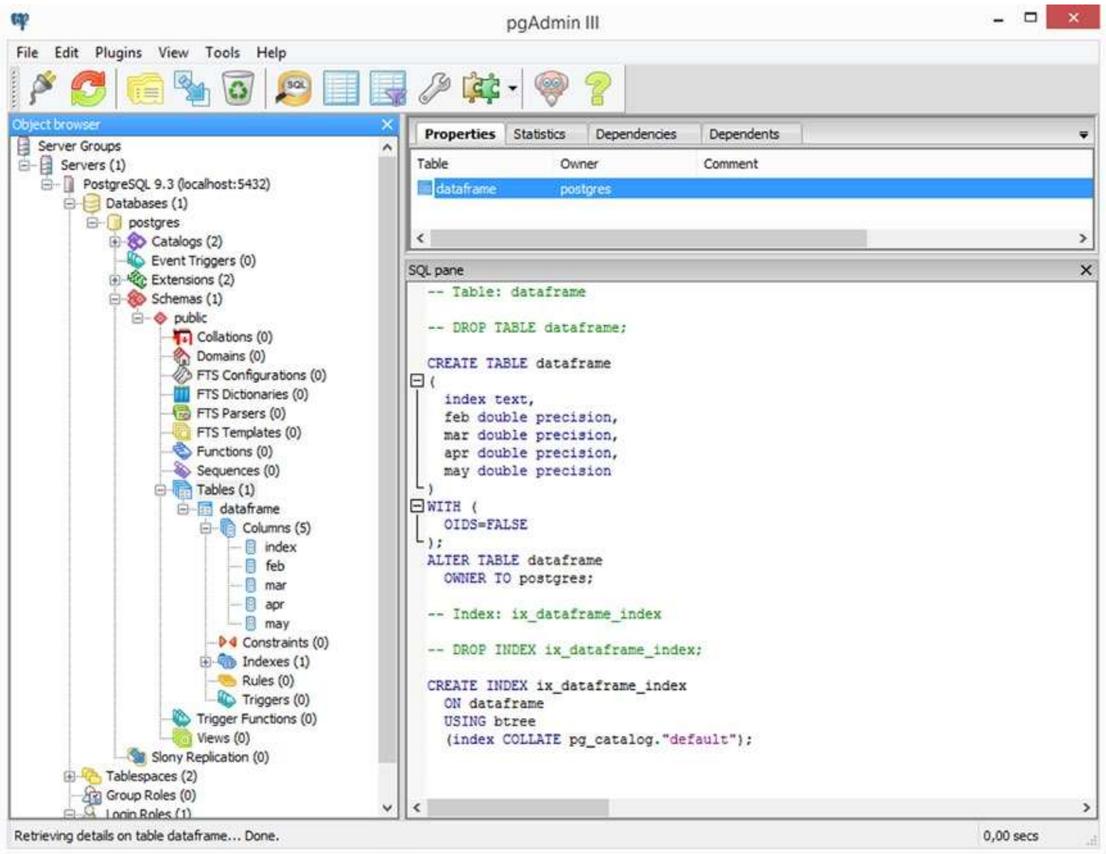


Table	Owner	Comment
dataframe	postgres	

```
-- Table: dataframe

-- DROP TABLE dataframe;

CREATE TABLE dataframe
(
  index text,
  feb double precision,
  mar double precision,
  apr double precision,
  may double precision
)
WITH (
  OIDS=FALSE
);
ALTER TABLE dataframe
  OWNER TO postgres;

-- Index: ix_dataframe_index

-- DROP INDEX ix_dataframe_index;

CREATE INDEX ix_dataframe_index
  ON dataframe
  USING btree
  (index COLLATE pg_catalog."default");
```

Retrieving details on table dataframe... Done.   0,00 secs

Figure 5-5. *The pdAdminIII 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 through a **psql** session.

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

In my case I am connected with the postgres user; it may be different in your case. Once 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 make a reading of 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 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, 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 a 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 and at the same time 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 within 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 according to the type of format are illustrated.

In the last part of the chapter you have seen how to interface to the most popular models of Database to record and/or read the data into it directly as 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.