CHAPTER 4

The pandas Library—An Introduction

With this chapter, you can finally get into the heart of this book: the pandas library. This fantastic Python library is a perfect tool for anyone who wants to practice data analysis using Python as a programming language.

First you will find out the fundamental aspects of this library and how to install it on your system, and finally you will become familiar with the two data structures called Series and DataFrame. In the course of the chapter you will work with a basic set of functions, provided by the pandas library, to perform the most common tasks in the data processing. Getting familiar with these operations will be a key issue for the rest of the book. This is the reason that it is very important for you to repeat this chapter until you will have familiarity with all its content.

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

Finally, you will see how it is possible to extend the concept of indexing to multiple levels at the same time: the hierarchical indexing.

pandas: The Python Data Analysis Library

Pandas is an open source Python library for highly specialized data analysis. Currently it is the reference point that all professionals using the Python language need to study and analyze data sets for statistical purposes of analysis and decision making.

This library has been designed and developed primarily by Wes McKinney starting in 2008; later, 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.

Pandas arises from the need to have a specific library for analysis of the data which provides, in the simplest possible way, all the instruments for the processing of data, 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 of the other modules, but also takes advantage of the high quality of performance in calculating of the NumPy module.

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

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

Throughout the book, in fact, we will see a series of basic operations for data analysis, normally used on the database tables or spreadsheets. Pandas in fact provides an extended set of functions and methods that allow you to perform, and in many cases even in the best way, these operations.

So pandas has as its main purpose to provide all the building blocks for anyone approaching the world of data analysis.

Installation

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

Installation from Anaconda

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

```
command.conda list pandas
```

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

```
# packages in environment at C:\Users\Fabio\Anaconda:
#
pandas                    0.14.1               np19py27_0
```

If in your case it should not be so, you will need to install the module pandas. 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.

If you want to upgrade your package to a newer version, the command 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 suggest any updates, and then ask if you want to proceed or not to the update.

```
Fetching package metadata: ....
Solving package specifications: .
Package plan for installation in environment C:\Users\Fabio\Anaconda:
```

The following packages will be downloaded:

```
package                    |            build
---------------------------|-----------------------------------
pytz-2014.9                |           py27_0         169 KB
requests-2.5.3             |           py27_0         588 KB
six-1.9.0                  |           py27_0          16 KB
conda-3.9.1                |           py27_0         206 KB
pip-6.0.8                  |           py27_0         1.6 MB
scipy-0.15.1               |       np19py27_0        71.3 MB
pandas-0.15.2              |       np19py27_0         4.3 MB
------------------------------------------------------------
                                       Total:        78.1 MB
```

The following packages will be UPDATED:

```
conda:    3.9.0-py27_0      --> 3.9.1-py27_0
pandas:   0.14.1-np19py27_0 --> 0.15.2-np19py27_0
pip:      1.5.6-py27_0      --> 6.0.8-py27_0
pytz:     2014.7-py27_0     --> 2014.9-py27_0
requests: 2.5.1-py27_0      --> 2.5.3-py27_0
scipy:    0.14.0-np19py27_0 --> 0.15.1-np19py27_0
six:      1.8.0-py27_0      --> 1.9.0-py27_0

Proceed ([y]/n)?
```

After pressing 'y,' Anaconda will begin to do the download of all modules updated from the network. So when you perform this step it will be necessary that the PC is connected to the network.

```
Fetching packages ...
scipy-0.15.1-n 100% |###############################| Time: 0:01:11   1.05 MB/s
Extracting packages ...
[      COMPLETE      ] |##################################################| 100%
Unlinking packages ...
[      COMPLETE      ] |##################################################| 100%
Linking packages ...
[      COMPLETE      ] |##################################################| 100%
```

Installation from PyPI

pandas can also be installed by PyPI:

```
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 module pandas like any other package.

```
On Debian and Ubuntu distributions:

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 module pandas starting from the source code, you can find what you need on Github to the link http://github.com/pydata/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 please 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

For those who are working on Windows and prefer to manage their packages themselves 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 application pip (see PyPI in Chapter 2).

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

In the choice of 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 any package manager that will help you in some way to manage the 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 distributions as in Anaconda.

Test Your pandas Installation

The pandas library also provides the ability to run after its installation a test to check on the executability of its 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 module nose in your Python distribution (see the "Nose Module" sidebar). If you do, then you can start the test by entering the following command:

```
nosetests pandas
```

The test will take several minutes to perform its task, and in the end it will show a list of the problems encountered.

NOSE MODULE

This module is designed for testing the 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 (http://pythontesting.net/framework/nose/nose-introduction/)

Getting Started with pandas

Given the nature of the topic covered in this chapter, centered on the explanation of the data structures and functions/methods applied to it, the writing of large listings or scripts is not required.

Thus the approach I felt better for this chapter is opening a Python shell and typing commands one by one. In this way, the player has the opportunity to become familiar with the individual functions and data structures, gradually explained in this chapter.

Furthermore, the data and functions defined in the various examples remain valid in the following ones, thus avoiding the reader needing to define each time around. The reader is invited, at the end of each example, to repeat the various commands, modifying them if appropriate, and to control how the values within the data structures vary during operation. This approach is great for getting familiar with the different topics covered in this chapter, leaving the reader the opportunity to interact freely with what you are reading and not to end up in easy automation as write and execute.

■ **Note** This chapter assumes that you have some familiarity with Python and NumPy in general. If you have any difficulty, you should 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 module pandas 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 function, object, or method with pd; this approach is not considered a good practice by the Python community in general.

Introduction to pandas Data Structures

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

- Series
- DataFrame

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 the problems, they do provide a valid and robust tool for most applications. In fact, in their simplicity, 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 factor will lead to a high manipulability of these data structures.

The Series

The **Series** is the object of the pandas library designed to represent one-dimensional data structures, similarly 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 has the purpose to hold 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 as specified in Figure 4-1, simply call the Series() constructor passing 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 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 within 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.

So in this case it will be necessary, during the constructor call, to include the **index** option assigning 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

For that concerning individual elements, you can select them as an ordinary numpy array, 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
```

or even in this case, use the corresponding labels, but specifying the list of labels within 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 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 Series from NumPy Arrays and Other Series

You can define new Series starting with NumPy arrays or existing Series.

```
>>> arr = np.array([1,2,3,4])
>>> s3 = pd.Series(arr)
>>> s3
0    1
1    2
2    3
3    4
dtype: int32

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

When doing this, however, you should always keep in mind that the values contained within the NumPy array or 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: int32
```

```
>>> arr[2] = -2
>>> s3
0    1
1    2
2   -2
3    4
dtype: int32
```

As we 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 NumPy library as the base for the development of the pandas library and as a result, for its data structures, many operations applicable to NumPy arrays are extended to the Series. One of these is the filtering of the values contained within the data structure through conditions.

For example, if you need to know which elements within the series have value greater than 8, you will write the following:

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

Operations and Mathematical Functions

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

Regarding the operators you can simply write the arithmetic expression.

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

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

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

Evaluating Values

Often within a Series there may be duplicate values and then you may need to have information on what are the samples contained, counting duplicates and whether a value is present or not in the Series.

In this regard, 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 within the Series excluding duplicates, you can use the **unique()** function. The return value is an array containing the unique values in the Series, though not necessarily in order.

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

A function similar to unique() is the **value_counts()** function, which not only returns the unique values but calculates occurrences within a Series.

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

Finally, **isin()** is a function that evaluates the membership, that is, given a list of values, this function lets you know if these values are contained within the data structure. Boolean values that are returned can be very useful during the filtering of data within 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 within pandas data structures to indicate the presence of an empty field or not definable numerically.

Generally, these NaN values are a problem and must be managed in some way, especially during data analysis. These data are generated especially when extracting data from some source gave some trouble, or even when the source is a 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 we will see how to apply different strategies to address the problem of NaN values.

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

```
>>> s2 = pd.Series([5,-3,np.NaN,14])
>>> s2
0     5
1    -3
2   NaN
3    14
```

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 contains 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 useful to be placed inside a filtering to make a condition.

```
>>> s2[s2.notnull()]
0     5
1    -3
3    14
dtype: float64
>>> s2[s2.isnull()]
2   NaN
dtype: float64
```

Series as Dictionaries

An alternative way to see a Series is to think of them 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 dict previously defined.

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

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

```
>>> colors = ['red','yellow','orange','blue','green']
>>> myseries = pd.Series(mydict, index=colors)
red       2000
yellow     500
orange    1000
blue      1000
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 the ability of Series to align data addressed differently between them by identifying their corresponding label.

In the following example you sum two Series having only some elements in common with label.

```
>>> mydict2 = {'red':400,'yellow':1000,'black':700}
>>> myseries2 = pd.Series(mydict2)
>>> myseries + myseries2
black      NaN
blue       NaN
orange     NaN
green      NaN
red       2400
yellow    1500
dtype: float64
```

You get a new object Series in which only the items with the same label are added. All other label 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 the Spreadsheet (the most familiar are Excel spreadsheets). This data structure is designed to extend the case of the 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 different type (numeric, string, Boolean, etc.).

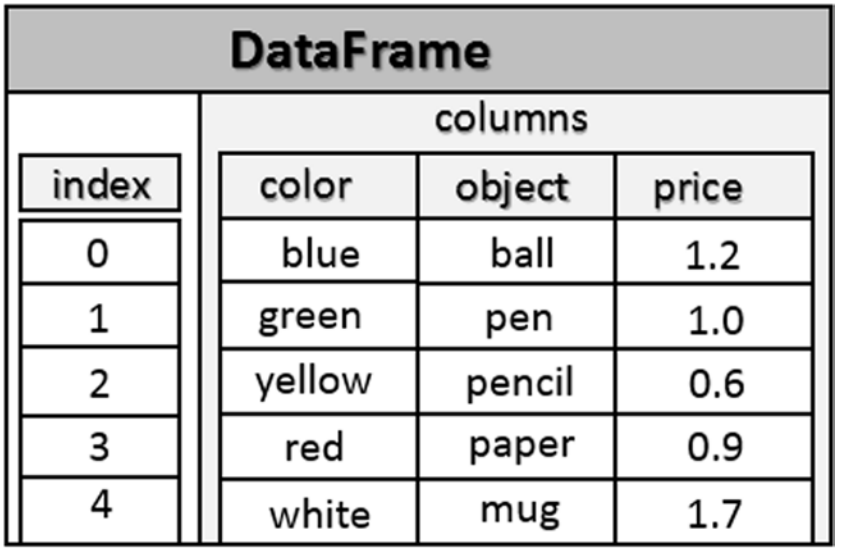

Figure 4-2. *The DataFrame structure*

Unlike Series, which had an **Index** array containing labels associated with each element, in the case of the data frame, there are two index arrays. The first, 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 instead 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 values are the Series that will form the columns of the data frame. Furthermore, all elements of each Series are mapped according to an array of labels, called 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 we 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 object dict from which we want to create a DataFrame contains more data than we are interested, you can make a selection. In the constructor of the data frame, 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 within the object dict.

```
>>> 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 within 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 assigning 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 **index** and **columns**, it is easy to imagine an alternative way to define a DataFrame. Instead of using a dict object, you can define within the constructor three arguments, in the following order: a data matrix, then an array containing the labels assigned to the **index** option, and finally 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 quickly and easily a matrix of values you can use **np.arange(16).reshape((4,4))** that generates a 4x4 matrix of increasing numbers 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

First, if we want to know the name of all the columns of a DataFrame is sufficient to specify the **columns** attribute on the instance of the DataFrame object.

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

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

```
>>> frame.index
Int64Index([0, 1, 2, 3, 4], dtype='int64')
```

As regards the values contained within the data structure, you can get the entire set of data using the **values** attribute.

```
>>> frame.values
array([['blue', 'ball', 1.2],
       ['green', 'pen', 1.0],
       ['yellow', 'pencil', 3.3],
       ['red', 'paper', 0.9],
       ['white', 'mug', 1.7]], dtype=object)
```

Or, if you are interested to select 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 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
```

Regarding the rows within a data frame, it is possible to use the **ix** attribute with the index value of the row that you want to extract.

```
>>> frame.ix[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, whereas the values have become the data of Series.

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

```
>>> frame.ix[[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 data frame 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 data frame 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, first you have 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, just 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 for identifying 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    3.3
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 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 the result, there is a new column called 'new' with the value within 12 replicated for each of its elements.

If, however, you want to do an update of 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 function **np.arange()** to update the values of a column with a predetermined sequence.

The columns of a data frame 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: int32
>>> 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, simply select the item and give it the new value.

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

Membership of a Value

You have already seen the function **isin()** applied to the Series to decide the membership of a set of values. Well, this feature is also applicable on DataFrame objects.

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

You get a DataFrame containing only Boolean values, where True has only the 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
0   NaN    NaN    NaN
1   NaN    pen      1
2   NaN    NaN    NaN
3   NaN    NaN    NaN
4   NaN    NaN    NaN
```

Deleting a Column

If you want to delete an entire column with all its contents, then use the **del** command.

```
>>> del frame['new']
>>> frame
   colors  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
```

Filtering

Even for a DataFrame you can apply the filtering through the application of certain conditions, for example if you want to get all values smaller than a certain number, for example 12.

```
>>> frame[frame < 12]
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white    NaN  NaN     NaN    NaN
```

You will get as returned object a DataFrame containing values less than 12, 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 the one represented 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 so as to consider 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 find a successful match. pandas will compensate for this inconsistency by adding the value NaN values missing.

```
>>> 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     22
2013    18   33     16
```

Transposition of a DataFrame

An operation that might be needed when dealing with tabular data structures is the transposition (that is, the columns become rows and rows columns). pandas allows you to do this in a very simple way. You can get the transpose of the data frame by adding the **T** attribute to its application.

```
>>> frame2.T
       2011  2012  2013
blue     17    27    18
red     NaN    22    33
white    13    22    16
```

The Index Objects

Now that you know what the Series and the data frame are and how they are structured, you can certainly perceive the peculiarities of these data structures. Indeed, the majority of their excellent characteristics in the data analysis are due to the presence of an Index object totally integrated within 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 as an array containing labels is converted into an Index object: you need to specify the **index** option within the constructor.

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

Unlike all other elements within pandas data structures (Series and data frame), the Index objects are immutable objects. Once declared, these cannot be changed. This ensures their secure sharing between the various data structures.

Each Index object has a number of methods and properties especially 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 index from a data structure. For example, **idmin()** and **idmax()** are two functions that return, respectively, the index with the lowest value and more.

```
>>> ser.idxmin()
'red'
>>> ser.idxmax()
'green'
```

Index with Duplicate Labels

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

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 within a data structure, if in correspondence of the same label there are more values, 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 data frame with duplicate indexes that will return the data frame.

In the case of data structures with small size, it is easy to identify any duplicate indexes, but if the structure becomes gradually larger this starts to become difficult. Just 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 within them.

This choice has proven somewhat successful. In fact, despite the enormous flexibility given by the dynamic structures that already exist, the capability to use the internal reference to the structure, such as that offered by the labels, allows those who must perform operations to carry out in a much more simple and direct way a series of operations that you will see in this and the next chapter.

In this section you will analyze in detail a number of basic features that take advantage of this mechanism of the indexes.

- Reindexing
- Dropping
- Alignment

Reindexing

It was previously stated that once declared within 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 make the reindexing of 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 this operation of reindexing, it is therefore 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 add NaN as corresponding value.

```
>>> ser.reindex(['three','four','five','one'])
three     7
four      4
five    NaN
one       2
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 'five' is present in the Series.

However, to measure the reindexing, the definition of the list of all the labels can be awkward, especially for a large data frame. So you could use some method that allows you to fill 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 an interpolation in order to obtain the complete sequence of numbers. To achieve this you will use the reindexing with the **method** option set to **ffill**. Moreover, you need to set a range of values for indexes. In this case, for specifying a set of values between 0 and 5, you can use **range(6)** as 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, the addition of a new column or index is possible, but being missing values in the original data structure, pandas add NaN values to them.

```
>>> frame.reindex(range(5), method='ffill',columns=['colors','price','new','object'])
   colors price  new  object
0    blue   1.2  NaN    ballpand
1   green   1.0  NaN     pen
2  yellow   0.6  NaN  pencil
3     red   0.9  NaN   paper
4   white   1.7  NaN     mug
```

Dropping

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

Also in this case, pandas provides a specific function for this operation: **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 generic Series 4 elements with four distinct labels.

```
>>> ser = Series(np.arange(4.), index=['red','blue','yellow','white'])
>>> ser
red       0
blue      1
yellow    2
white     3
dtype: float64
```

Now, for example, 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
blue     1
white    3
dtype: float64
```

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

```
>>> ser.drop(['blue','white'])
red       0
yellow    2
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, 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 is able to perform the alignment of the indexes coming from two different data structures. This is especially true when you are performing an arithmetic operation between them. In fact, during these operations, not only may the indexes between the two structures be in a different order, but they also may 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 the alignment of the 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. Well, 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
brown    NaN
green    NaN
white      4
yellow     6
dtype: float64
```

In the case of the data frame, 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  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
```

Operations between Data Structures

Now that you have become 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 we 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'll 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 DataFrame 'frame1 + frame2', you'll have to use the following format:

```
>>> frame1.add(frame2)
        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
```

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 data frame 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 even between different structures as for example, a DataFrame and a Series. For example, we 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: int32
```

The two newly defined data structures have been created specifically so that the indexes of Series match with 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 data frame 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 pandas library functions

Functions by Element

The pandas library is built on the foundations of NumPy, and then extends many of its features adapting them to new data structures as Series and DataFrame. Among these are the **universal functions**, called **ufunc**. This class of functions is particular because it 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 within the data frame, 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 thing is that they operate on a one-dimensional array, giving a single number for result. For example, we 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 also in this way:

```
>>> 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, however, this time it is only one value for the column, but if you prefer to apply the function by row instead of by column, you have to specify the **axis** option set 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()** returns 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, apply the function as before. But in this case as an object returned you get a DataFrame and no longer 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

However, the majority of the statistical functions for arrays are still valid for DataFrame, so the use of 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
pen       7
pencil    8
paper     9
dtype: float64
```

There is also a function called **describe()** that allows to obtain a 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 makes use of the indexing is sorting. Sorting the data is often a necessity and it is very important to be able to do easily. Pandas provides the **sort_index()** function that returns a new object which is identical to the start, but in which the elements are ordered.

You 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: int64fra
>>> ser.sort_index()
blue      0
green     4
red       5
white     8
yellow    3
dtype: int64
```

As you can see the items were sorted in the alphabetical order of the labels in ascending order (from A to Z). This is the default behavior, but you can set the opposite order, using the **ascending** option set to False.

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

As regards 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, just continue to use the function **sort_index()** without arguments as you've seen before, or if you prefer to order by columns, you will need to use the **axis** options set 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 within 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 will use the **order()** function.

```
>>> ser.order()
blue      0
yellow    3
green     4
red       5
white     8
dtype: int64
```

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

```
>>> frame.sort_index(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 criteria of sorting 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_index(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 value.

```
>>> ser.rank()
red       4
blue      1
yellow    2
white     5
green     3
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, just add the **method** option with the 'first' value assigned.

```
>>> ser.rank(method='first')
red       4
blue      1
yellow    2
white     5
green     3
dtype: float64
```

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

```
>>> ser.rank(ascending=False)
red       2
blue      5
yellow    4
white     1
green     3
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.Seri
es([3,4,3,4,5,4,3,2],['2006','2007','2008','2009','2010','2011','2012','2013'])
>>> seq = pd.Seri
es([1,2,3,4,4,3,2,1],['2006','2007','2008','2009','2010','2011','2012','2013'])
>>> seq.corr(seq2)
0.77459666924148329
>>> seq.cov(seq2)
0.8571428571428571
```

Another case could be that covariance and correlation are applied to a single DataFrame. In this case, they return their corresponding matrices in form of two new DataFrame objects.

```
>>> frame2 = 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 method **corrwith()**, you can calculate the pairwise correlations between the columns or rows of a data frame with a Series or another DataFrame().

```
>>> serred       0
blue      1
yellow    2
white     3
green     9
dtype: float64
>>> 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

During the previous sections we have seen how easily the missing data can be formed. They are recognizable within the data structures with the value NaN (Not a Number). So, having values that are not defined in a data structure is a condition quite common for those who carry out 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. In fact, for example, within the pandas library, the calculation of descriptive statistics excludes NaN values implicitly.

Assigning a NaN Value

Just in case you would like to specifically assign a NaN value to an element in a data structure, you can use the value **np.NaN**(or np.nan) of the NumPy library.

```
>>> ser = pd.Series([0,1,2,np.NaN,9], index=['red','blue','yellow','white','green'])
>>> ser
red        0
blue       1
yellow     2
white    NaN
green      9
dtype: float64
>>> ser['white'] = None
>>> ser
red        0
blue       1
yellow     2
white    NaN
green      9
dtype: float64
```

Filtering Out NaN Values

There are various options to eliminate the NaN values during the data analysis. However, the elimination by hand, element by element, can be very tedious and risky, because you never get the certainty of having eliminated all the NaN values. The **dropna()** function comes to your aid.

```
>>> ser.dropna()
red       0
blue      1
yellow    2
green     9
dtype: float64
```

Another possibility is to directly perform the filtering function by placing the **notnull()** in the selection condition.

```
>>> ser[ser.notnull()]
red       0
blue      1
yellow    2
green     9
dtype: float64
```

If you're dealing with the DataFrame it gets a little more complex. If you use the **dropna()** function on this type of object, it is sufficient that there is only one value NaN on a column or a row to eliminate it completely.

```
>>> 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  NaN    6
green   NaN  NaN  NaN
red       2  NaN    5
>>> 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, in order to inform 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  NaN    6
red      2  NaN    5
```

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 could replace them with other numbers. For most purposes, the fillna() function could be a great choice. This method takes one argument, the value with which to replace any NaN. It can be the same for all, as in the following case.

```
>>> frame3.fillna(0)
       ball  mug  pen
blue      6    0    6
green     0    0    0
red       2    0    5
```

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

```
>>> frame3.fillna({'ball':1,'mug':0,'pen':99})
       ball  mug  pen
blue      6    0    6
green     1    0   99
red       2    0    5
```

Hierarchical Indexing and Leveling

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

You can 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
MultiIndex(levels=[[u'blue', u'red', u'white'], [u'down', u'left', u'right', u'up']],
           labels=[[2, 2, 2, 0, 0, 1, 1, 1], [3, 0, 2, 3, 0, 3, 0, 1]])ù
```

Through the specification of a hierarchical indexing, the selection of 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 we want to select a specific value, you will specify both indexes.

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

The hierarchical indexing plays a critical role in reshaping the data and group-based operations such as creating a pivot-table. For example, the data could be used just rearranged in a data frame using a special function called **unstack()**. This function converts the Series with hierarchical index in 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 we want is to perform the reverse operation, which is to convert a DataFrame in a Series, you will 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: int32
```

As regards the DataFrame, it is possible to define a hierarchical index both for the rows and for the columns. At the time of the declaration of the DataFrame, you have to define an array of arrays for both the **index** option and the **columns** option.

```
>>> 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 could need to rearrange the order of the levels on an axis or do a sorting for values at a specific level.

The **swaplevel()** function accepts as argument 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 **sortlevel()** function orders the data considering only those of a certain level.

```
>>> mframe.sortlevel('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 make 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 make a statistic for a given level of the column, for example, the **id**, you must specify the second axis as 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

In this chapter, the library pandas has been introduced. You have learned how to install it and then you have seen a general overview based on its characteristics.

In more detail, you saw the two basic structures data, called 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 some operations on them. Finally you looked at the possibility of extending the complexity of these structures creating hierarchies of indexes, thus distributing the data contained in them in different sub-levels.

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