CHAPTER 4

The pandas Library—An Introduction

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

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

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

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

pandas: The Python Data Analysis Library

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

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

© Fabio Nelli 2018
F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_4

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

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

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

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

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

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

Installation of pandas

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

Installation from Anaconda

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

```
conda list pandas
```

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

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

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

```
conda install pandas
```

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

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

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

   - pandas
```

The following new packages will be installed:

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

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

```
conda update pandas
```

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

Installation from PyPI

pandas can also be installed by PyPI using this command:

```
pip install pandas
```

Installation on Linux

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

On Debian and Ubuntu distributions, use this command:

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

While on OpenSuse and Fedora, enter the following command:

```
zypper in python-pandas
```

Installation from Source

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

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

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

A Module Repository for Windows

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

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

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

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

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

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

Testing Your pandas Installation

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

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

```
nosetests pandas
```

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

NOSE MODULE

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

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

Getting Started with pandas

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

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

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

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

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

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

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

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

Introduction to pandas Data Structures

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

- Series
- Dataframes

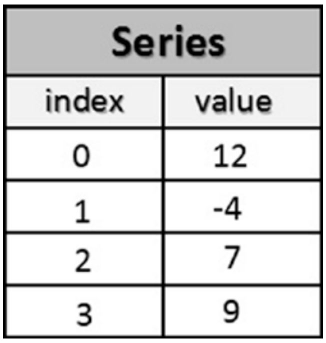
The *series*, as you will see, constitutes the data structure designed to accommodate a sequence of one-dimensional data, while the *dataframe*, a more complex data structure, is designed to contain cases with several dimensions.

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

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

The Series

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



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

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

Declaring a Series

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

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

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

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

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

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

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

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

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

Selecting the Internal Elements

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

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

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

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

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

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

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

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

Assigning Values to the Elements

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

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

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

Defining a Series from NumPy Arrays and Other Series

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

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

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

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

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

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

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

Filtering Values

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

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

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

Operations and Mathematical Functions

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

You can simply write the arithmetic expression for the operators.

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

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

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

Evaluating Vales

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

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

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

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

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

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

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

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

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

NaN Values

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

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

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

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

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

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

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

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

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

Series as Dictionaries

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

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

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

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

Operations Between Series

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

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

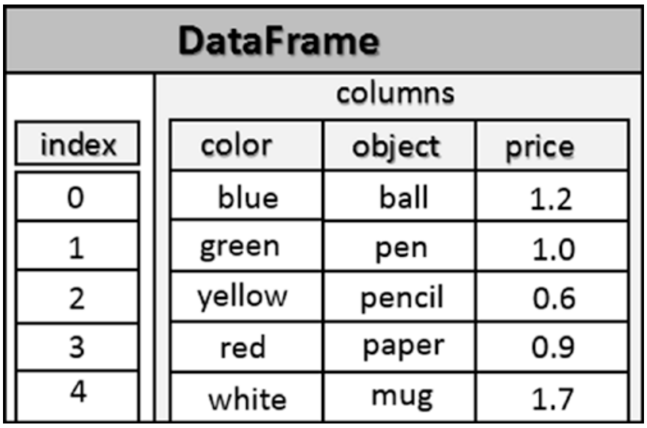
In the following example, you add two series having only some elements in common with the label.

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

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

The DataFrame

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



index	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

Figure 4-2. *The dataframe structure*

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

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

Defining a Dataframe

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

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

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

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

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

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

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

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

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

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

Selecting Elements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Assigning Values

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Membership of a Value

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

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

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

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

Deleting a Column

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

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

Filtering

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

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

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

DataFrame from Nested dict

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

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

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

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

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

Transposition of a Dataframe

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

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

The Index Objects

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

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

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

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

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

Methods on Index

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

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

Index with Duplicate Labels

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

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

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

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

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

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

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

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

Other Functionalities on Indexes

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

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

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

- Reindexing
- Dropping
- Alignment

Reindexing

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dropping

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Arithmetic and Data Alignment

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

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

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

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

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

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

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

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

Operations Between Data Structures

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

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

Flexible Arithmetic Methods

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

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

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

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

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

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

Operations Between DataFrame and Series

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

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

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

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

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

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

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

Function Application and Mapping

This section covers the pandas library functions.

Functions by Element

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

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

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

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

Functions by Row or Column

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

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

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

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

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

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

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

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

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

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

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

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

Statistics Functions

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

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

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

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

Sorting and Ranking

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Correlation and Covariance

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

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

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

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

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

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

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

"Not a Number" Data

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

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

Assigning a NaN Value

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

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

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

Filtering Out NaN Values

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

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

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

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

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

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

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

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

Filling in NaN Occurrences

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

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

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

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

Hierarchical Indexing and Leveling

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Reordering and Sorting Levels

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

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

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

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

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

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

Summary Statistic by Level

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

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

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

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

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

Conclusions

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

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

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