CHAPTER 6

pandas in Depth: Data Manipulation

In the previous chapter you have seen how to acquire data from data sources such as databases or files. Once you have the data in DataFrame format, they are ready to be manipulated. The manipulation of the data has the purpose of preparing the data so that they can be more easily subjected to analysis. In fact, their manipulation will depend a lot on purposes of those who must carry out the analysis, and it will be performed for making more explicit the information you are looking for. Especially in preparation for the next phase, the data must be ready to the data visualization that will follow in the next chapter.

In this chapter you will go in depth with the functionality that the pandas library offers for this stage of the data analysis. The three phases of data manipulation will be treated individually, illustrating the various operations with a series of examples and on how best to use the functions of this library for carrying out such operations. The three phases of data manipulation are

- Data preparation
- Data transformation
- Data aggregation

Data Preparation

Before you start manipulating data itself, it is necessary to prepare the data and assemble them in the form of data structures such that they can be manipulated later with the tools made available by the pandas library. The different procedures for data preparation are listed below.

- loading
- assembling
    - merging
    - concatenating
    - combining
- reshaping (pivoting)
- removing

As regards loading, all of the previous chapter is centered on this topic. In the loading phase, there is also that part of the preparation which concerns the conversion from many different formats into a data structure such as DataFrame. But even after you have gotten the data, probably from different sources and formats, and unified it into a DataFrame, you will need to perform further operations of preparation. In this chapter, and in particular in this section, you'll see how to perform all the operations necessary for the incorporation of data into a unified data structure.

The data contained in the pandas objects can be assembled together in different ways:

- Merging—the **pandas.merge( )** function connects the rows in a DataFrame based on one or more keys. This mode is very familiar to those who are confident with the SQL language, since it also implements join operations.
- Concatenating—the **pandas.concat( )** function concatenates the objects along an axis.
- Combining—the **pandas.DataFrame.combine_first( )** function is a method that allows you to connect overlapped data in order to fill in missing values in a data structure by taking data from another structure.

Furthermore, part of the preparation process is also pivoting, which consists of the exchange between rows and columns.

Merging

The merging operation, which corresponds to the JOIN operation for those who are familiar with SQL, consists of a combination of data through the connection of rows using one or more keys.

In fact, anyone working with relational databases usually makes use of the JOIN query with SQL to get data from different tables using some reference values (keys) shared between them. On the basis of these keys it is possible to obtain new data in a tabular form as the result of the combination of other tables. This operation with the library pandas is called **merging**, and **merge()** is the function to perform this kind of operation.

First, you have to import the pandas library and define two DataFrame that will serve you as examples for this section.

```
>>> import numpy as np
>>> import pandas as pd
>>> frame1 = pd.DataFrame( {'id':['ball','pencil','pen','mug','ashtray'],
...                         'price': [12.33,11.44,33.21,13.23,33.62]})
>>> frame1
        id  price
0     ball  12.33
1   pencil  11.44
2      pen  33.21
3      mug  13.23
4  ashtray  33.62
>>> frame2 = pd.DataFrame( {'id':['pencil','pencil','ball','pen'],
...                         'color': ['white','red','red','black']})
>>> frame2
   color      id
0  white  pencil
1    red  pencil
2    red    ball
3  black     pen
```

Carry out the merging applying the **merge( )** function to the two DataFrame objects.

```
>>> pd.merge(frame1,frame2)
       id  price  color
0    ball  12.33    red
1  pencil  11.44  white
2  pencil  11.44    red
3     pen  33.21  black
```

As you can see from the result, the returned DataFrame consists of all rows that have an **ID** in common between the two DataFeame. In addition to the common column, the columns from both the first and the second DataFrame are added.

In this case you used the **merge( )** function without specifying any column explicitly. In fact, in most cases you need to decide which is the column on which to base the merging.

To do this, add the **on** option with the column name as the key for the merging.

```
>>> frame1 = pd.DataFrame( {'id':['ball','pencil','pen','mug','ashtray'],
...                       'color': ['white','red','red','black','green'],
...                       'brand': ['OMG','ABC','ABC','POD','POD']})
>>> frame1
  brand  color       id
0   OMG  white     ball
1   ABC    red   pencil
2   ABC    red      pen
3   POD  black      mug
4   POD  green  ashtray
>>> frame2 = pd.DataFrame( {'id':['pencil','pencil','ball','pen'],
...                       'brand': ['OMG','POD','ABC','POD']})
>>> frame2
  brand      id
0   OMG  pencil
1   POD  pencil
2   ABC    ball
3   POD     pen
```

Now in this case you have two DataFrame having columns with the same name. So if you launch a merging you do not get any results.

```
>>> pd.merge(frame1,frame2)
Empty DataFrame
Columns: [brand, color, id]
Index: []
```

So it is necessary to explicitly define the criterion of merging that pandas must follow, specifying the name of the key column in the **on** option.

```
>>> pd.merge(frame1,frame2,on='id')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
1     ABC    red  pencil     OMG
2     ABC    red  pencil     POD
3     ABC    red     pen     POD
```

```
>>> pd.merge(frame1,frame2,on='brand')
  brand  color     id_x    id_y
0   OMG  white     ball  pencil
1   ABC    red   pencil    ball
2   ABC    red      pen    ball
3   POD  black      mug  pencil
4   POD  black      mug     pen
5   POD  green  ashtray  pencil
6   POD  green  ashtray     pen
```

As expected, the results vary considerably depending on the criteria of merging.

Often, however, the opposite problem arises, that is, to have two DataFrames in which the key columns do not have the same name. To remedy this situation, you have to use the **left_on** and **right_on** options that specify the key column for the first and for the second DataFrame. Now you can see an example.

```
>>> frame2.columns = ['brand','sid']
>>> frame2
  brand     sid
0   OMG  pencil
1   POD  pencil
2   ABC    ball
3   POD     pen
>>> pd.merge(frame1, frame2, left_on='id', right_on='sid')
  brand_x  color      id brand_y     sid
0     OMG  white    ball     ABC    ball
1     ABC    red  pencil     OMG  pencil
2     ABC    red  pencil     POD  pencil
3     ABC    red     pen     POD     pen
```

By default, the **merge( )** function performs an **inner join;** the keys in the result are the result of an intersection.

Other possible options are the **left join**, the **right join,** and the **outer join**. The outer join produces the union of all keys, combining the effect of a left join with a right join. To select the type of join you have to use the **how** option.

```
>>> frame2.columns['brand','id']
>>> pd.merge(frame1,frame2,on='id')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
1     ABC    red  pencil     OMG
2     ABC    red  pencil     POD
3     ABC    red     pen     POD
>>> pd.merge(frame1,frame2,on='id',how='outer')
  brand_x  color       id brand_y
0     OMG  white     ball     ABC
1     ABC    red   pencil     OMG
2     ABC    red   pencil     POD
3     ABC    red      pen     POD
4     POD  black      mug     NaN
5     POD  green  ashtray     NaN
```

```
>>> pd.merge(frame1,frame2,on='id',how='left')
  brand_x  color       id brand_y
0     OMG  white     ball     ABC
1     ABC    red   pencil     OMG
2     ABC    red   pencil     POD
3     ABC    red      pen     POD
4     POD  black      mug     NaN
5     POD  green  ashtray     NaN
>>> pd.merge(frame1,frame2,on='id',how='right')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
1     ABC    red  pencil     OMG
2     ABC    red  pencil     POD
3     ABC    red     pen     POD
```

To make the merge of multiple keys, you simply just add a list to the **on** option.

```
>>> pd.merge(frame1,frame2,on=['id','brand'],how='outer')
  brand  color       id
0   OMG  white     ball
1   ABC    red   pencil
2   ABC    red      pen
3   POD  black      mug
4   POD  green  ashtray
5   OMG    NaN   pencil
6   POD    NaN   pencil
7   ABC    NaN     ball
8   POD    NaN      pen
```

Merging on Index

In some cases, instead of considering the columns of a DataFrame as keys, the indexes could be used as keys on which to make the criteria for merging. Then in order to decide which indexes to consider, set the **left_index** or **right_index** options to True to activate them, with the ability to activate them both.

```
>>> pd.merge(frame1,frame2,right_index=True, left_index=True)
  brand_x  color    id_x brand_y    id_y
0     OMG  white    ball     OMG  pencil
1     ABC    red  pencil     POD  pencil
2     ABC    red     pen     ABC    ball
3     POD  black     mug     POD     pen
```

But the DataFrame objects have a **join()** function which is much more convenient when you want to do the merging by indexes. It can also be used to combine many DataFrame objects having the same or the same indexes but with columns not overlapping.

```
In fact, if you launch

>>> frame1.join(frame2)
```

You will get an error code because some columns of the frame1 have the same name of frame2. Then rename the columns of frame2 before launching the **join( )** function.

```
>>> frame2.columns = ['brand2','id2']
>>> frame1.join(frame2)
  brand  color       id brand2     id2
0   OMG  white     ball    OMG  pencil
1   ABC    red   pencil    POD  pencil
2   ABC    red      pen    ABC    ball
3   POD  black      mug    POD     pen
4   POD  green  ashtray    NaN     NaN
```

Here you've performed a merging but based on the values of the indexes instead of the columns. This time there is also the index 4 that was present only in frame1, but the values corresponding to the columns of frame2 report NaN as value.

Concatenating

Another type of data combination is referred to as **concatenation**. NumPy provides a **concatenate()** function to do this kind of operation with arrays.

```
>>> array1
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> array2 = np.arange(9).reshape((3,3))+6
>>> array2
array([[ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
>>> np.concatenate([array1,array2],axis=1)
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11],
       [ 6,  7,  8, 12, 13, 14]])
>>> np.concatenate([array1,array2],axis=0)
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

As regards the pandas library and its data structures like Series and DataFrame, the fact of having labeled axes allows you to further generalize the concatenation of arrays. The **concat()** function is provided by pandas for this kind of operation.

```
>>> ser1 = pd.Series(np.random.rand(4), index=[1,2,3,4])
>>> ser1
1    0.636584
2    0.345030
3    0.157537
4    0.070351
dtype: float64
>>> ser2 = pd.Series(np.random.rand(4), index=[5,6,7,8])
>>> ser2
5    0.411319
6    0.359946
7    0.987651
8    0.329173
dtype: float64
>>> pd.concat([ser1,ser2])
1    0.636584
2    0.345030
3    0.157537
4    0.070351
5    0.411319
6    0.359946
7    0.987651
8    0.329173
dtype: float64
```

By default, the **concat()** function works on axis = 0, having as returned object a Series. If you set the axis = 1, then the result will be a DataFrame.

```
>>> pd.concat([ser1,ser2],axis=1)
          0         1
1  0.636584       NaN
2  0.345030       NaN
3  0.157537       NaN
4  0.070351       NaN
5       NaN  0.411319
6       NaN  0.359946
7       NaN  0.987651
8       NaN  0.329173
```

From the result you can see that there is no overlap of data, therefore what you have just done is an outer join. This can be changed by setting the **join** option to 'inner'.

```
>>> pd.concat([ser1,ser3],axis=1,join='inner')
          0         0   1
1  0.636584  0.636584 NaN
2  0.345030  0.345030 NaN
3  0.157537  0.157537 NaN
4  0.070351  0.070351 NaN
```

A problem in this kind of operation is that the concatenated parts are not identifiable in the result. For example, you want to create a hierarchical index on the axis of concatenation. To do this you have to use the **keys** option.

```
>>> pd.concat([ser1,ser2], keys=[1,2])
1  1    0.636584
   2    0.345030
   3    0.157537
   4    0.070351
2  5    0.411319
   6    0.359946
   7    0.987651
   8    0.329173
dtype: float64
```

In the case of combinations between Series along the axis = 1 the keys become the column headers of the DataFrame.

```
>>> pd.concat([ser1,ser2], axis=1, keys=[1,2])
          1         2
1  0.636584       NaN
2  0.345030       NaN
3  0.157537       NaN
4  0.070351       NaN
5       NaN  0.411319
6       NaN  0.359946
7       NaN  0.987651
8       NaN  0.329173
```

So far you have seen the concatenation applied to the Series, but the same logic can be applied to the DataFrame.

```
>>> frame1 = pd.DataFrame(np.random.rand(9).reshape(3,3), index=[1,2,3],
columns=['A','B','C'])
>>> frame2 = pd.DataFrame(np.random.rand(9).reshape(3,3), index=[4,5,6],
columns=['A','B','C'])
>>> pd.concat([frame1, frame2])
          A         B         C
1  0.400663  0.937932  0.938035
2  0.202442  0.001500  0.231215
3  0.940898  0.045196  0.723390
4  0.568636  0.477043  0.913326
5  0.598378  0.315435  0.311443
6  0.619859  0.198060  0.647902

>>> pd.concat([frame1, frame2], axis=1)
          A         B         C         A         B         C
1  0.400663  0.937932  0.938035       NaN       NaN       NaN
2  0.202442  0.001500  0.231215       NaN       NaN       NaN
3  0.940898  0.045196  0.723390       NaN       NaN       NaN
4       NaN       NaN       NaN  0.568636  0.477043  0.913326
5       NaN       NaN       NaN  0.598378  0.315435  0.311443
6       NaN       NaN       NaN  0.619859  0.198060  0.647902
```

Combining

There is another situation in which there is combination of data that cannot be obtained either with merging or with concatenation. Take the case in which you want the two datasets to have indexes that overlap in their entirety or at least partially.

One applicable function to Series is **combine_first()**, which performs this kind of operation along with an data alignment.

```
>>> ser1 = pd.Series(np.random.rand(5),index=[1,2,3,4,5])
>>> ser1
1    0.942631
2    0.033523
3    0.886323
4    0.809757
5    0.800295
dtype: float64
>>> ser2 = pd.Series(np.random.rand(4),index=[2,4,5,6])
>>> ser2
2    0.739982
4    0.225647
5    0.709576
6    0.214882
dtype: float64
>>> ser1.combine_first(ser2)
1    0.942631
2    0.033523
3    0.886323
4    0.809757
5    0.800295
6    0.214882
dtype: float64
>>> ser2.combine_first(ser1)
1    0.942631
2    0.739982
3    0.886323
4    0.225647
5    0.709576
6    0.214882
dtype: float64
```

Instead, if you want a partial overlap, you can specify only the portion of the Series you want to overlap.

```
>>> ser1[:3].combine_first(ser2[:3])
1    0.942631
2    0.033523
3    0.886323
4    0.225647
5    0.709576
dtype: float64
```

Pivoting

In addition to assembling the data in order to unify the values collected from different sources, another fairly common operation is **pivoting**. In fact, arrangement of the values by row or by column is not always suited to your goals. Sometimes you would like to rearrange the data carrying column values on rows or vice versa.

Pivoting with Hierarchical Indexing

You have already seen that DataFrame can support hierarchical indexing. This feature can be exploited to rearrange the data in a DataFrame. In the context of pivoting you have two basic operations:

- stacking: rotates or pivots the data structure converting columns to rows
- unstacking: converts rows into columns

```
>>> frame1 = pd.DataFrame(np.arange(9).reshape(3,3),
...                     index=['white','black','red'],
...                     columns=['ball','pen','pencil'])
>>> frame1
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8
```

Using the **stack()** function on the DataFrame, you will get the pivoting of the columns in rows, thus producing a Series:

```
>>> frame1.stack()
white  ball      0
       pen       1
       pencil    2
black  ball      3
       pen       4
       pencil    5
red    ball      6
       pen       7
       pencil    8
dtype: int32
```

From this hierarchically indexed series, you can reassemble the DataFrame into a pivoted table by use of the `unstack()` function.

```
>>> ser5.unstack()
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8
```

You can also do the unstack on a different level, specifying the number of levels or its name as the argument of the function.

```
>>> ser5.unstack(0)
        white  black  red
ball        0      3    6
pen         1      4    7
pencil      2      5    8
```

Pivoting from “Long” to “Wide” Format

The most common way to store data sets is produced by the punctual registration of data that will fill a line of the text file, for example, CSV, or a table of a database. This happens especially when you have instrumental readings, calculation results iterated over time, or the simple manual input of a series of values. A similar case of these files is for example the logs file, which is filled line by line by accumulating data in it.

The peculiar characteristic of this type of data set is to have entries on various columns, often duplicated in subsequent lines. Always remaining in tabular format of data, when you are in such cases you can refer them to as **long** or **stacked** format.

To get a clearer idea about that, for example, consider the following DataFrame.

```
>>> longframe = pd.DataFrame({ 'color':['white','white','white',
...                                      'red','red','red',
...                                      'black','black','black'],
...                            'item':['ball','pen','mug',
...                                     'ball','pen','mug',
...                                     'ball','pen','mug'],
...                            'value': np.random.rand(9)})
>>> longframe
   color  item     value
0  white  ball  0.091438
1  white   pen  0.495049
2  white   mug  0.956225
3    red  ball  0.394441
4    red   pen  0.501164
5    red   mug  0.561832
6  black  ball  0.879022
7  black   pen  0.610975
8  black   mug  0.093324
```

This mode of data recording, however, has some disadvantages. One, for example, is precisely the multiplicity and repetition of some fields. Considering the columns as keys, the data with this format will be difficult to read, especially in fully understanding the relationships between the key values and the rest of the columns.

Instead of the long format, there is another way to arrange the data in a table that is called **wide**. This mode is easier to read, allowing easy connection with other tables, and it occupies much less space. So in general it is a more efficient way of storing the data, although less practical, especially if during the filling of the data.

As a criterion, select a column, or a set of them, as the primary key; then, the values contained in it must be unique.

In this regard, pandas gives you a function that allows you to make a transformation of a DataFrame from the long type to the wide type. This function is **pivot()** and it accepts as arguments the column, or columns, which will assume the role of key.

Starting from the previous example you choose to create a DataFrame in wide format by choosing the **color** column as the key, and **item** as a second key, the values of which will form the new columns of the data frame.

```
>>> wideframe = longframe.pivot('color','item')
>>> wideframe
          value
item       ball       mug       pen
color
black  0.879022  0.093324  0.610975
red    0.394441  0.561832  0.501164
white  0.091438  0.956225  0.495049
```

As you can now see, in this format, the DataFrame is much more compact and data contained in it are much more readable.

Removing

The last stage of data preparation is the removal of columns and rows. You have already seen this part in Chapter 4. However, for completeness, the description is reiterated here. Define a DataFrame by way of example.

```
>>> frame1 = pd.DataFrame(np.arange(9).reshape(3,3),
...                     index=['white','black','red'],
...                     columns=['ball','pen','pencil'])
>>> frame1
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8
```

In order to remove a column, you have to simply use the **del** command applied to the DataFrame with the column name specified.

```
>>> del frame1['ball']
>>> frame1
       pen  pencil
white    1       2
black    4       5
red      7       8
```

Instead, to remove an unwanted row, you have to use the **drop()** function with the label of the corresponding index as argument.

```
>>> frame1.drop('white')
       pen  pencil
black    4       5
red      7       8
```

Data Transformation

So far you have seen how to prepare data for analysis. This process in effect represents a reassembly of the data contained within a DataFrame, with possible additions by other DataFrame and removal of unwanted parts.

Now begin with the second stage of data manipulation: the **data transformation**. After you arrange the form of data and their disposal within the data structure, it is important to transform their values. In fact, in this section you will see some common issues and the steps required to overcome them using functions of the pandas library.

Some of these operations involve the presence of duplicate or invalid values, with possible removal or replacement. Other operations relate instead modifying the indexes. Other steps include handling and processing the numerical values of the data and also of strings.

Removing Duplicates

Duplicate rows might be present in a DataFrame for various reasons. In DataFrames of enormous size the detection of these rows can be very problematic. Also in this case, pandas provides us with a series of tools to analyze the duplicate data present in large data structures.

First, create a simple DataFrame with some duplicate rows.

```
>>> dframe = pd.DataFrame({ 'color': ['white','white','red','red','white'],
...                       'value': [2,1,3,3,2]})
>>> dframe
   color  value
0  white      2
1  white      1
2    red      3
3    red      3
4  white      2
```

The **duplicated()** function applied to a DataFrame can detect the rows which appear to be duplicated. It returns a Series of Booleans where each element corresponds to a row, with **True** if the row is duplicated (i.e., only the other occurrences, not the first), and with **False** if there are no duplicates in the previous elements.

```
>>> dframe.duplicated()
0    False
1    False
2    False
3     True
4     True
dtype: bool
```

The fact of having as the return value a Boolean Series can be useful in many cases, especially for the filtering. In fact, if you want to know what are the duplicate rows, just type the following:

```
>>> dframe[dframe.duplicated()]
   color  value
3    red      3
4  white      2
```

Generally, all duplicated rows are to be deleted from the DataFrame; to do that, pandas provides the **drop_duplicates()** function, which returns the DataFrame without duplicate rows.

```
>>> dframe[dframe.duplicated()]
   color  value
3    red      3
4  white      2
```

Mapping

The pandas library provides a set of functions which, as you shall see in this section, exploit mapping to perform some operations. The mapping is nothing more than the creation of a list of matches between two different values, with the ability to bind a value to a particular label or string.

To define a mapping there is no better object than dict objects.

```
map = {
   'label1' : 'value1,
   'label2' : 'value2,
   ...
}
```

The functions that you will see in this section perform specific operations but all of them are united from accepting a dict object with matches as an argument.

- **replace():** replaces values
- **map():** creates a new column
- **rename():** replaces the index values

Replacing Values via Mapping

Often in the data structure that you have assembled there are values that do not meet your needs. For example, the text may be in a foreign language, or may be a synonym of another value, or may not be expressed in the desired shape. In such cases, a replace operation of various values is often a necessary process.

Define, as an example, a DataFrame containing various objects and colors, including two colors that are not in English. Often during the assembly operations is likely to keep maintaining data with values in a form that is not desired.

```
>>> frame = pd.DataFrame({ 'item':['ball','mug','pen','pencil','ashtray'],
...                     'color':['white','rosso','verde','black','yellow'],
                        'price':[5.56,4.20,1.30,0.56,2.75]})
>>> frame
    color    item
0   white    ball
1     red     mug
2   green     pen
3   black  pencil
4  yellow ashtray
```

Thus to be able to replace the incorrect values in new values is necessary to define a mapping of correspondences, containing as key to replace the old values and values as the new ones.

```
>>> newcolors = {
...    'rosso': 'red',
...    'verde': 'green'
... }
```

Now the only thing you can do is to use the **replace()** function with the mapping as an argument.

```
>>> frame.replace(newcolors)
    color     item  price
0   white     ball   5.56
1     red      mug   4.20
2   green      pen   1.30
3   black   pencil   0.56
4  yellow  ashtray   2.75
```

As you can see from the result, the two colors have been replaced with the correct values within the DataFrame. A common case, for example, is the replacement of the NaN values with another value, for example 0. Also here you can use the **replace(),** which performs its job very well.

```
>>> ser = pd.Series([1,3,np.nan,4,6,np.nan,3])
>>> ser
0      1
1      3
2    NaN
3      4
4      6
5    NaN
6      3
dtype: float64
>>> ser.replace(np.nan,0)
0    1
1    3
2    0
3    4
4    6
5    0
6    3
dtype: float64
```

Adding Values via Mapping

In the previous example, you have seen the case of the substitution of values through a mapping of correspondences. In this case you continue to exploit the mapping of values with another example. In this case you are exploiting mapping to add values in a column depending on the values contained in another. The mapping will always be defined separately.

```
>>> frame = pd.DataFrame({ 'item':['ball','mug','pen','pencil','ashtray'],
...                       'color':['white','red','green','black','yellow']})
>>> frame
    color     item
0   white     ball
1     red      mug
2   green      pen
3   black   pencil
4  yellow  ashtray
```

Let's suppose you want to add a column to indicate the price of the item shown in the DataFrame. Before you do this, it is assumed that you have a price list available somewhere, in which the price for each type of item is described. Define then a dict object that contains a list of prices for each type of item.

```
>>> price = {
...    'ball' : 5.56,
...    'mug' : 4.20,
...    'bottle' : 1.30,
...    'scissors' : 3.41,
...    'pen' : 1.30,
...    'pencil' : 0.56,
...    'ashtray' : 2.75
... }
```

The **map()** function applied to a Series or to a column of a DataFrame accepts a function or an object containing a dict with mapping. So in your case you can apply the mapping of the prices on the column item, making sure to add a column to the price data frame.

```
>>> frame['price'] = frame['item'].map(prices)
>>> frame
    color     item  price
0   white     ball   5.56
1     red      mug   4.20
2   green      pen   1.30
3   black   pencil   0.56
4  yellow  ashtray   2.75
```

Rename the Indexes of the Axes

In a manner very similar to what you saw for the values contained within the Series and the DataFrame, even the axis label can be transformed in a very similar way using the mapping. So to replace the label indexes, pandas provides the **rename()** function, which takes the mapping as argument, that is, a dict object.

```
>>> frame
    color     item  price
0   white     ball   5.56
1     red      mug   4.20
2   green      pen   1.30
3   black   pencil   0.56
4  yellow  ashtray   2.75
```

```
>>> reindex = {
...   0: 'first',
...   1: 'second',
...   2: 'third',
...   3: 'fourth',
...   4: 'fifth'}
>>> frame.rename(reindex)
         color     item  price
first    white     ball   5.56
second     red      mug   4.20
third    green      pen   1.30
fourth   black   pencil   0.56
fifth   yellow  ashtray   2.75
```

As you can see, by default, the indexes are renamed. If you want to rename columns you must use the **columns** option. Thus this time you assign various mapping explicitly to the two **index** and **columns** options.

```
>>> recolumn = {
...    'item':'object',
...    'price': 'value'}
>>> frame.rename(index=reindex, columns=recolumn)
         color   object  value
first    white     ball   5.56
second     red      mug   4.20
third    green      pen   1.30
fourth   black   pencil   0.56
fifth   yellow  ashtray   2.75
```

Also here, for the simplest cases in which you have a single value to be replaced, it can further explicate the arguments passed to the function of avoiding having to write and assign many variables.

```
>>> frame.rename(index={1:'first'}, columns={'item':'object'})
        color   object  price
0       white     ball   5.56
first     red      mug   4.20
2       green      pen   1.30
3       black   pencil   0.56
4      yellow  ashtray   2.75
```

So far you have seen that the **rename()** function returns a DataFrame with the changes, leaving unchanged the original DataFrame. If you want the changes to take effect on the object on which you call the function, you will set the **inplace** option to True.

```
>>> frame.rename(columns={'item':'object'}, inplace=True)
>>> frame
    color   object  price
0   white     ball   5.56
1     red      mug   4.20
2   green      pen   1.30
3   black   pencil   0.56
4  yellow  ashtray   2.75
```

Discretization and Binning

A more complex process of transformation that you will see in this section is **discretization**. Sometimes it can happen, especially in some experimental cases, to handle large quantities of data generated in sequence. To carry out an analysis of the data, however, it is necessary to transform this data into discrete categories, for example, by dividing the range of values of such readings in smaller intervals and counting the occurrence or statistics within each of them. Another case might be to have a huge amount of samples due to precise readings on a population. Even here, to facilitate analysis of the data it is necessary to divide the range of values into categories and then analyze the occurrences and statistics related to each of them.

In your case, for example, you may have a reading of an experimental value between 0 and 100. These data are collected in a list.

```
>>> results = [12,34,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

You know that the experimental values have a range from 0 to 100; therefore you can uniformly divide this interval, for example, into four equal parts, i.e., bins. The first contains the values between 0 and 25, the second between 26 and 50, the third between 51 and 75, and the last between 76 and 100.

To do this binning with pandas, first you have to define an array containing the values of separation of bin:

```
>>> bins = [0,25,50,75,100]
```

Then there is a special function called **cut()** and apply it to the array of results also passing the bins.

```
>>> cat = pd.cut(results, bins)
>>> cat
   (0, 25]
  (25, 50]
  (50, 75]
  (50, 75]
  (25, 50]
 (75, 100]
 (75, 100]
   (0, 25]
   (0, 25]
  (50, 75]
  (50, 75]
  (25, 50]
 (75, 100]
   (0, 25]
  (25, 50]
 (75, 100]
 (75, 100]
Levels (4): Index(['(0, 25]', '(25, 50]', '(50, 75]', '(75, 100]'], dtype=object)
```

The object returned by the **cut()** function is a special object of **Categorical** type. You can consider it as an array of strings indicating the name of the bin. Internally it contains a **levels** array indicating the names of the different internal categories and a **labels** array that contains a list of numbers equal to the elements of **results** (i.e., the array subjected to binning). The number corresponds to the bin to which the corresponding element of **results** is assigned.

```
>>> cat.levels
Index([u'(0, 25]', u'(25, 50]', u'(50, 75]', u'(75, 100]'], dtype='object')
>>> cat.labels
array([0, 1, 2, 2, 1, 3, 3, 0, 0, 2, 2, 1, 3, 0, 1, 3, 3], dtype=int64)
```

Finally to know the occurrences for each bin, that is, how many results fall into each category, you have to use the value_**counts()** function.

```
>>> pd.value_counts(cat)
(75, 100]    5
(0, 25]      4
(25, 50]     4
(50, 75]     4
dtype: int64
```

As you can see, each class has the lower limit with a bracket and the upper limit with a parenthesis. This notation is consistent with mathematical notation that is used to indicate the intervals. If the bracket is square, the number belongs to the range (limit closed), and if it is round the number does not belong to the interval (limit open).

You can give names to various bins by calling them first in an array of strings and then assigning to the labels options inside the cut() function that you have used to create the Categorical object.

```
>>> bin_names = ['unlikely','less likely','likely','highly likely']
>>> pd.cut(results, bins, labels=bin_names)
      unlikely
   less likely
        likely
        likely
   less likely
 highly likely
 highly likely
      unlikely
      unlikely
        likely
        likely
   less likely
 highly likely
      unlikely
   less likely
 highly likely
 highly likely
Levels (4): Index(['unlikely', 'less likely', 'likely', 'highly likely'], dtype=object)
```

If the **cut()** function is passed as an argument to an integer instead of explicating the bin edges, this will divide the range of values of the array in many intervals as specified by the number.

The limits of the interval will be taken by the minimum and maximum of the sample data, namely, the array subjected to binning.

```
>>> pd.cut(results, 5)
 (2.904, 22.2]
  (22.2, 41.4]
  (60.6, 79.8]
  (41.4, 60.6]
  (22.2, 41.4]
    (79.8, 99]
    (79.8, 99]
 (2.904, 22.2]
 (2.904, 22.2]
  (41.4, 60.6]
  (60.6, 79.8]
  (41.4, 60.6]
    (79.8, 99]
  (22.2, 41.4]
  (41.4, 60.6]
    (79.8, 99]
    (79.8, 99]
Levels (5): Index(['(2.904, 22.2]', '(22.2, 41.4]', '(41.4, 60.6]',
                   '(60.6, 79.8]', '(79.8, 99]'], dtype=object)
```

In addition to **cut()**, pandas provides another method for binning: **qcut()**. This function divides the sample directly into quintiles. In fact, depending on the distribution of the data sample, using **cut()** rightly you will have a different number of occurrences for each bin. Instead **qcut()** will ensure that the number of occurrences for each bin is equal, but the edges of each bin to vary.

```
>>> quintiles = pd.qcut(results, 5)
>>> quintiles
    [3, 24]
   (24, 46]
 (62.6, 87]
 (46, 62.6]
   (24, 46]
   (87, 99]
   (87, 99]
    [3, 24]
    [3, 24]
 (46, 62.6]
 (62.6, 87]
   (24, 46]
 (62.6, 87]
    [3, 24]
 (46, 62.6]
   (87, 99]
 (62.6, 87]
Levels (5): Index(['[3, 24]', '(24, 46]', '(46, 62.6]', '(62.6, 87]',
                   '(87, 99]'], dtype=object)
```

```
>>> pd.value_counts(quintiles)
[3, 24]       4
(62.6, 87]    4
(87, 99]      3
(46, 62.6]    3
(24, 46]      3
dtype: int64
```

As you can see, in the case of quintiles, the intervals bounding the bin differ from those generated by the **cut()** function. Moreover, if you look at the occurrences for each bin will find that **qcut()** tried to standardize the occurrences for each bin, but in the case of quintiles, the first two bins have an occurrence in more because the number of results is not divisible by five.

Detecting and Filtering Outliers

During the data analysis, the need to detect the presence of abnormal values within a data structure often arises. By way of example, create a DataFrame with three columns from 1,000 completely random values:

```
>>> randframe = pd.DataFrame(np.random.randn(1000,3))
```

With the **describe()** function you can see the statistics for each column.

```
>>> randframe.describe()
                 0            1            2
count  1000.000000  1000.000000  1000.000000
mean      0.021609    -0.022926    -0.019577
std       1.045777     0.998493     1.056961
min      -2.981600    -2.828229    -3.735046
25%      -0.675005    -0.729834    -0.737677
50%       0.003857    -0.016940    -0.031886
75%       0.738968     0.619175     0.718702
max       3.104202     2.942778     3.458472
```

For example, you might consider outliers those that have a value greater than three times the standard deviation. To have only the standard deviation of each column of the DataFrame, use the **std()** function.

```
>>> randframe.std()
0    1.045777
1    0.998493
2    1.056961
dtype: float64
```

Now you apply the filtering of all the values of the DataFrame, applying the corresponding standard deviation for each column. Thanks to the **any()** function, you can apply the filter on each column.

```
>>> randframe[(np.abs(randframe) > (3*randframe.std())).any(1)]
            0         1         2
69  -0.442411 -1.099404  3.206832
576 -0.154413 -1.108671  3.458472
907  2.296649  1.129156 -3.735046
```

Permutation

The operations of permutation (random reordering) of a Series or the rows of a DataFrame are easy to do using the **numpy.random.permutation()** function.

For this example, create a DataFrame containing integers in ascending order.

```
>>> nframe = pd.DataFrame(np.arange(25).reshape(5,5))
>>> nframe
    0   1   2   3   4
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
4  20  21  22  23  24
```

Now create an array of five integers from 0 to 4 arranged in random order with the **permutation()** function. This will be the new order in which to set the values of a row of DataFrame.

```
>>> new_order = np.random.permutation(5)
>>> new_order
array([2, 3, 0, 1, 4])
```

Now apply it to the DataFrame on all lines, using the **take()** function.

```
>>> nframe.take(new_order)
    0   1   2   3   4
2  10  11  12  13  14
3  15  16  17  18  19
0   0   1   2   3   4
1   5   6   7   8   9
4  20  21  22  23  24
```

As you can see, the order of the rows has been changed; now the indices follow the same order as indicated in the **new_order** array.

You can submit even a portion of the entire DataFrame to a permutation. It generates an array that has a sequence limited to a certain range, for example, in our case from 2 to 4.

```
>>> new_order = [3,4,2]
>>> nframe.take(new_order)
    0   1   2   3   4
3  15  16  17  18  19
4  20  21  22  23  24
2  10  11  12  13  14
```

Random Sampling

You have just seen how to extract a portion of the DataFrame determined by subjecting it to permutation. Sometimes, when you have a huge DataFrame, you may have the need to sample it randomly, and the quickest way to do this is by using the **np.random.randint()** function.

```
>>> sample = np.random.randint(0, len(nframe), size=3)
>>> sample
array([1, 4, 4])
>>> nframe.take(sample)
    0   1   2   3   4
1   5   6   7   8   9
4  20  21  22  23  24
4  20  21  22  23  24
```

As you can see from this random sampling you can get the same sample even more times.

String Manipulation

Python is a popular language thanks to its ease of use in the processing of strings and text. Most operations can easily be made by using built-in functions provided by Python. For more complex cases of matching and manipulation, it is necessary the use of regular expressions.

Built-in Methods for Manipulation of Strings

In many cases you have composite strings in which you would like to separate the various parts and then assign them to the correct variables. The **split()** function allows us to separate parts of a text, taking as a reference point a separator, for example a comma.

```
>>> text = '16 Bolton Avenue , Boston'
>>> text.split(',')
['16 Bolton Avenue ', 'Boston']
```

As we can see in the first element, you have a string with a space character at the end. To overcome this problem and often a frequent problem, you have to use the **split()** function along with the **strip()** function that takes care of doing the trim of whitespace (including newlines).

```
>>> tokens = [s.strip() for s in text.split(',')]
>>> tokens
['16 Bolton Avenue', 'Boston']
```

The result is an array of strings. If the number of elements is small and always the same, a very interesting way to make assignments may be this:

```
>>> address, city = [s.strip() for s in text.split(',')]
>>> address
'16 Bolton Avenue'
>>> city
'Boston'
```

So far you have seen how to split a text into parts, but often you also need the opposite, namely concatenating various strings between them to form a more extended text.

The most intuitive and simple way is to concatenate the various parts of the text with the operator '+'.

```
>>> address + ',' + city
'16 Bolton Avenue, Boston'
```

This can be useful when you have only two or three strings to be concatenated. If the parts to be concatenated are much more, a more practical approach in this case will be to use the **join()** function assigned to the separator character, with which you want to join the various strings between them.

```
>>> strings = ['A+','A','A-','B','BB','BBB','C+']
>>> ';'.join(strings)
'A+;A;A-;B;BB;BBB;C+'
```

Another category of operations that can be performed on the string is the search for pieces of text in them, i.e., substrings. Python provides, in this respect, the keyword which represents the best way of detecting substrings.

```
>>> 'Boston' in text
True
```

However, there are two functions that could serve to this purpose: **index()** and **find()**.

```
>>> text.index('Boston')
19
>>> text.find('Boston')
19
```

In both cases, it returns the number of the corresponding character in the text where you have the substring. The difference in the behavior of these two functions can be seen, however, when the substring is not found:

```
>>> text.index('New York')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> text.find('New York')
-1
```

In fact, the **index()** function returns an error message, and **find()** returns -1 if the substring is not found. In the same area, you can know how many times a character or combination of characters (substring) occurs within a text. The **count()** function provides you with this number.

```
>>> text.count('e')
2
>>> text.count('Avenue')
1
```

Another operation that can be performed on strings is the replacement or elimination of a substring (or a single character). In both cases you will use the **replace()** function, where if you are prompted to replace a substring with a blank character, the operation will be equivalent to the elimination of the substring from the text.

```
>>> text.replace('Avenue','Street')
'16 Bolton Street , Boston'
>>> text.replace('1','')
'16 Bolton Avenue, Boston'
```

Regular Expressions

The regular expressions provide a very flexible way to search and match string patterns within a text. A single expression, generically called **regex**, is a string formed according to the regular expression language. There is a built-in Python module called **re**, which is responsible for the operation of the regex.

So first of all, when you want to make use of regular expressions, you will need to import the module.

```
>>> import re
```

The **re** module provides a set of functions that can be divided into three different categories:

- pattern matching
- substitution
- splitting

Now you start with a few examples. For example, the regex for expressing a sequence of one or more whitespace characters is **\s+**. As you saw in the previous section, to split a text into parts through a separator character you used the **split()**. There is a **split()** function even for the **re** module that performs the same operations, only it is able to accept a regex pattern as the criterion of separation, which makes it considerably more flexible.

```
>>> text = "This is      an\t odd  \n text!"
>>> re.split('\s+', text)
['This', 'is', 'an', 'odd', 'text!']
```

But analyze more deeply the mechanism of **re** module. When you call the **re.split()** function, the regular expression is first compiled, then subsequently calls the **split()** function on the text argument. You can compile the regex function with the **re.compile()** function, thus obtaining a reusable object regex and so gaining in terms of CPU cycles.

This is especially true in the operations of iterative search of a substring in a set or an array of strings.

```
>>> regex = re.compile('\s+')
```

So if you make an **regex** object with the **compile()** function, you can apply **split()** directly to it in the following way.

```
>>> regex.split(text)
['This', 'is', 'an', 'odd', 'text!']
```

As regards matching a regex pattern with any other business substrings in the text, you can use the **findall()** function. It returns a list of all the substrings in the text that meet the requirements of the regex.

For example, if you want to find in a string all the words starting with “A” uppercase, or for example, with “a” regardless whether upper- or lowercase, you need to enter what follows:

```
>>> text = 'This is my address: 16 Bolton Avenue, Boston'
>>> re.findall('A\w+',text)
['Avenue']
>>> re.findall('[A,a]\w+',text)
['address', 'Avenue']
```

There are two other functions related to the function **findall()**: **match()** and **search()**. While **findall()** returns all matches within a list, the function **search()** returns only the first match. Furthermore, the object returned by this function is a particular object:

```
>>> re.search('[A,a]\w+',text)
<_sre.SRE_Match object at 0x0000000007D7ECC8>
```

This object does not contain the value of the substring that responds to the regex pattern, but its start and end positions within the string.

```
>>> search = re.search('[A,a]\w+',text)
>>> search.start()
11
>>> search.end()
18
>>> text[search.start():search.end()]
'address'
```

The **match()** function performs the matching only at the beginning of the string; if there is no match with the first character, it goes no further in research within the string. If you do not find any match then it will not return any objects.

```
>>> re.match('[A,a]\w+',text)
>>>
```

If **match()** has a response, then it returns an object identical to what you saw for the **search()** function.

```
>>> re.match('T\w+',text)
<_sre.SRE_Match object at 0x0000000007D7ECC8>
>>> match = re.match('T\w+',text)
>>> text[match.start():match.end()]
'This'
```

Data Aggregation

The last stage of data manipulation is data aggregation. For data aggregation you generally mean a transformation that produces a single integer from an array. In fact, you have already made many operations of data aggregation, for example, when we calculated the sum(), mean(), count(). In fact, these functions operate on a set of data and shall perform a calculation with a consistent result consisting of a single value. However, a more formal manner and the one with more control in data aggregation is that which includes the categorization of a set.

The categorization of a set of data carried out for grouping is often a critical stage in the process of data analysis. It is a process of transformation since after the division into different groups, you apply a function that converts or transforms the data in some way depending on the group they belong to. Very often the two phases of grouping and application of a function are performed in a single step.

Also for this part of the data analysis, pandas provides a tool very flexible and high performance: **GroupBy**.

Again, as in the case of join, those familiar with relational databases and the SQL language can find similarities. Nevertheless, languages such as SQL are quite limited when applied to operations on groups. In fact, given the flexibility of a programming language like Python, with all the libraries available, especially pandas, you can perform very complex operations on groups.

GroupBy

Now you will analyze in detail what the process of GroupBy is and how it works. Generally, it refers to its internal mechanism as a process called **SPLIT-APPLY-COMBINE**. So in its pattern of operation you may conceive this process as divided into three different phases expressed precisely by three operations:

- splitting: division into groups of datasets
- applying: application of a function on each group
- combining: combination of all the results obtained by different groups

Analyze better the three different phases (see Figure 6-1). In the first phase, that of splitting, the data contained within a data structure, such as a Series or a DataFrame, are divided into several groups, according to a given criterion, which is often linked to indexes or just certain values in a column. In the jargon of SQL, values contained in this column are reported as keys. Furthermore, if you are working with two-dimensional objects such as the DataFrame, the grouping criterion may be applied both to the line (axis = 0) for that column (axis = 1).

The second phase, that of applying, consists in applying a function, or better a calculation expressed precisely by a function, which will produce a new and single value, specific to that group.

The last phase, that of combining, will collect all the results obtained from each group and combine them together to form a new object.

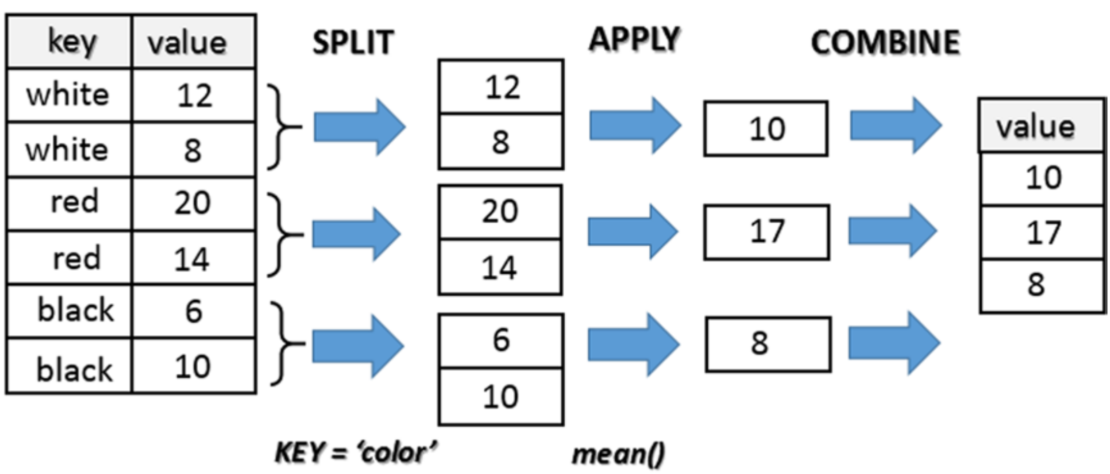

Figure 6-1. *The Split-Apply-Combine mechanism*

A Practical Example

You have just seen that the process of data aggregation in pandas is divided into various phases calls precisely split-apply-combine. With these pandas are not expressed explicitly with the functions as you would have expected, but by a **groupby()** function that generates an **GroupBy** object then that is the core of the whole process.

But to understand this mechanism, you must switch to a practical example. So, first, define a DataFrame containing both numeric and string values.

```
>>> frame = pd.DataFrame({ 'color': ['white','red','green','red','green'],
...                     'object': ['pen','pencil','pencil','ashtray','pen'],
...                     'price1' : [5.56,4.20,1.30,0.56,2.75],
...                     'price2' : [4.75,4.12,1.60,0.75,3.15]})
>>> frame
   color   object  price1  price2
0  white      pen    5.56    4.75
1    red   pencil    4.20    4.12
2  green   pencil    1.30    1.60
3    red  ashtray    0.56    0.75
4  green      pen    2.75    3.15
```

Suppose you want to calculate the average **price1** column using group labels listed in the column color. There are several ways to do this. You can for example access the **price1** column and call the **groupby()** function with the column color.

```
>>> group = frame['price1'].groupby(frame['color'])
>>> group
<pandas.core.groupby.SeriesGroupBy object at 0x00000000098A2A20>
```

The object that we got is a **GroupBy** object. In the operation that you just did there was not really any calculation; there was just a collection of all the information needed to calculate to be executed. What you have done is in fact a process of grouping, in which all rows having the same value of color are grouped into a single item.

To analyze in detail how the division into groups of rows of DataFrame was made, you call the attribute groups GroupBy object.

```
>>> group.groups
{'white': [0L], 'green': [2L, 4L], 'red': [1L, 3L]}
```

As you can see, each group is listed explicitly specifying the rows of the data frame assigned to each of them. Now it is sufficient to apply the operation on the group to obtain the results for each individual group.

```
>>> group.mean()
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
```

```
>>> group.sum()
color
green    4.05
red      4.76
white    5.56
Name: price1, dtype: float64
```

Hierarchical Grouping

You have seen how to group the data according to the values of a column as a key choice. The same thing can be extended to multiple columns, i.e., make a grouping of multiple keys hierarchical.

```
>>> ggroup = frame['price1'].groupby([frame['color'],frame['object']])
>>> ggroup.groups
{('red', 'ashtray'): [3L], ('red', 'pencil'): [1L], ('green', 'pen'): [4L], ('green', '
], ('white', 'pen'): [0L]}

>>> ggroup.sum()
color  object
green  pen        2.75
       pencil     1.30
red    ashtray    0.56
       pencil     4.20
white  pen        5.56
Name: price1, dtype: float64
```

So far you have applied the grouping to a single column of data, but in reality it can be extended to multiple columns or the entire data frame. Also if you do not need to reuse the object GroupBy several times, it is convenient to combine in a single passing all of the grouping and calculation to be done, without defining any intermediate variable.

```
>>> frame[['price1','price2']].groupby(frame['color']).mean()
       price1  price2
color
green   2.025   2.375
red     2.380   2.435
white   5.560   4.750
>>> frame.groupby(frame['color']).mean()
       price1  price2
color
green   2.025   2.375
red     2.380   2.435
white   5.560   4.750
```

Group Iteration

The **GroupBy** object supports the operation of an iteration for generating a sequence of 2-tuples containing the name of the group together with the data portion.

```
>>> for name, group in frame.groupby('color'):
...     print name
...     print group
...
green
   color  object  price1  price2
2  green  pencil    1.30    1.60
4  green     pen    2.75    3.15
red
  color   object  price1  price2
1   red   pencil    4.20    4.12
3   red  ashtray    0.56    0.75
white
   color object  price1  price2
0  white    pen    5.56    4.75
```

In the example you have just seen, you only applied the print variable for illustration. In fact, you replace the printing operation of a variable with the function to be applied on it.

Chain of Transformations

From these examples you have seen that for each grouping, when subjected to some function calculation or other operations in general, regardless of how it was obtained and the selection criteria, the result will be a data structure Series (if we selected a single column data) or DataFrame, which then retains the index system and the name of the columns.

```
>>> result1 = frame['price1'].groupby(frame['color']).mean()
>>> type(result1)
<class 'pandas.core.series.Series'>
>>> result2 = frame.groupby(frame['color']).mean()
>>> type(result2)
<class 'pandas.core.frame.DataFrame'>
```

So it is possible to select a single column at any point in the various phases of this process. Here are three cases in which the selection of a single column in three different stages of the process applies. This example illustrates the great flexibility of this system of grouping provided by pandas.

```
>>> frame['price1'].groupby(frame['color']).mean()
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
>>> frame.groupby(frame['color'])['price1'].mean()
color
```

```
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
>>> (frame.groupby(frame['color']).mean())['price1']
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
```

In addition, after an operation of aggregation the names of some columns may not be very meaningful in certain cases. In fact it is often useful to add a prefix to the column name that describes the type of business combination. Adding a prefix, instead of completely replacing the name, is very useful for keeping track of the source data from which they are derived aggregate values. This is important if you apply a process of transformation chain (a series of data frame is generated from each other) in which it is important to somehow keep some reference with the source data.

```
>>> means = frame.groupby('color').mean().add_prefix('mean_')
>>> means
       mean_price1  mean_price2
color
green        2.025        2.375
red          2.380        2.435
white        5.560        4.750
```

Functions on Groups

Although many methods have not been implemented specifically for use with GroupBy, they actually work correctly with data structures as the Series. You saw in the previous section how easy it is to get the Series by a GroupBy object, specifying the name of the column and then by applying the method to make the calculation. For example, you can use the calculation of quantiles with the **quantiles()** function.

```
>>> group = frame.groupby('color')
>>> group['price1'].quantile(0.6)
color
green    2.170
red      2.744
white    5.560
Name: price1, dtype: float64
```

You can also define their own aggregation functions. Define the function separately and then you pass as an argument to the **mark()** function. For example, you could calculate the range of the values of each group.

```
>>> def range(series):
...     return series.max() - series.min()
...
>>> group['price1'].agg(range)
color
green    1.45
```

```
red      3.64
white    0.00
Name: price1, dtype: float64
```

The agg() function() allows you to use aggregate functions on an entire DataFrame.

```
>>> group.agg(range)
       price1  price2
color
green    1.45    1.55
red      3.64    3.37
white    0.00    0.00
```

Also you can use more aggregate functions at the same time always with the **mark()** function passing an array containing the list of operations to be done, which will become the new columns.

```
>>> group['price1'].agg(['mean','std',range])
        mean       std  range
color
green  2.025  1.025305   1.45
red    2.380  2.573869   3.64
white  5.560       NaN   0.00
```

Advanced Data Aggregation

In this section you will be introduced to **transform()** and **apply()** functions, which will allow you to perform many kinds of group operations, some very complex.

Now suppose we want to bring together in the same DataFrame the following: (i) the DataFrame of origin (the one containing the data) and (ii) that obtained by the calculation of group aggregation, for example, the sum.

```
>>> frame = pd.DataFrame({ 'color':['white','red','green','red','green'],
...                       'price1':[5.56,4.20,1.30,0.56,2.75],
...                       'price2':[4.75,4.12,1.60,0.75,3.15]})
>>> frame
   color  price1  price2
0  white    5.56    4.75
1    red    4.20    4.12
2  green    1.30    1.60
3    red    0.56    0.75
4  green    2.75    3.15
>>> sums = frame.groupby('color').sum().add_prefix('tot_')
>>> sums
       tot_price1  tot_price2
color
green        4.05        4.75
red          4.76        4.87
white        5.56        4.75
```

```
>>> merge(frame,sums,left_on='color',right_index=True)
   color  price1  price2  tot_price1  tot_price2
0  white    5.56    4.75        5.56        4.75
1    red    4.20    4.12        4.76        4.87
3    red    0.56    0.75        4.76        4.87
2  green    1.30    1.60        4.05        4.75
4  green    2.75    3.15        4.05        4.75
```

So thanks to the **merge()**, you managed to add the results of a calculation of aggregation in each line of the data frame to start. But actually there is another way to do this type of operation. That is by using the **transform()**. This function performs the calculation of aggregation as you have seen before, but at the same time shows the values calculated based on the key value on each line of the data frame to start.

```
>>> frame.groupby('color').transform(np.sum).add_prefix('tot_')
   tot_price1  tot_price2
0        5.56        4.75
1        4.76        4.87
2        4.05        4.75
3        4.76        4.87
4        4.05        4.75
```

As you can see the transform() method is a more specialized function that has very specific requirements: the function passed as an argument must produce a single scalar value (aggregation) to be broadcasted.

The method to cover more general GroupBy is applicable to apply(). This method applies in its entirety the scheme split-apply-combine. In fact, this function divides the object into parts in order to be manipulated, invokes the passage of function on each piece, and then tries to chain together the various parts.

```
>>> frame = DataFrame( { 'color':['white','black','white','white','black','black'],
...                      'status':['up','up','down','down','down','up'],
...                      'value1':[12.33,14.55,22.34,27.84,23.40,18.33],
...                      'value2':[11.23,31.80,29.99,31.18,18.25,22.44]})
>>> frame
   color status  value1  value2
0  white     up   12.33   11.23
1  black     up   14.55   31.80
2  white   down   22.34   29.99
3  white   down   27.84   31.18
4  black   down   23.40   18.25

>>> frame.groupby(['color','status']).apply( lambda x: x.max())
              color status  value1  value2
color status
black down    black   down   23.40   18.25
      up      black     up   18.33   31.80
white down    white   down   27.84   31.18
      up      white     up   12.33   11.23
5  black     up   18.33   22.44
```

```
>>> frame.rename(index=reindex, columns=recolumn)
         color   object  value
first    white     ball   5.56
second     red      mug   4.20
third    green      pen   1.30
fourth   black   pencil   0.56
fifth   yellow  ashtray   2.75
>>> temp = date_range('1/1/2015', periods=10, freq= 'H')
>>> temp
<class 'pandas.tseries.index.DatetimeIndex'>
[2015-01-01 00:00:00, ..., 2015-01-01 09:00:00]
Length: 10, Freq: H, Timezone: None
>>> timeseries = Series(np.random.rand(10), index=temp)
>>> timeseries
2015-01-01 00:00:00    0.368960
2015-01-01 01:00:00    0.486875
2015-01-01 02:00:00    0.074269
2015-01-01 03:00:00    0.694613
2015-01-01 04:00:00    0.936190
2015-01-01 05:00:00    0.903345
2015-01-01 06:00:00    0.790933
2015-01-01 07:00:00    0.128697
2015-01-01 08:00:00    0.515943
2015-01-01 09:00:00    0.227647
Freq: H, dtype: float64

>>> timetable = DataFrame( {'date': temp, 'value1' : np.random.rand(10),
...                                       'value2' : np.random.rand(10)})
>>> timetable
                 date    value1    value2
0 2015-01-01 00:00:00  0.545737  0.772712
1 2015-01-01 01:00:00  0.236035  0.082847
2 2015-01-01 02:00:00  0.248293  0.938431
3 2015-01-01 03:00:00  0.888109  0.605302
4 2015-01-01 04:00:00  0.632222  0.080418
5 2015-01-01 05:00:00  0.249867  0.235366
6 2015-01-01 06:00:00  0.993940  0.125965
7 2015-01-01 07:00:00  0.154491  0.641867
8 2015-01-01 08:00:00  0.856238  0.521911
9 2015-01-01 09:00:00  0.307773  0.332822
```

We add to the DataFrame preceding a column that represents a set of text values that we will use as key values.

```
>>> timetable['cat'] = ['up','down','left','left','up','up','down','right','right','up']
>>> timetable
                 date    value1    value2    cat
0 2015-01-01 00:00:00  0.545737  0.772712     up
1 2015-01-01 01:00:00  0.236035  0.082847   down
2 2015-01-01 02:00:00  0.248293  0.938431   left
3 2015-01-01 03:00:00  0.888109  0.605302   left
4 2015-01-01 04:00:00  0.632222  0.080418     up
5 2015-01-01 05:00:00  0.249867  0.235366     up
6 2015-01-01 06:00:00  0.993940  0.125965   down
7 2015-01-01 07:00:00  0.154491  0.641867  right
8 2015-01-01 08:00:00  0.856238  0.521911  right
9 2015-01-01 09:00:00  0.307773  0.332822     up
```

The example shown above, however, has duplicate key values.

Conclusions

In this chapter you saw the three basic parts which divide the data manipulation: preparation, processing, and data aggregation. Thanks to a series of examples you've got to know a set of library functions that allow pandas to perform these operations.

You saw how to apply these functions on simple data structures so that you can become familiar with how they work and understand its applicability to more complex cases.

Eventually you get knowledge of all the tools necessary to prepare a data set for the next phase of data analysis: data visualization.

In the next chapter, you will be presented with the Python library Matplotlib, which can convert the data structures in any chart.