

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.

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 SomePackage-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 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.).

DataFrame			
	columns		
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 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 id	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

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, `idxmin()` and `idxmax()` 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
white	up	-1.964055	1.312100	-0.914750	-0.941930
	down	-1.886825	1.700858	-1.060846	-0.197669
red	up	-1.561761	1.225509	-0.244772	0.345843
	down	2.668155	0.528971	-1.633708	0.921735

```
>>> mframe.swaplevel('colors', 'status')
```

objects		pen		paper	
id		1	2	1	2
status colors					
up	white	-1.964055	1.312100	-0.914750	-0.941930
down	white	-1.886825	1.700858	-1.060846	-0.197669
up	red	-1.561761	1.225509	-0.244772	0.345843
down	red	2.668155	0.528971	-1.633708	0.921735

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

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

Summary Statistic by Level

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

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

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

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

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

Conclusions

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

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

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

CHAPTER 5

pandas: Reading and Writing Data

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

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

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

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

I/O API Tools

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

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

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

CSV and Textual Files

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

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

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

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

Reading Data in CSV or Text Files

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

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

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

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

Listing 5-1. `ch05_01.csv`

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

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

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

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


```
>>> pd.read_table('ch05_01.csv', sep=',')
```

	white	red	blue	green	animal
0	1	5	2	3	cat
1	2	7	8	5	dog
2	3	3	6	7	horse
3	2	2	8	3	duck
4	4	4	2	1	mouse

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

Listing 5-2. ch05_02.csv

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

```
>>> pd.read_csv('ch05_02.csv')
```

	1	5	2	3	cat
0	2	7	8	5	dog
1	3	3	6	7	horse
2	2	2	8	3	duck
3	4	4	2	1	mouse

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

```
>>> pd.read_csv('ch05_02.csv', header=None)
```

	0	1	2	3	4
0	1	5	2	3	cat
1	2	7	8	5	dog
2	3	3	6	7	horse
3	2	2	8	3	duck
4	4	4	2	1	mouse

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

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

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

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

Listing 5-3. `ch05_03.csv`

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

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

Using RegEx to Parse TXT Files

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

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

Table 5-1. *Metacharacters*

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

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

Listing 5-4. ch05_04.txt

```
white red blue green
```

```
1 5 2 3
```

```
2 7 8 5
```

```
3 3 6 7
```

```
>>> pd.read_table('ch05_04.txt', sep='\s+', engine='python')
```

```
white red blue green
```

```
0 1 5 2 3
```

```
1 2 7 8 5
```

```
2 3 3 6 7
```

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

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

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

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

Listing 5-5. ch05_05.txt

```
000END123AAA122
```

```
001END124BBB321
```

```
002END125CCC333
```

```
>>> pd.read_table('ch05_05.txt', sep='\D+', header=None, engine='python')
```

```
0 1 2
```

```
0 0 123 122
```

```
1 1 124 321
```

```
2 2 125 333
```

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

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

Listing 5-6. `ch05_06.txt`

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

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

Reading TXT Files Into Parts

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

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

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

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

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

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

Writing Data in CSV

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

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

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

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

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

Listing 5-7. `ch05_07.csv`

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

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

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

Listing 5-8. `ch05_07b.csv`

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

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

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

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

```
>>> frame3.to_csv('ch05_08.csv')
```

Listing 5-9. ch05_08.csv

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

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

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


Listing 5-10. ch05_09.csv

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

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

Reading and Writing HTML Files

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

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

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

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

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

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

```
conda install html5lib
```

Writing Data in HTML

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

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

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

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

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

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

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

```

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

```

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

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

```

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

```

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

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

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

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

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

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

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

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

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

Reading Data from an HTML File

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

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

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

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

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

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

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

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

Reading Data from XML

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

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

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

Listing 5-11. `books.xml`

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

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

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

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

```
>>> xml = objectify.parse('books.xml')
>>> xml
<lxml.etree._ElementTree object at 0x000000009734E08>
```

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> def etree2df(root):
...     column_names = []
...     for i in range(0, len(root.getchildren()[0].getchildren())):
...         column_names.append(root.getchildren()[0].getchildren()[i].tag)
...     xml:frame = pd.DataFrame(columns=column_names)
...     for j in range(0, len(root.getchildren())):
...         obj = root.getchildren()[j].getchildren()
...         texts = []
...         for k in range(0, len(column_names)):
...             texts.append(obj[k].text)
...         row = dict(zip(column_names, texts))
...         row_s = pd.Series(row)
...         row_s.name = j
...         xml:frame = xml:frame.append(row_s)
...     return xml:frame
...
>>> etree2df(root)
```

	Author	Title	Genre	Price	PublishDate
0	Ross, Mark	XML Cookbook	Computer	23.56	2014-22-01
1	Bracket, Barbara	XML for Dummies	Computer	35.95	2014-12-16

Reading and Writing Data on Microsoft Excel Files

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> frame = pd.DataFrame(np.random.random((4,4)),
...                        index = ['exp1', 'exp2', 'exp3', 'exp4'],
...                        columns = ['Jan2015', 'Feb2015', 'Mar2015', 'Apr2005'])
>>> frame
      Jan2015  Feb2015  Mar2015  Apr2005
exp1  0.030083  0.065339  0.960494  0.510847
exp2  0.531885  0.706945  0.964943  0.085642
exp3  0.981325  0.868894  0.947871  0.387600
exp4  0.832527  0.357885  0.538138  0.357990
>>> frame.to_excel('data2.xlsx')
```

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

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

Figure 5-3. The dataframe in the Excel file

JSON Data

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

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

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

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

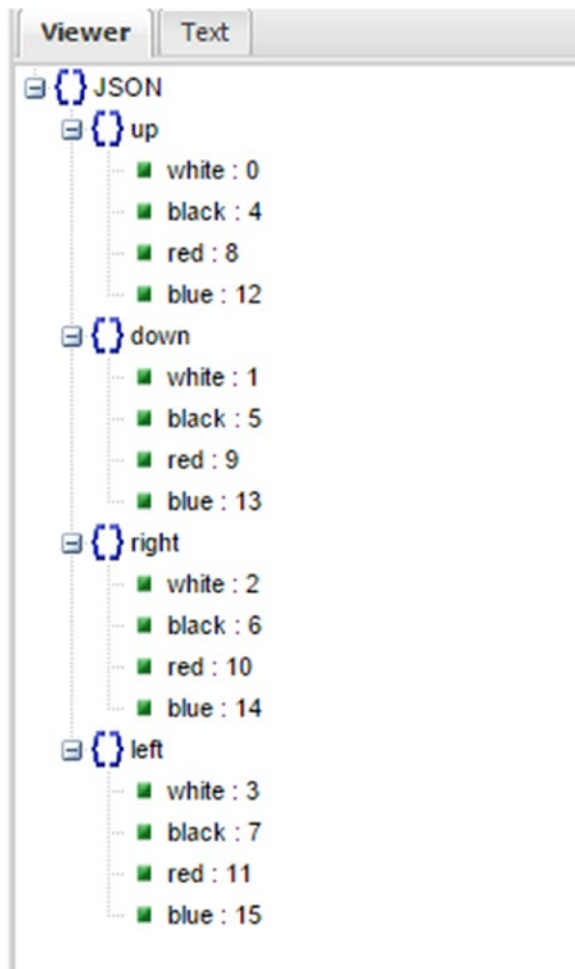


Figure 5-4. *JSONViewer*

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

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

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

Listing 5-12. frame.json

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

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

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

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

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

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

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

Listing 5-13. books.json

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

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

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

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

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

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

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

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

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

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

The Format HDF5

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

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

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

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

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

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

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

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

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

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

```
>>> store = HDFStore('mydata.h5')
>>> store['obj1'] = frame
```

From here, you can guess how you can store multiple data structures within the same HDF5 file, specifying for each of them a label.

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

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

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


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

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

Pickle—Python Object Serialization

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

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

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

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

Serialize a Python Object with cPickle

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

```
>>> import pickle
```

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

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

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

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

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

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

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

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

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

Pickling with pandas

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

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

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

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

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

```
>>> pd.read_pickle('frame.pkl')
```

	0	1	2	3
up	0	1	2	3
down	4	5	6	7
left	8	9	10	11
right	12	13	14	15

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

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

Interacting with Databases

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

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

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

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

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

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

For PostgreSQL:

```
>>> engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')
```

For MySQL

```
>>> engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')
```

For Oracle

```
>>> engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')
```

For MSSQL

```
>>> engine = create_engine('mssql+pyodbc://mydsn')
```

For SQLite

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

Loading and Writing Data with SQLite3

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

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

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

```
>>> frame = pd.DataFrame( np.arange(20).reshape(4,5),
...                        columns=['white', 'red', 'blue', 'black', 'green'])
>>> frame
```

	white	red	blue	black	green
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

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

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

Convert the dataframe in a table within the database.

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

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

```
>>> pd.read_sql('colors', engine)
```

	index	white	red	blue	black	green
0	0	0	1	2	3	4
1	1	5	6	7	8	9
2	2	10	11	12	13	14
3	3	15	16	17	18	19

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

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

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

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

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

Now you can enter data using the SQL INSERT statement.

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

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

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

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

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

```

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

```

As you can see, this approach is quite laborious.

Loading and Writing Data with PostgreSQL

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

```

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

```

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

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

With Anaconda:

```
conda install psycopg2
```

Or if you are using PyPi:

```
pip install psycopg2
```

Now you can establish a connection with the database:

```

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

```

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

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

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

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

Create a dataframe object:

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

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

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

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

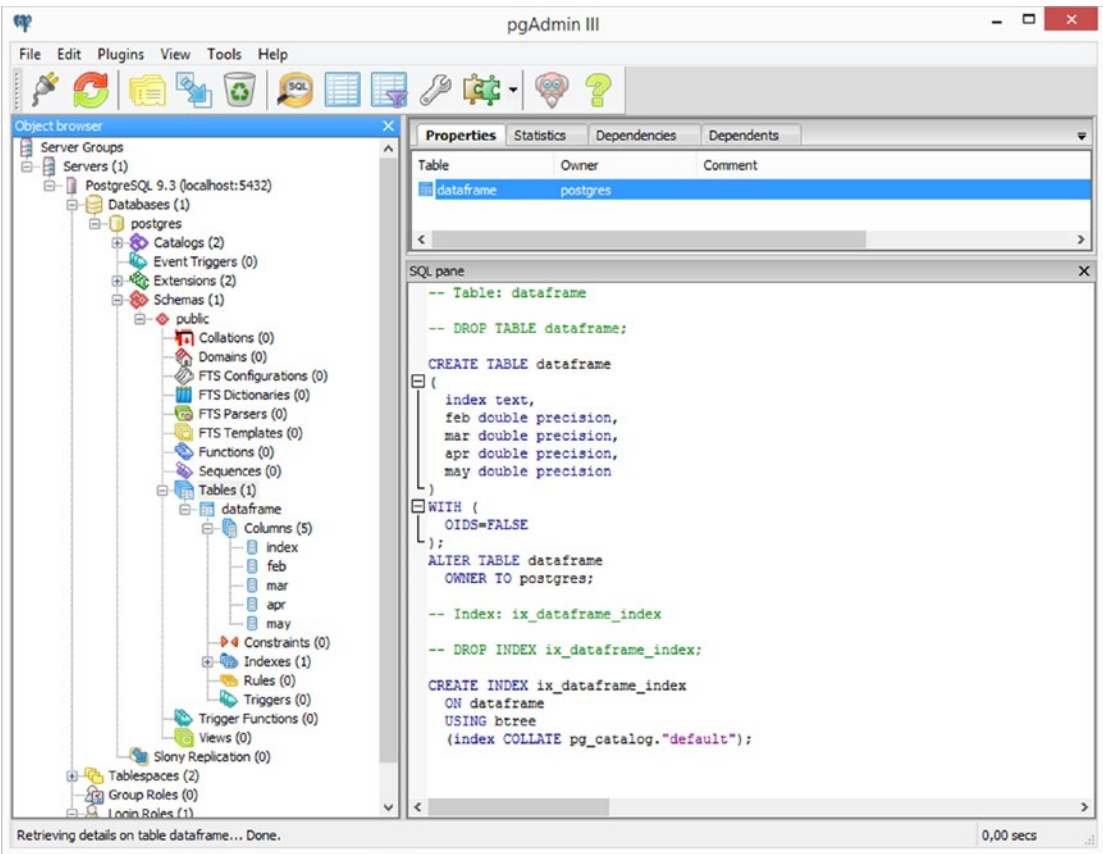


Figure 5-5. The pgAdmin III application is a perfect graphical DB manager for PostgreSQL

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

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

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

```
postgres=# SELECT * FROM DATAFRAME;
```

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

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

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

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

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

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

Reading and Writing Data with a NoSQL Database: MongoDB

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

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

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

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

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

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

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

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

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

Now it is the time to load the data in the collection. Create a DataFrame.

```
>>> frame = pd.DataFrame( np.arange(20).reshape(4,5),
...                        columns=['white','red','blue','black','green'])
>>> frame
```

	white	red	blue	black	green
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

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

```
>>> import json
>>> record = json.loads(frame.T.to_json()).values()
>>> record
```

```
[{'blue': 7, 'green': 9, 'white': 5, 'black': 8, 'red': 6},
{'blue': 2, 'green': 4, 'white':
0, 'black': 3, 'red': 1}, {'blue': 17, 'green': 19, 'white': 15,
'black': 18, 'red': 16}, {'u
'blue': 12, 'green': 14, 'white': 10, 'black': 13, 'red': 11}]
```

Now you are finally ready to insert a document in the collection, and you can do this with the **insert()** function.

```
>>> collection.mydocument.insert(record)
[ObjectId('54fc3afb9bfbee47f4260357'), ObjectId('54fc3afb9bfbee47f4260358'),
ObjectId('54fc3afb9bfbee47f4260359'), ObjectId('54fc3afb9bfbee47f426035a')]
```

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

```
>>> cursor = collection['mydocument'].find()
>>> dataframe = (list(cursor))
>>> del dataframe['_id']
>>> dataframe
```

	black	blue	green	red	white
0	8	7	9	6	5
1	3	2	4	1	0
2	18	17	19	16	15
3	13	12	14	11	10

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

Conclusions

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

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

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

CHAPTER 6

pandas in Depth: Data Manipulation

In the previous chapter you saw how to acquire data from data sources such as databases and files. Once you have the data in the dataframe format, they are ready to be manipulated. It's important to prepare the data so that they can be more easily subjected to analysis and manipulation. Especially in preparation for the next phase, the data must be ready for visualization.

In this chapter you will go in depth into the functionality that the pandas library offers for this stage of 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, 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 here.

- Loading
- Assembling

- Merging
- Concatenating
- Combining
- Reshaping (pivoting)
- Removing

The previous chapter covered loading. In the loading phase, there is also that part of the preparation that concerns the conversion from many different formats into a data structure such as a dataframe. But even after you have 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 to get the data into a unified data structure.

The data contained in the pandas objects can be assembled 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 uses 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 dataframes that will serve 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 by 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. In addition to the common column, the columns from 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 dataframes having columns with the same name. So if you launch a merge, you do not get any results.

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

It is necessary to explicitly define the criteria for 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, which 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 merge multiple keys, you simply 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 an Index

In some cases, instead of considering the columns of a dataframe as keys, indexes could be used as keys for merging. Then in order to decide which indexes to consider, you 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 no columns overlapping.

In fact, if you launch

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

You will get an error code because some columns in `frame1` have the same name as `frame2`. Then rename the columns in `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 merge, 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 a 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 = np.arange(9).reshape((3,3))
>>> 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]])
```

With the pandas library and its data structures like series and dataframe, 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 a 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

The problem with 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 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 by column values on rows or vice versa.

Pivoting with Hierarchical Indexing

You have already seen that a 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:

```
>>> ser5 = 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 datasets 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 dataset 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, 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 has some disadvantages. One, for example, is the multiplicity and repetition of some fields. Considering the columns as keys, the data in 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 dataframe.

```
>>> wideframe = longframe.pivot('color','item')
>>> wideframe
```

	value		
item	ball	mug	pen
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 the 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 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 an 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 in a dataframe, with possible additions by other dataframe and removal of unwanted parts.

Now we begin 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 by modifying the indexes. Other steps include handling and processing the numerical values of the data and 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. In this case, pandas provides a series of tools to analyze the duplicate data present in large data structures.

First, create a simple dataframe with some duplicate rows.

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

The `dfame.duplicated()` function applied to a dataframe can detect the rows that 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.

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

Having a Boolean series as a return value can be useful in many cases, especially for the filtering. In fact, if you want to know which are the duplicate rows, just type the following:

```
>>> dfame[dfame.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 dataframes without duplicate rows.

```
>>> df[df.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. 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 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 they all accept a dict object.

- `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 an undesirable form.

```
>>> 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	price
0	white	ball	5.56
1	rosso	mug	4.20
2	verde	pen	1.30
3	black	pencil	0.56
4	yellow	ashtray	2.75

To be able to replace the incorrect values with new values, it is necessary to define a mapping of correspondences, containing as a key the new values.

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

Now the only thing you can do is 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 NaN values with another value, for example 0. You can use `replace()`, which performs its job very well.

```
>>> ser = pd.Series([1,3,np.nan,4,6,np.nan,3])
>>> ser
0    1.0
1    3.0
2    NaN
3    4.0
4    6.0
5    NaN
6    3.0
dtype: float64
>>> ser.replace(np.nan,0)
0    1.0
1    3.0
2    0.0
3    4.0
4    6.0
5    0.0
6    3.0
dtype: float64
```

Adding Values via Mapping

In the previous example, you saw how to substitute values by mapping 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.

```
>>> prices = {
...     '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 dataframe.

```
>>> 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 an 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. 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, you can avoid 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 be used, 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 into smaller intervals and counting the occurrence or statistics in them. Another case might be when you have a huge number 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.

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 you use 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 `categories` array indicating the names of the different internal categories and a `codes` 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.categories
IntervalIndex([0, 25], (25, 50], (50, 75], (75, 100]]
              closed='right'
              dtype='interval[int64]')
>>> cat.codes
array([0, 1, 2, 2, 1, 3, 3, 0, 0, 2, 2, 1, 3, 0, 1, 3, 3], dtype=int8)
```

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, by using `cut()`, 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 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 data analysis, the need to detect the presence of abnormal values in 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 the 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 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 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 often.

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 to use regular expressions.

Built-in Methods for String Manipulation

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 you to separate parts of the text, taking as a reference point a separator, for example, a comma.

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

As you can see in the first element, you have a string with a space character at the end. To overcome this common problem, you have to use the `split()` function along with the **`strip()`** function, which trims the 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 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 you have many parts to be concatenated, a more practical approach in this case is to use the `join()` function assigned to the separator character, with which you want to join the various strings.

```
>>> 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 searching for pieces of text in them, i.e., substrings. Python provides the keyword that 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 characters 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 the 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 replacing or eliminating 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

Regular expressions provide a very flexible way to search and match string patterns within 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*.

First of all, when you want to use 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 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 text into parts through a separator character, you used `split()`. There is a `split()` function even for the `re` module that performs the same operations, only it can accept a regex pattern as the criteria 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!']
```

Let's 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 a `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!']
```

To match a regex pattern to 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 the following:

```
>>> 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 `findall()` function—`match()` and `search()`. While `findall()` returns all matches within a list, the `search()` function 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; span=(11, 18), match='address'>
```

This object does not contain the value of the substring that responds to the regex pattern, but returns 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 matching only at the beginning of the string; if there is no match to the first character, it goes no farther 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, it returns an object identical to what you saw for the `search()` function.

```
>>> re.match('T\\w+',text)
<_sre.SRE_Match object; span=(0, 4), match='This'>
>>> match = re.match('T\\w+',text)
>>> text[match.start():match.end()]
'This'
```

Data Aggregation

The last stage of data manipulation is data aggregation. Data aggregation involves a transformation that produces a single integer from an array. In fact, you have already made many operations of data aggregation, for example, when you calculated the `sum()`, `mean()`, and `count()`. In fact, these functions operate on a set of data and 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 that's 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 the process of GroupBy and how it works. Generally, it refers to its internal mechanism as a process called *split-apply-combine*. In its pattern of operation you may conceive this process as divided into three phases expressed 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 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 given criteria, which is often linked to indexes or to 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 a dataframe, the grouping criterion may be applied both to the line (*axis* = 0) for that column (*axis* = 1).

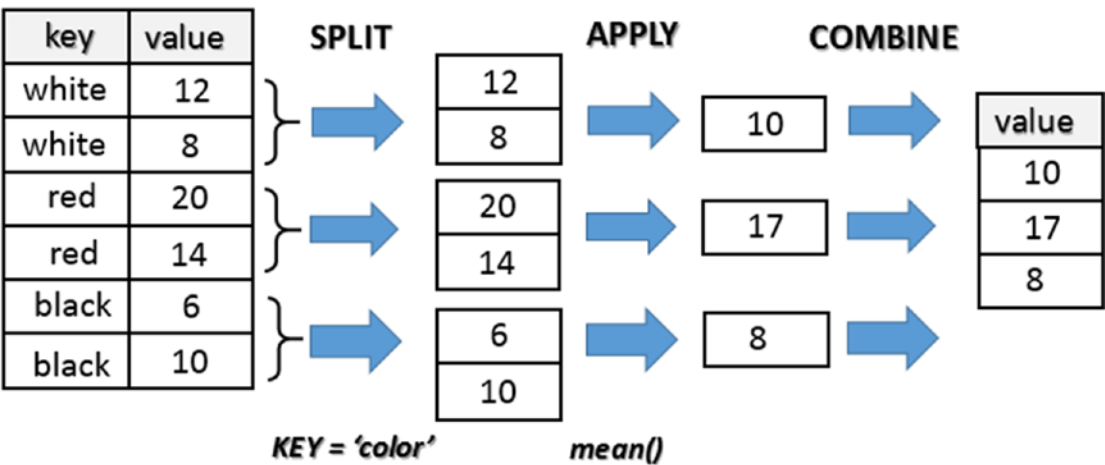


Figure 6-1. The split-apply-combine mechanism

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

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

A Practical Example

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

To better understand this mechanism, let's switch to a practical example. First define a dataframe containing 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 of the `price1` column using group labels listed in the `color` column. There are several ways to do this. You can for example access the `price1` column and call the `groupby()` function with the `color` column.

```
>>> group = frame['price1'].groupby(frame['color'])
>>> group
<pandas.core.groupby.SeriesGroupBy object at 0x0000000098A2A20>
```

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 the average. What you have done is `group`, in which all rows having the same value of `color` are grouped into a single item.

To analyze in detail how the dataframe was divided into groups of rows, you call the attribute `groups` of `GroupBy` object.

```
>>> group.groups
{'green': Int64Index([2, 4], dtype='int64'),
 'red': Int64Index([1, 3], dtype='int64'),
 'white': Int64Index([0], dtype='int64')}
```

As you can see, each group is listed and explicitly specifies the rows of the dataframe 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
{('green', 'pen'): Int64Index([4], dtype='int64'),
 ('green', 'pencil'): Int64Index([2], dtype='int64'),
 ('red', 'ashtray'): Int64Index([3], dtype='int64'),
 ('red', 'pencil'): Int64Index([1], dtype='int64'),
 ('white', 'pen'): Int64Index([0], dtype='int64')}

>>> 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 to the entire dataframe. 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 to generate a sequence of two-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 just saw, 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 a 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'>
```

It is therefore 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 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 derive aggregate values. This is important if you apply a process of transformation chain (a series or dataframe is generated from each other) in which it is important to 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, by 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
```

You can also use more aggregate functions at the same time, 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 the `transform()` and `apply()` functions, which 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: the dataframe of origin (the one containing the data) and 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

```

Thanks to `merge()`, can add the results of the aggregation in each line of the dataframe to start. But there is another way to do this type of operation. That is by using `transform()`. This function performs 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 dataframe 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 split-apply-combine scheme. In fact, this function divides the object into parts in order to be manipulated, invokes the passage of functions on each piece, and then tries to chain together the various parts.

```
>>> frame = pd.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 = pd.date_range('1/1/2015', periods=10, freq= 'H')
>>> temp
```

```
DatetimeIndex(['2015-01-01 00:00:00', '2015-01-01 01:00:00',
               '2015-01-01 02:00:00', '2015-01-01 03:00:00',
               '2015-01-01 04:00:00', '2015-01-01 05:00:00',
               '2015-01-01 06:00:00', '2015-01-01 07:00:00',
               '2015-01-01 08:00:00', '2015-01-01 09:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Length: 10, Freq: H, Timezone: None

```
>>> timeseries = pd.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 = pd.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

You then add to the dataframe preceding a column that represents a set of text values that you 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 here, however, has duplicate key values.

Conclusions

In this chapter, you saw the three basic parts that divide the data manipulation: preparation, processing, and data aggregation. Thanks to a series of examples, you learned about 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 their applicability to more complex cases.

Eventually, you now have the knowledge you need to prepare a dataset 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 data structures in any chart.

CHAPTER 7

Data Visualization with matplotlib

After discussing in the previous chapters Python libraries that were responsible for data processing, now it is time for you to see a library that takes care of visualization. This library is matplotlib.

Data visualization is often underestimated in data analysis, but it is actually a very important factor because incorrect or inefficient data representation can ruin an otherwise excellent analysis. In this chapter, you will discover the various aspects of the matplotlib library, including how it is structured, and how to maximize the potential that it offers.

The matplotlib Library

matplotlib is a Python library specializing in the development of two-dimensional charts (including 3D charts). In recent years, it has been widespread in scientific and engineering circles (<http://matplotlib.org>).

Among all the features that have made it the most used tool in the graphical representation of data, there are a few that stand out:

- Extreme simplicity in its use
- Gradual development and interactive data visualization
- Expressions and text in LaTeX
- Greater control over graphic elements
- Export to many formats, such as PNG, PDF, SVG, and EPS

matplotlib is designed to reproduce as much as possible an environment similar to MATLAB in terms of both graphical view and syntactic form. This approach has proved successful, as it has been able to exploit the experience of software (MATLAB) that has been on the market for several years and is now widespread in all professional technical-scientific circles. Not only is matplotlib based on a scheme known and quite familiar to most experts in the field, but also it also exploits those optimizations that over the years have led to a deducibility and simplicity in its use, which makes this library also an excellent choice for those approaching data visualization for the first time, especially those without any experience with applications such as MATLAB or similar.

In addition to simplicity and deducibility, the matplotlib library inherited *interactivity* from MATLAB as well. That is, the analyst can insert command after command to control the gradual development of a graphical representation of data. This mode is well suited to the more interactive approaches of Python as the IPython QtConsole and IPython Notebook (see Chapter 2), thus providing an environment for data analysis that has little to envy from other tools such as Mathematica, IDL, or MATLAB.

The genius of those who developed this beautiful library was to use and incorporate the good things currently available and in use in science. This is not only limited, as we have seen, to the operating mode of MATLAB and similar, but also to models of textual formatting of scientific expressions and symbols represented by LaTeX. Because of its great capacity for display and presentation of scientific expressions, LaTeX has been an irreplaceable element in any scientific publication or documentation, where the need to visually represent expressions like integrals, summations, and derivatives is mandatory. Therefore matplotlib integrates this remarkable instrument in order to improve the representative capacity of charts.

In addition, you must not forget that matplotlib is not a separate application but a library of a programming language like Python. So it also takes full advantage of the potential that programming languages offer. matplotlib looks like a graphics library that allows you to programmatically manage the graphic elements that make up a chart so that the graphical display can be controlled in its entirety. The ability to program the graphical representation allows management of the reproducibility of the data representation across multiple environments and especially when you make changes or when the data is updated.

Moreover, since matplotlib is a Python library, it allows you to exploit the full potential of other libraries available to any developer that implements with this language. In fact, with regard to data analysis, matplotlib normally cooperates with a set of other libraries such as NumPy and pandas, but many other libraries can be integrated without any problem.

Finally, graphical representations obtained through encoding with this library can be exported in the most common graphic formats (such as PNG and SVG) and then be used in other applications, documentation, web pages, etc.

Installation

There are many options for installing the matplotlib library. If you choose to use a distribution of packages like Anaconda or Enthought Canopy, installing the matplotlib package is very simple. For example, with the conda package manager, you have to enter the following:

```
conda install matplotlib
```

If you want to directly install this package, the commands to insert vary depending on the operating system.

On Debian-Ubuntu Linux systems, use this:

```
sudo apt-get install python-matplotlib
```

On Fedora-Redhat Linux systems, use this:

```
sudo yum install python-matplotlib
```

On Windows or MacOS, you should use pip for installing matplotlib.

The IPython and IPython QtConsole

In order to get familiar with all the tools provided by the Python world, I chose to use IPython both from a terminal and from the QtConsole. This is because IPython allows you to exploit the interactivity of its enhanced terminal and, as you will see, IPython QtConsole also allows you to integrate graphics directly inside the console.

To run an IPython session, simply run the following command:

```
ipython
```

```
Python 3.6.3 (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)]  
Type "copyright", "credits" or "license" for more information.
```

```
IPython 3.6.3 -- An enhanced Interactive Python. Type '?' for help.
```

```
In [1]:
```

Whereas if you want to run the Jupyter QtConsole with the ability to display graphics within the line commands of the session, you use:

```
jupyter qtconsole
```

A window with a new open IPython session will immediately appear on the screen, as shown in Figure 7-1.

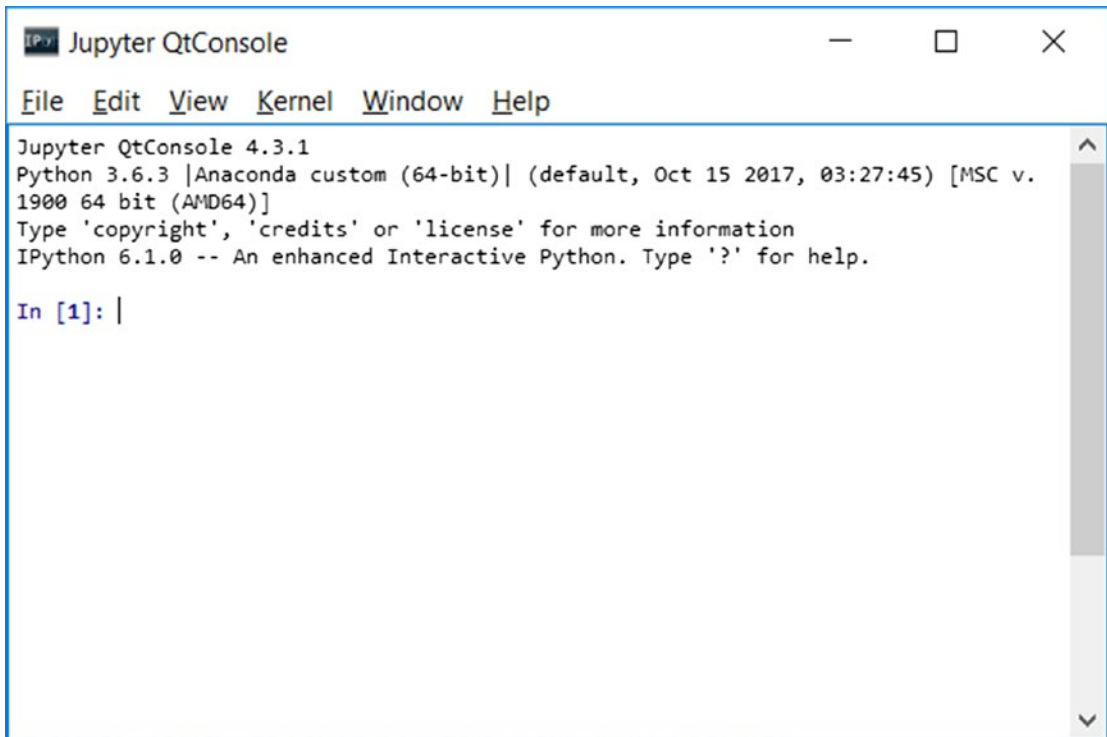


Figure 7-1. The IPython QtConsole

However, if you want to continue using a standard Python session you are free to do so. If you do not like working with IPython and want to continue to use Python from the terminal, all the examples in this chapter will still be valid.

The matplotlib Architecture

One of the key tasks that matplotlib must take on is provide a set of functions and tools that allow representation and manipulation of a *Figure* (the main object), along with all internal objects of which it is composed. However, matplotlib not only deals with graphics but also provides all the tools for the event handling and the ability to animate graphics. So, thanks to these additional features, matplotlib proves to be capable of producing interactive charts based on the events triggered by pressing a key on the keyboard or on mouse movement.

The architecture of matplotlib is logically structured into three layers, which are placed at three different levels (see Figure 7-2). The communication is unidirectional, that is, each layer can communicate with the underlying layer, while the lower layers cannot communicate with the top ones.

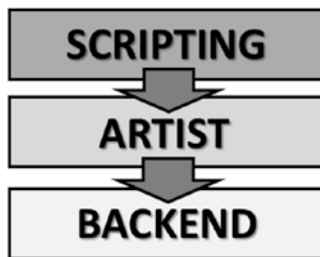


Figure 7-2. *The three layers of the matplotlib architecture*

The three layers are as follows:

- Scripting
- Artist
- Backend

Backend Layer

In the diagram of the matplotlib architecture, the layer that works at the lowest level is the *Backend* layer. This layer contains the matplotlib APIs, a set of classes that play the role of implementation of the graphic elements at a low level.

- FigureCanvas is the object that embodies the concept of drawing area.
- Renderer is the object that draws on FigureCanvas.
- Event is the object that handles user inputs (keyboard and mouse events).

Artist Layer

As an intermediate layer, we have a layer called *Artist*. All the elements that make up a chart, such as the title, axis labels, markers, etc., are instances of the Artist object. Each of these instances plays its role within a hierarchical structure (as shown in Figure 7-3).

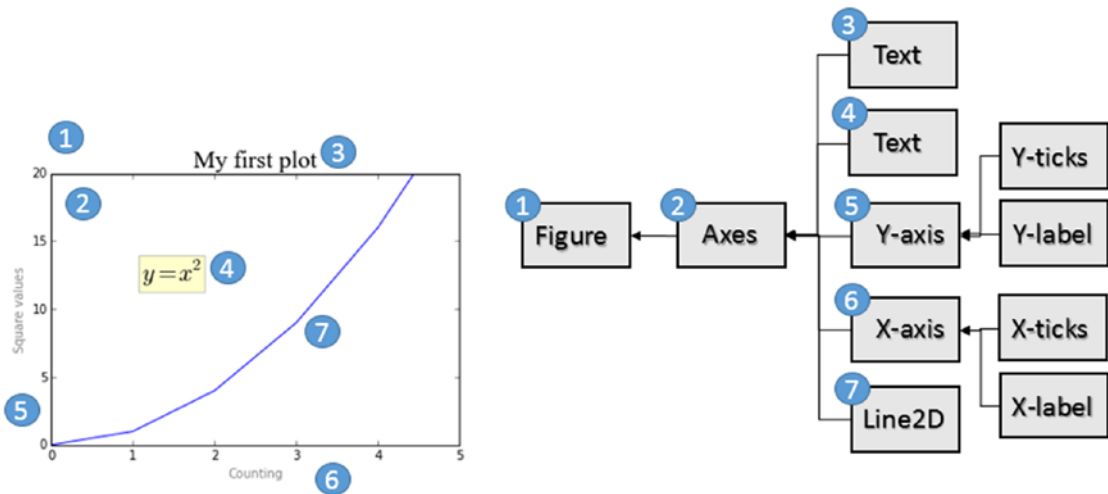


Figure 7-3. Each element of a chart corresponds to an instance of Artist structured in a hierarchy

There are two Artist classes: primitive and composite.

- The *primitive artists* are individual objects that constitute the basic elements to form a graphical representation in a plot, for example a `Line2D`, or as a geometric figure such as a `Rectangle` or `Circle`, or even pieces of text.
- The *composite artists* are those graphic elements present in a chart that are composed of several base elements, namely, the primitive artists. Composite artists are for example the `Axis`, `Ticks`, `Axes`, and `Figures` (see Figure 7-4).

Generally, working at this level you will have to deal often with objects in higher hierarchy as `Figure`, `Axes`, and `Axis`. So it is important to fully understand what these objects are and what role they play within the graphical representation. Figure 7-4 shows the three main Artist objects (composite artists) that are generally used in all implementations performed at this level.

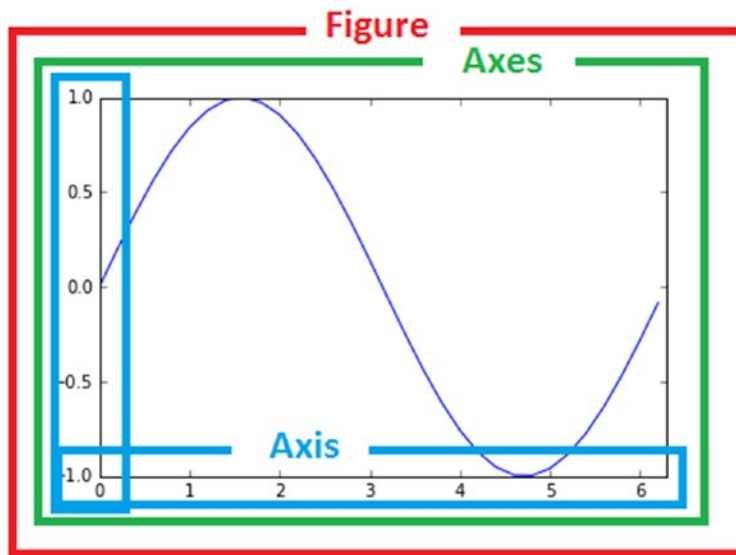


Figure 7-4. The three main artist objects in the hierarchy of the Artist layer

- *Figure* is the object with the highest level in the hierarchy. It corresponds to the entire graphical representation and generally can contain many *Axes*.

- *Axes* is generally what you mean as plot or chart. Each Axis object belongs to only one Figure, and is characterized by two Artist Axis (three in the three-dimensional case). Other objects, such as the title, the x label, and the y label, belong to this composite artist.
- *Axis* objects that take into account the numerical values to be represented on Axes, define the limits and manage the ticks (the mark on the axes) and tick labels (the label text represented on each tick). The position of the tick is adjusted by an object called a *Locator* while the formatting tick label is regulated by an object called a *Formatter*.

Scripting Layer (pyplot)

Artist classes and their related functions (the matplotlib API) are particularly suitable to all developers, especially for those who work on web application servers or develop the GUI. But for purposes of calculation, and in particular for the analysis and visualization of data, the scripting layer is best. This layer consists of an interface called *pyplot*.

pylab and pyplot

In general there is talk of *pylab* and *pyplot*. But what is the difference between these two packages? Pylab is a module that is installed along with matplotlib, while pyplot is an internal module of matplotlib. Often you will find references to one or the other approach.

```
from pylab import *

and

import matplotlib.pyplot as plt
import numpy as np
```

Pylab combines the functionality of pyplot with the capabilities of NumPy in a single namespace, and therefore you do not need to import NumPy separately. Furthermore, if you import pylab, pyplot and NumPy functions can be called directly without any reference to a module (namespace), making the environment more similar to MATLAB.

```
plot(x,y)
array([1,2,3,4])
```

Instead of

```
plt.plot()
np.array([1,2,3,4])
```

The *pyplot* package provides the classic Python interface for programming the matplotlib library, has its own namespace, and requires the import of the NumPy package separately. This approach is the one chosen for this book; it is the main topic of this chapter; and it will be used for the rest of the book. In fact this choice is shared and approved by most Python developers.

pyplot

The pyplot module is a collection of command-style functions that allow you to use matplotlib much like MATLAB. Each pyplot function will operate or make some changes to the Figure object, for example, the creation of the Figure itself, the creation of a plotting area, representation of a line, decoration of the plot with a label, etc.

Pyplot also is *stateful*, in that it tracks the status of the current figure and its plotting area. The functions called act on the current figure.

A Simple Interactive Chart

To get familiar with the matplotlib library and in a particular way with Pyplot, you will start creating a simple interactive chart. Using matplotlib, this operation is very simple; in fact, you can achieve it using only three lines of code.

But first you need to import the pyplot package and rename it as plt.

```
In [1]: import matplotlib.pyplot as plt
```

In Python, the constructors generally are not necessary; everything is already implicitly defined. In fact when you import the package, the plt object with all its graphics capabilities have already been instantiated and ready to use. In fact, you simply use the plot() function to pass the values to be plotted.

Thus, you can simply pass the values that you want to represent as a sequence of integers.

```
In [2]: plt.plot([1,2,3,4])
```

```
Out[2]: [<matplotlib.lines.Line2D at 0xa3eb438>]
```

As you can see, a `Line2D` object has been generated. The object is a line that represents the linear trend of the points included in the chart.

Now it is all set. You just have to give the command to show the plot using the `show()` function.

```
In [3]: plt.show()
```

The result will be the one shown in Figure 7-5. It looks just a window, called the *plotting window*, with a toolbar and the plot represented within it, just as with MATLAB.

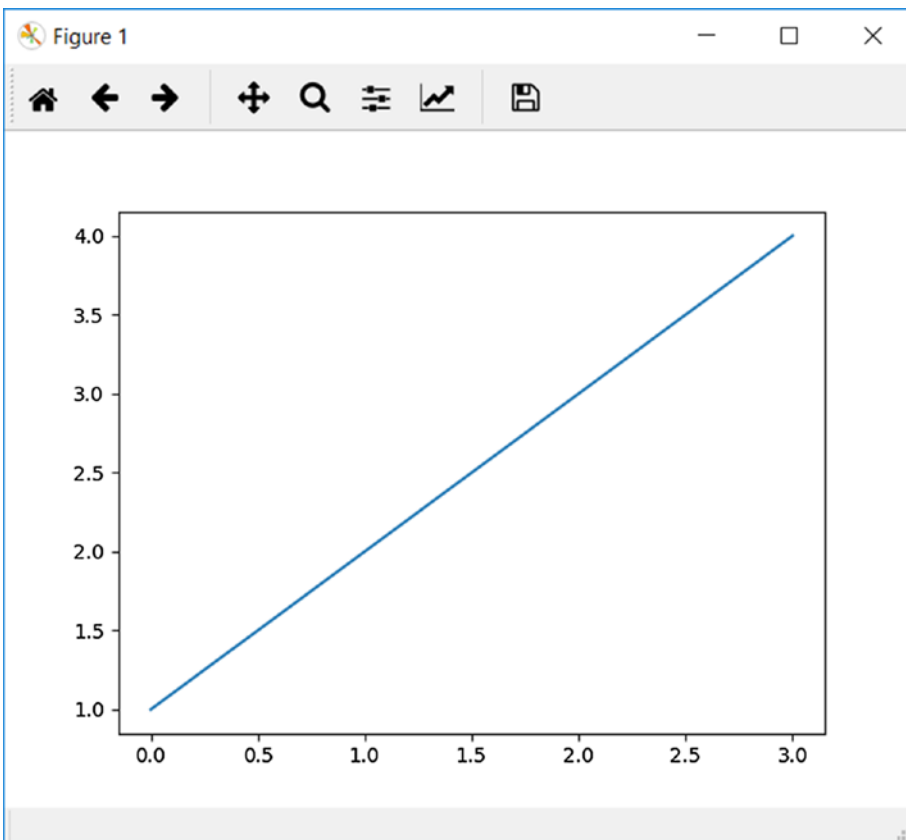

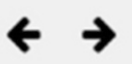







Figure 7-5. *The plotting window*

The Plotting Window

The plotting window is characterized by a toolbar at the top in which there are a series of buttons.

-  Resets the original view
-  Goes to the previous/next view
-  Pans axes with left mouse, zoom with right
-  Zooms to rectangle
-  Configures subplots
-  Saves/exports the figure
-  Edits the axis, curve, and image parameters

The code entered into the IPython console corresponds on the Python console to the following series of commands:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3,4])
[<matplotlib.lines.Line2D at 0x0000000007DABFD0>]
>>> plt.show()
```

If you are using the IPython QtConsole, you may have noticed that after calling the `plot()` function the chart is displayed directly without explicitly invoking the `show()` function (see Figure 7-6).

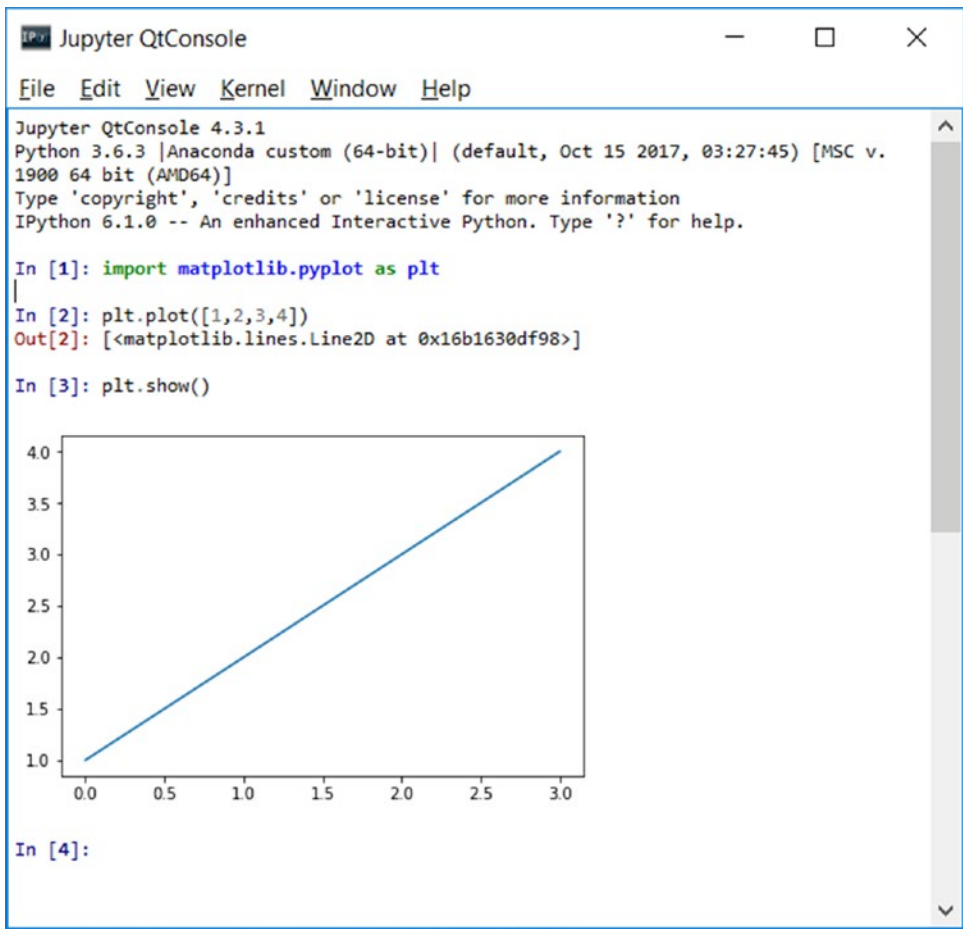


Figure 7-6. The QtConsole shows the chart directly as output

If you pass only a list of numbers or an array to the `plt.plot()` function, matplotlib assumes it is the sequence of y values of the chart, and it associates them to the natural sequence of values x: 0,1,2,3,

Generally a plot represents value pairs (x, y), so if you want to define a chart correctly, you must define two arrays, the first containing the values on the x-axis and the second containing the values on the y-axis. Moreover, the `plot()` function can accept a third argument, which describes the specifics of how you want the point to be represented on the chart.

Set the Properties of the Plot

As you can see in Figure 7-6, the points were represented by a blue line. In fact, if you do not specify otherwise, the plot is represented taking into account a default configuration of the `plt.plot()` function:

- The size of the axes matches perfectly with the range of the input data
- There is neither a title nor axis labels
- There is no legend
- A blue line connecting the points is drawn

Therefore you need to change this representation to have a real plot in which each pair of values (x, y) is represented by a red dot (see Figure 7-7).

If you're working on IPython, close the window to get back to the active prompt for entering new commands. Then you have to call back the `show()` function to observe the changes made to the plot.

```
In [4]: plt.plot([1,2,3,4],[1,4,9,16], 'ro')
Out[4]: [<matplotlib.lines.Line2D at 0x93e6898>]
```

```
In [5]: plt.show()
```

Instead, if you're working on Jupyter QtConsole you see a different plot for each new command you enter.

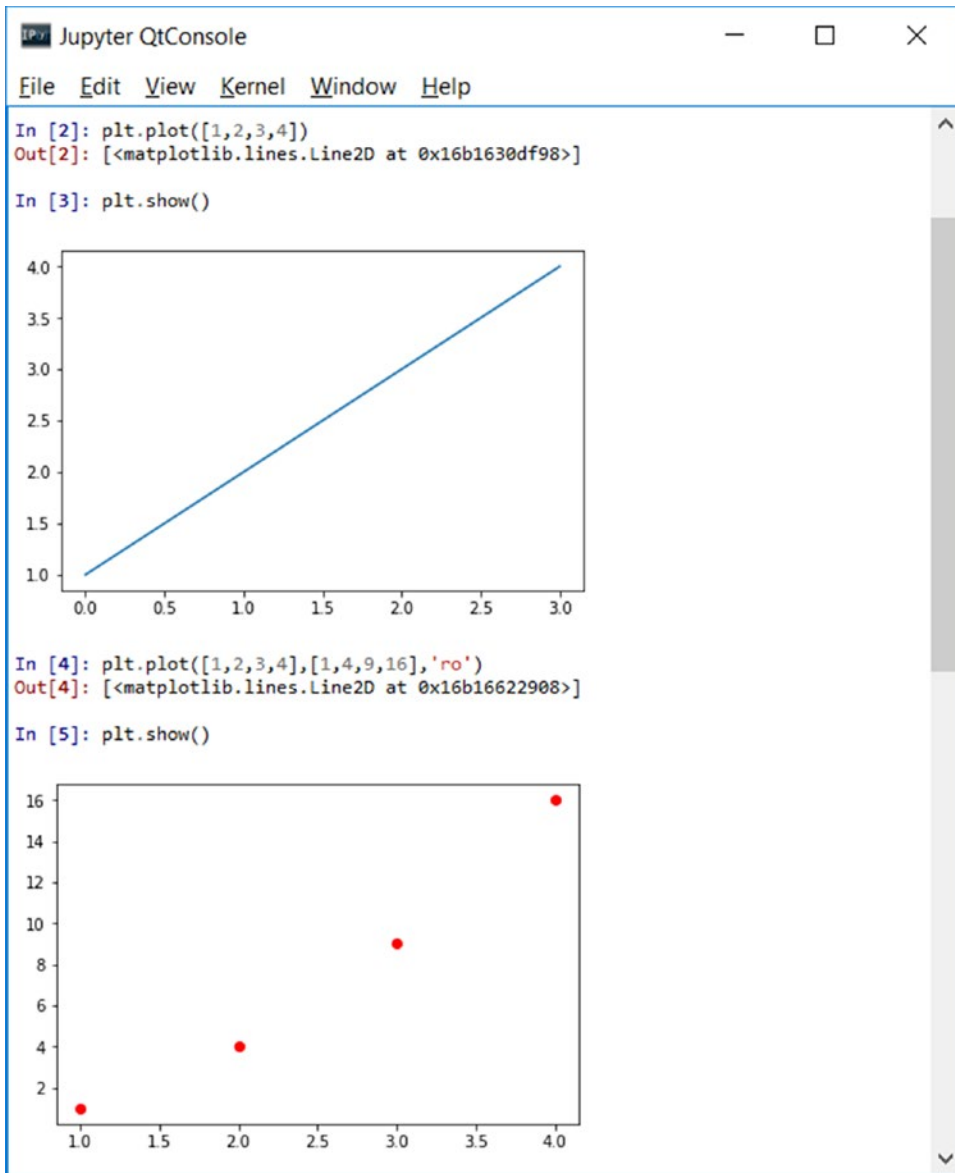


Figure 7-7. The pairs of (x,y) values are represented in the plot by red circles

Note At this point in the book, you already have a very clear idea about the difference between the various environments. To avoid confusion from this point, I will consider the IPython QtConsole as the sole development environment.

You can define the range both on the x-axis and on the y-axis by defining the details of a list `[xmin, xmax, ymin, ymax]` and then passing it as an argument to the `axis()` function.

Note In the IPython QtConsole, to generate a chart it is sometimes necessary to enter more rows of commands. To avoid generating a chart every time you press Enter (start a new line) along with losing the setting previously specified, you have to press Ctrl+Enter. When you want to finally generate the chart, just press Enter twice.

You can set several properties, one of which is the title that can be entered using the `title()` function.

```
In [4]: plt.axis([0,5,0,20])
...: plt.title('My first plot')
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[4]: [<matplotlib.lines.Line2D at 0x97f1c18>]
```

In Figure 7-8 you can see how the new settings made the plot more readable. In fact, the end points of the dataset are now represented within the plot rather than at the edges. Also the title of the plot is now visible at the top.

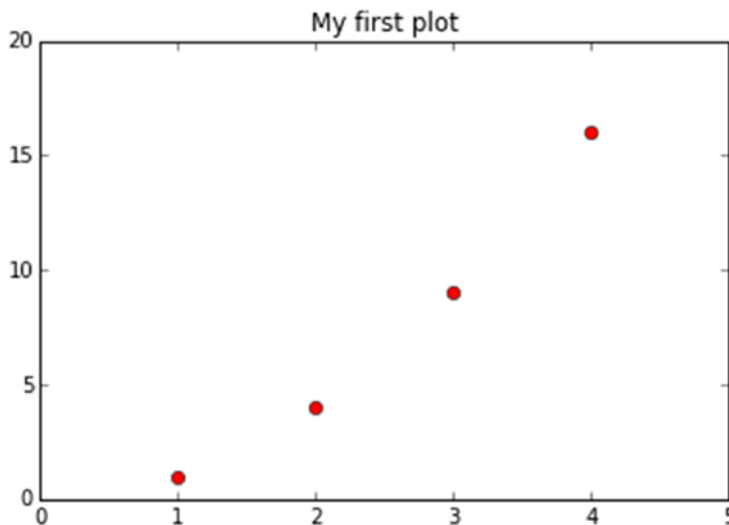


Figure 7-8. The plot after the properties have been set

matplotlib and NumPy

Even the `matplotlib` library, despite being a fully graphical library, has its foundation as the NumPy library. In fact, you have seen so far how to pass lists as arguments, both to represent the data and to set the extremes of the axes. Actually, these lists have been converted internally in NumPy arrays.

Therefore, you can directly enter NumPy arrays as input data. This array of data, which have been processed by `pandas`, can be directly used with `matplotlib` without further processing.

As an example, you see how it is possible to plot three different trends in the same plot (see Figure 7-9). You can choose for this example the `sin()` function belonging to the `math` module. So you will need to import it. To generate points following a sinusoidal trend, you will use the NumPy library. Generate a series of points on the x-axis using the `arange()` function, while for the values on the y-axis you will use the `map()` function to apply the `sin()` function on all the items of the array (without using a `for` loop).

```
In [5]: import math
In [6]: import numpy as np
In [7]: t = np.arange(0,2.5,0.1)
...: y1 = np.sin(math.pi*t)
...: y2 = np.sin(math.pi*t+math.pi/2)
...: y3 = np.sin(math.pi*t-math.pi/2)
In [8]: plt.plot(t,y1,'b*',t,y2,'g^',t,y3,'ys')
Out[8]:
[<matplotlib.lines.Line2D at 0xcbd2e48>,
 <matplotlib.lines.Line2D at 0xcbe10b8>,
 <matplotlib.lines.Line2D at 0xcbe15c0>]
```

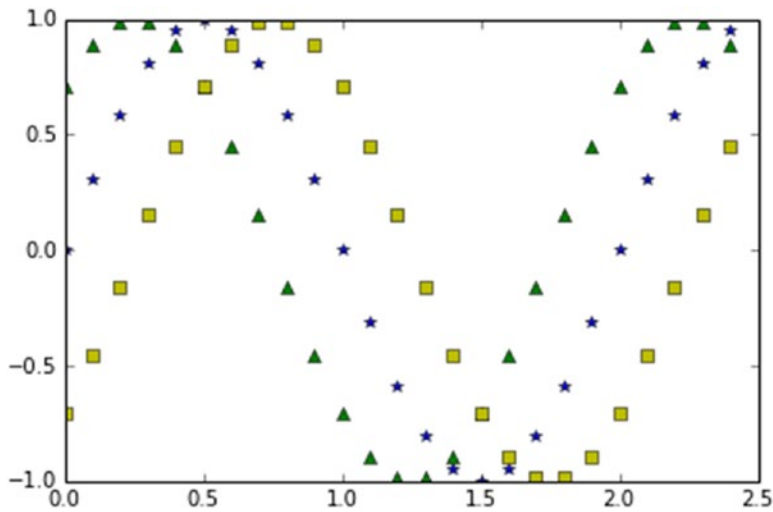


Figure 7-9. Three sinusoidal trends phase-shifted by $\pi / 4$ represented by markers

Note If you are not using the IPython QtConsole set with matplotlib inline or you are implementing this code on a simple Python session, insert the `plt.show()` command at the end of the code to obtain the chart shown in Figure 7-10.

As you can see in Figure 7-9, the plot represents the three different temporal trends with three different colors and markers. In these cases, when the trend of a function is so obvious, the plot is perhaps not the most appropriate representation, but it is better to use the lines (see Figure 7-10). To differentiate the three trends with something other than color, you can use the pattern composed of different combinations of dots and dashes (- and .).

```
In [9]: plt.plot(t,y1,'b--',t,y2,'g',t,y3,'r-.')
```

```
Out[9]:
```

```
[<matplotlib.lines.Line2D at 0xd1eb550>,  
<matplotlib.lines.Line2D at 0xd1eb780>,  
<matplotlib.lines.Line2D at 0xd1ebd68>]
```

Note If you are not using the IPython QtConsole set with matplotlib inline or you are implementing this code on a simple Python session, insert the `plt.show()` command at the end of the code to obtain the chart shown in Figure 7-10.

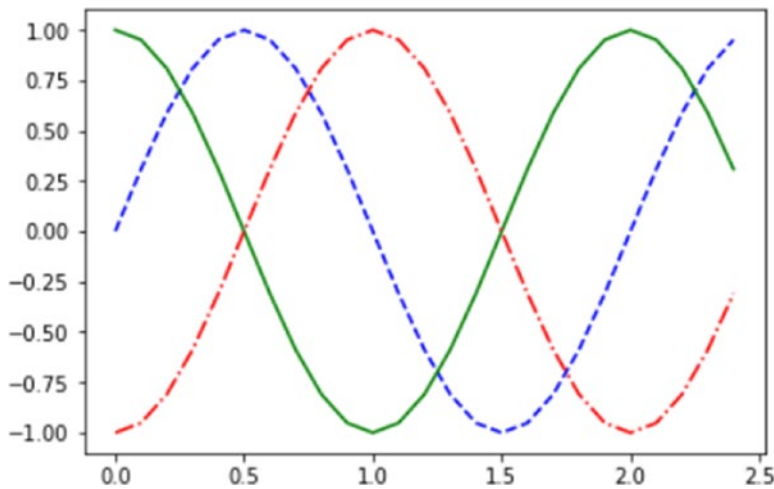


Figure 7-10. This chart represents the three sinusoidal patterns with colored lines

Using the kwargs

The objects that make up a chart have many attributes that characterize them. These attributes are all default values, but can be set through the use of *keyword args*, often referred as *kwargs*.

These keywords are passed as arguments to functions. In reference documentation of the various functions of the matplotlib library, you will always find them referred to as *kwargs* in the last position. For example the `plot()` function that you are using in these examples is referred to in the following way.

```
matplotlib.pyplot.plot(*args, **kwargs)
```

For a practical example, the thickness of a line can be changed if you set the `linewidth` keyword (see Figure 7-11).

```
In [10]: plt.plot([1,2,4,2,1,0,1,2,1,4],linewidth=2.0)
Out[10]: [matplotlib.lines.Line2D at 0xc909da0>]
```

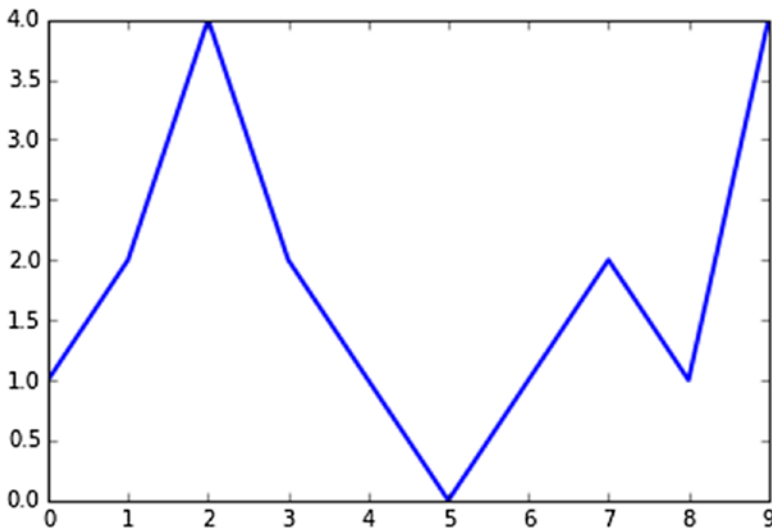



Figure 7-11. The thickness of a line can be set directly from the `plot()` function

Working with Multiple Figures and Axes

So far you have seen how all pyplot commands are routed to the display of a single figure. Actually, matplotlib allows you to manage multiple figures simultaneously, and within each figure, it offers the ability to view different plots defined as subplots.

So when you are working with pyplot, you must always keep in mind the concept of the current Figure and current Axes (that is, the plot shown within the figure).

Now you will see an example where two subplots are represented in a single figure. The `subplot()` function, in addition to subdividing the figure in different drawing areas, is used to focus the commands on a specific subplot.

The argument passed to the `subplot()` function sets the mode of subdivision and determines which is the current subplot. The current subplot will be the only figure that will be affected by the commands. The argument of the `subplot()` function is composed of three integers. The first number defines how many parts the figure is split into vertically. The second number defines how many parts the figure is divided into horizontally. The third issue selects which is the current subplot on which you can direct commands.

Now you will display two sinusoidal trends (sine and cosine) and the best way to do that is to divide the canvas vertically in two horizontal subplots (as shown in Figure 7-12). So the numbers to pass as an argument are 211 and 212.

```

In [11]: t = np.arange(0,5,0.1)
... : y1 = np.sin(2*np.pi*t)
... : y2 = np.sin(2*np.pi*t)
In [12]: plt.subplot(211)
...: plt.plot(t,y1,'b-.-')
...: plt.subplot(212)
...: plt.plot(t,y2,'r--')
Out[12]: [<matplotlib.lines.Line2D at 0xd47f518>]

```

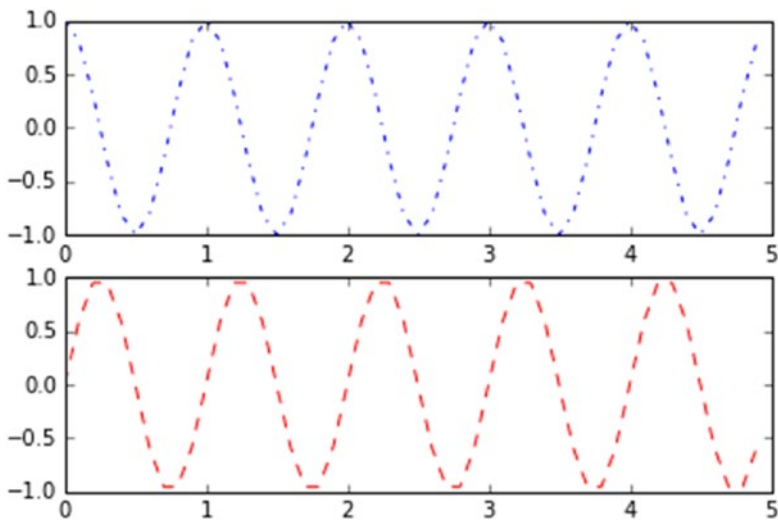


Figure 7-12. The figure has been divided into two horizontal subplots

Now you do the same thing by dividing the figure in two vertical subplots. The numbers to be passed as arguments to the `subplot()` function are 121 and 122 (as shown in Figure 7-13).

```

In [ ]: t = np.arange(0.,1.,0.05)
...: y1 = np.sin(2*np.pi*t)
...: y2 = np.cos(2*np.pi*t)
In [ ]: plt.subplot(121)
...: plt.plot(t,y1,'b-.-')
...: plt.subplot(122)
...: plt.plot(t,y2,'r--')
Out[94]: [<matplotlib.lines.Line2D at 0xed0c208>]

```

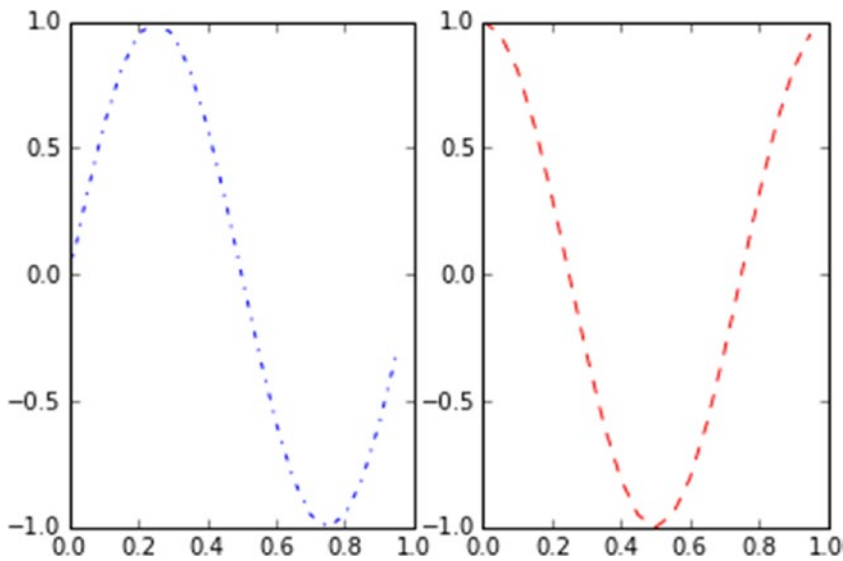


Figure 7-13. *The figure has been divided into two vertical subplots*

Adding Elements to the Chart

In order to make a chart more informative, many times it is not enough to represent the data using lines or markers and assign the range of values using two axes. In fact, there are many other elements that can be added to a chart in order to enrich it with additional information.

In this section you will see how to add elements to the chart as text labels, a legend, and so on.

Adding Text

You've already seen how you can add the title to a chart with the `title()` function. Two other textual indications you can add the *axis labels*. This is possible through the use of two other specific functions, called `xlabel()` and `ylabel()`. These functions take as an argument a string, which will be the shown text.

Note Command lines forming the code to represent your chart are growing in number. You do not need to rewrite all the commands each time, but using the arrow keys on the keyboard, you can call up the list of commands previously passed and edit them by adding new rows (in the text are indicated in bold).

Now add two axis labels to the chart. They will describe which kind of value is assigned to each axis (as shown in Figure 7-14).

```
In [10]: plt.axis([0,5,0,20])
...: plt.title('My first plot')
...: plt.xlabel('Counting')
...: plt.ylabel('Square values')
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[10]: [<matplotlib.lines.Line2D at 0x990f3c8>]
```

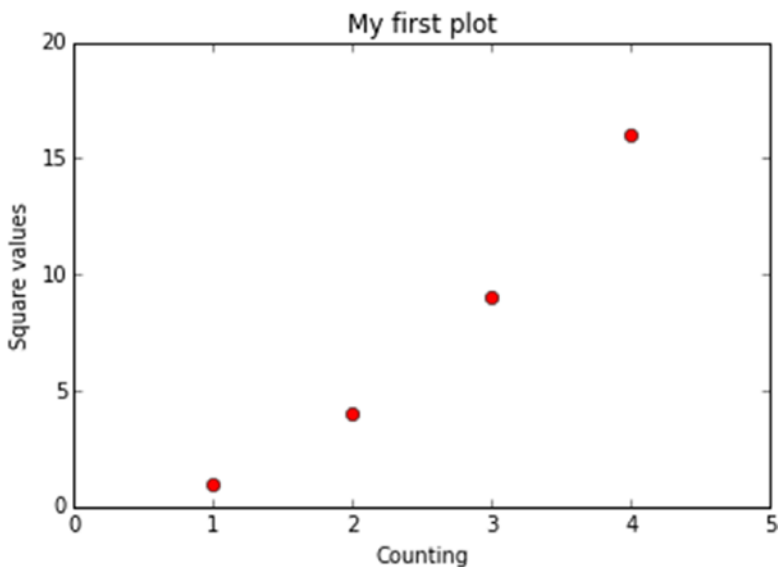


Figure 7-14. A plot is more informative by adding axis labels

Thanks to the keywords, you can change the characteristics of the text. For example, you can modify the title by changing the font and increasing the size of the characters. You can also modify the color of the axis labels to accentuate the title of the plot (as shown in Figure 7-15).

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[116]: [<matplotlib.lines.Line2D at 0x11f17470>]
```

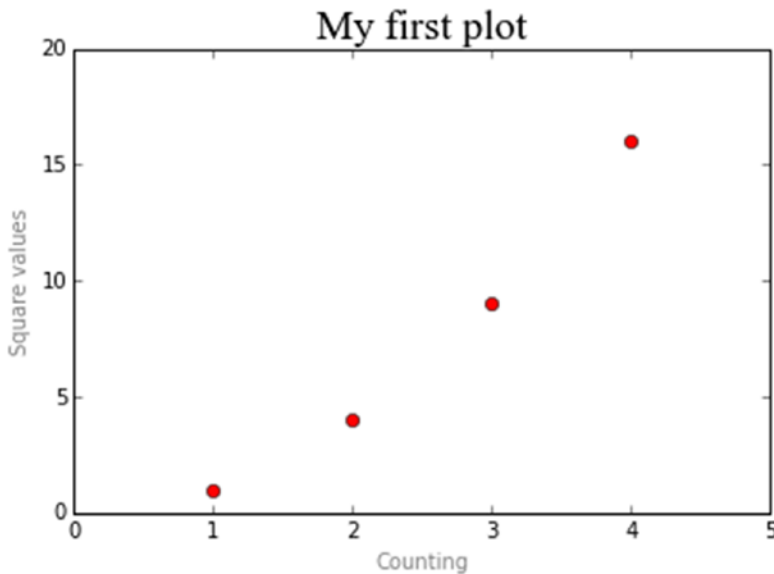


Figure 7-15. The text can be modified by setting the keywords

But matplotlib is not limited to this: pyplot allows you to add text to any position within a chart. This feature is performed by a specific function called `text()`.

```
text(x,y,s, fontdict=None, **kwargs)
```

The first two arguments are the coordinates of the location where you want to place the text. `s` is the string of text to be added, and `fontdict` (optional) is the font that you want to use. Finally, you can add the keywords.

Add the label to each point of the plot. Because the first two arguments to the `text()` function are the coordinates of the graph, you have to use the coordinates of the four points of the plot shifted slightly on the y-axis.

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[108]: [<matplotlib.lines.Line2D at 0x10f76898>]
```

As you can see in Figure 7-16, now each point of the plot has a label.

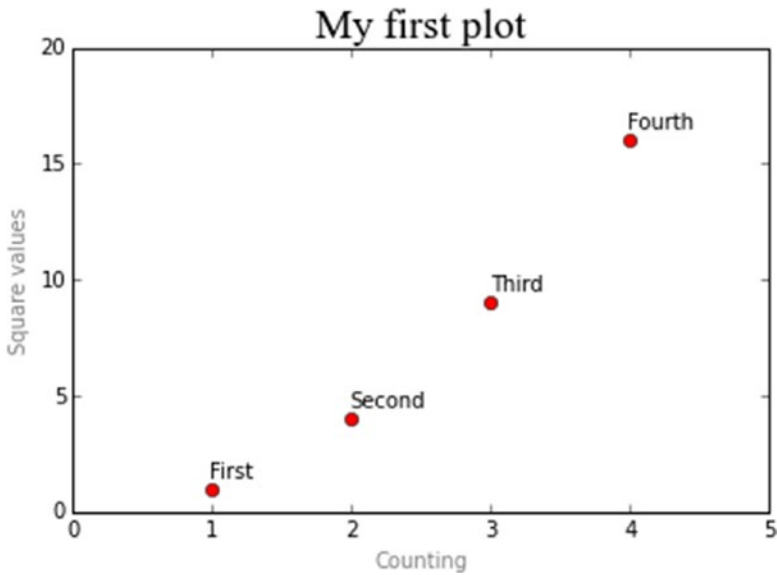


Figure 7-16. Every point of the plot has an informative label

Since matplotlib is a graphics library designed to be used in scientific circles, it must be able to exploit the full potential of scientific language, including mathematical expressions. matplotlib offers the possibility to integrate LaTeX expressions, thereby allowing you to insert mathematical expressions within the chart.

To do this, you can add a LaTeX expression to the text, enclosing it between two \$ characters. The interpreter will recognize them as LaTeX expressions and convert them to the corresponding graphic, which can be a mathematical expression, a formula, mathematical characters, or just Greek letters. Generally you have to precede the string containing LaTeX expressions with an `r`, which indicates raw text, in order to avoid unintended escape sequences.

Here, you can also use the keywords to further enrich the text to be shown in the plot. Therefore, as an example, you can add the formula describing the trend followed by the point of the plot and enclose it in a colored bounding box (see Figure 7-17).

```

In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,r'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
    'alpha':0.2})
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')

```

```
Out[130]: [matplotlib.lines.Line2D at 0x13920860]
```

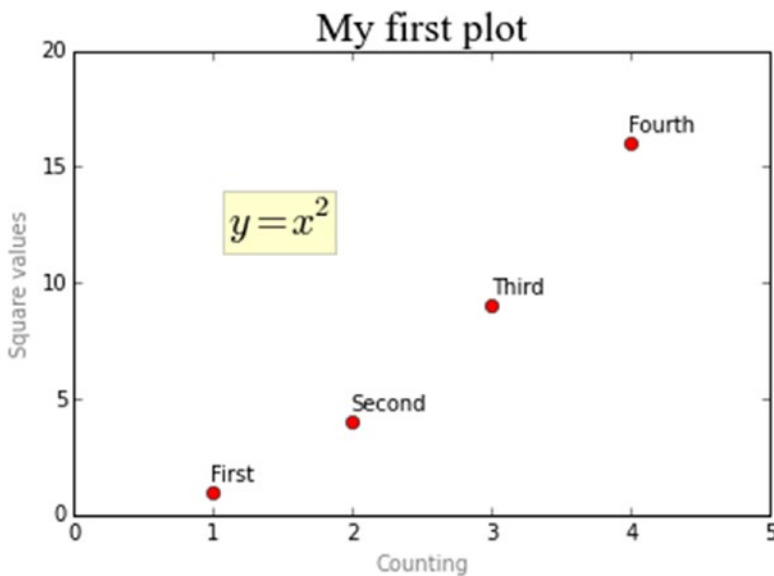


Figure 7-17. Any mathematical expression can be seen in the context of a chart

To get a complete view on the potential offered by LaTeX, consult Appendix A of this book.

Adding a Grid

Another element you can add to a plot is a grid. Often its addition is necessary in order to better understand the position occupied by each point on the chart.

Adding a grid to a chart is a very simple operation: just add the `grid()` function, passing `True` as an argument (see Figure 7-18).

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,r'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
    'alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[108]: [<matplotlib.lines.Line2D at 0x10f76898>]
```

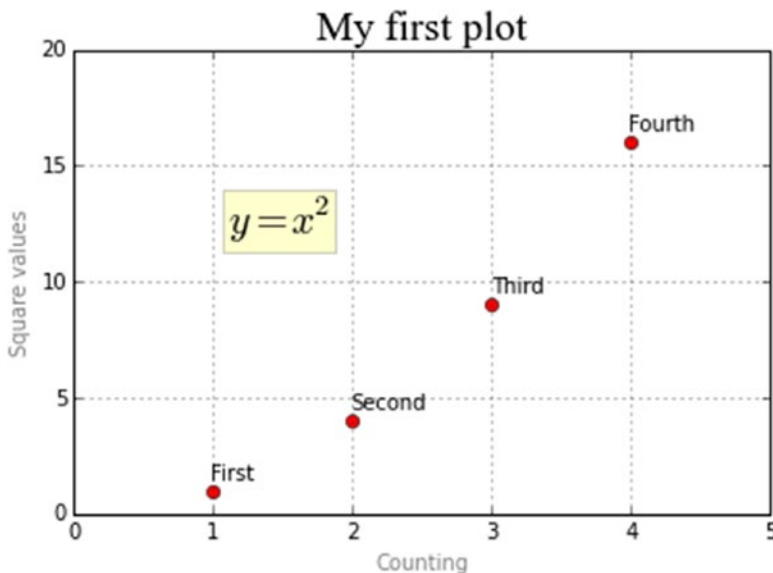


Figure 7-18. A grid makes it easier to read the values of the data points represented on a chart

Adding a Legend

Another very important component that should be present in any chart is the legend. pyplot also provides a specific function for this type of object: `legend()`.

Add a legend to your chart with the `legend()` function and a string indicating the words with which you want the series to be shown. In this example, you assign the First series name to the input data array (see Figure 7-19).

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
    'alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
...: plt.legend(['First series'])
Out[156]: <matplotlib.legend.Legend at 0x16377550>
```

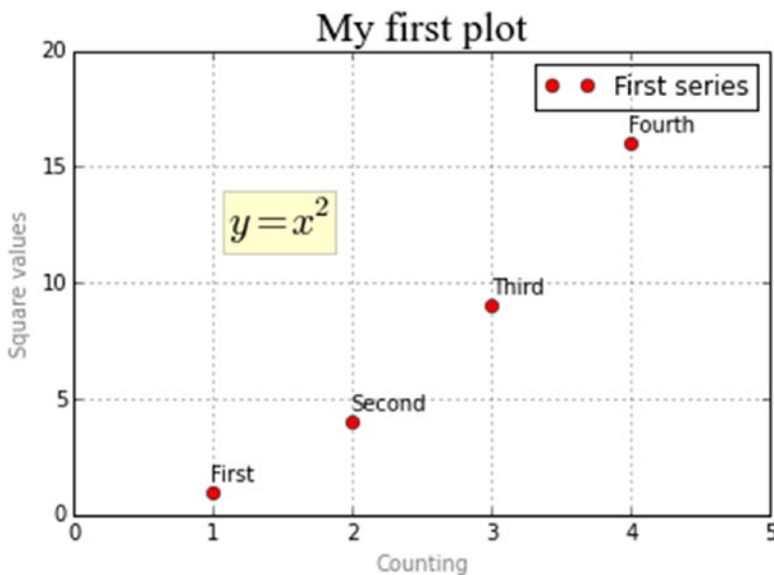


Figure 7-19. A legend is added in the upper-right corner by default

As you can see in Figure 7-19, the legend is added in the upper-right corner by default. Again if you want to change this behavior you will need to add a few kwargs. For example, the position occupied by the legend is set by assigning numbers from 0 to 10 to the `loc` kwarg. Each of these numbers characterizes one of the corners of the chart (see Table 7-1). A value of 1 is the default, that is, the upper-right corner. In the next example, you will move the legend in the upper-left corner so it will not overlap with the points represented in the plot.

Table 7-1. *The Possible Values for the loc Keyword*

Location Code	Location String
0	best
1	upper-right
2	upper-left
3	lower-right
4	lower-left
5	right
6	center-left
7	center-right
8	lower-center
9	upper-center
10	center

Before you begin to modify the code to move the legend, I want to add a small notice. Generally, the legends are used to indicate the definition of a series to the reader via a label associated with a color and/or a marker that distinguishes it in the plot. So far in the examples, you have used a single series that was expressed by a single `plot()` function. Now, you have to focus on a more general case in which the same plot shows more series simultaneously. Each series in the chart will be characterized by a specific color and a specific marker (see Figure 7-20). In terms of code, instead, each series will be characterized by a call to the `plot()` function and the order in which they are defined will correspond to the order of the text labels passed as an argument to the `legend()` function.

```

In [ ]: import matplotlib.pyplot as plt
...: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
...: 'alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
...: plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
...: plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
...: plt.legend(['First series','Second series','Third series'],loc=2)
Out[170]: <matplotlib.legend.Legend at 0x1828d7b8>

```

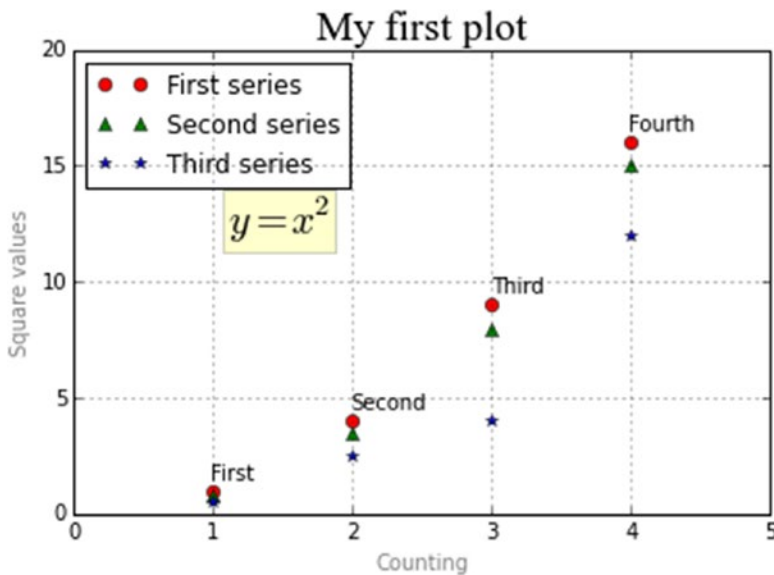


Figure 7-20. A legend is necessary in every multiseries chart

Saving Your Charts

In this section you will learn how to save your chart in different ways depending on your needs. If you need to reproduce your chart in different notebooks or Python sessions, or reuse them in future projects, it is a good practice to save the Python code. On the other hand, if you need to make reports or presentations, it can be very useful to save your chart as an image. Moreover, it is possible to save your chart as a HTML page, and this could be very useful when you need to share your work on Web.

Saving the Code

As you can see from the examples in the previous sections, the code concerning the representation of a single chart is growing into a fair number of rows. Once you think you've reached a good point in your development process, you can choose to save all rows of code in a `.py` file that you can recall at any time.

You can use the magic command `save%` followed by the name of the file you want to save followed by the number of input prompts containing the row of code that you want to save. If all the code is written in only one prompt, as your case, you have to add only its number; otherwise if you want to save the code written in many prompts, for example from 10 to 20, you have to indicate this range with the two numbers separated by a `-`, that is, `10-20`.

In your case, you would save the Python code underlying the representation of your first chart contained into the input prompt with the number 171.

```
In [171]: import matplotlib.pyplot as plt
...
```

You need to insert the following command to save the code into a new `.py` file.

```
%save my_first_chart 171
```

After you launch the command, you will find the `my_first_chart.py` file in your working directory (see Listing 7-1).

Listing 7-1. my_first_chart.py

```
# coding: utf-8
import matplotlib.pyplot as plt
plt.axis([0,5,0,20])
plt.title('My first plot',fontsize=20,fontname='Times New Roman')
plt.xlabel('Counting',color='gray')
plt.ylabel('Square values',color='gray')
plt.text(1,1.5,'First')
plt.text(2,4.5,'Second')
plt.text(3,9.5,'Third')
plt.text(4,16.5,'Fourth')
plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
'alpha':0.2})
plt.grid(True)
plt.plot([1,2,3,4],[1,4,9,16],'ro')
plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
plt.legend(['First series','Second series','Third series'],loc=2)
```

Later, when you open a new IPython session, you will have your chart and start to change the code at the point where you had saved it by entering the following command:

```
ipython qtconsole --matplotlib inline -m my_first_chart.py
```

Or you can reload the entire code in a single prompt in the QtConsole using the magic command `%load`.

```
%load my_first_chart.py
```

Or you can run it during a session with the magic command `%run`.

```
%run my_first_chart.py
```

Note On my system, this command works only after launching the two previous commands.

Converting Your Session to an HTML File

Using the IPython QtConsole, you can convert all the code and graphics present in your current session to an HTML page. Simply choose *File-->Save to HTML/XHTML* from the menu (as shown in Figure 7-21).

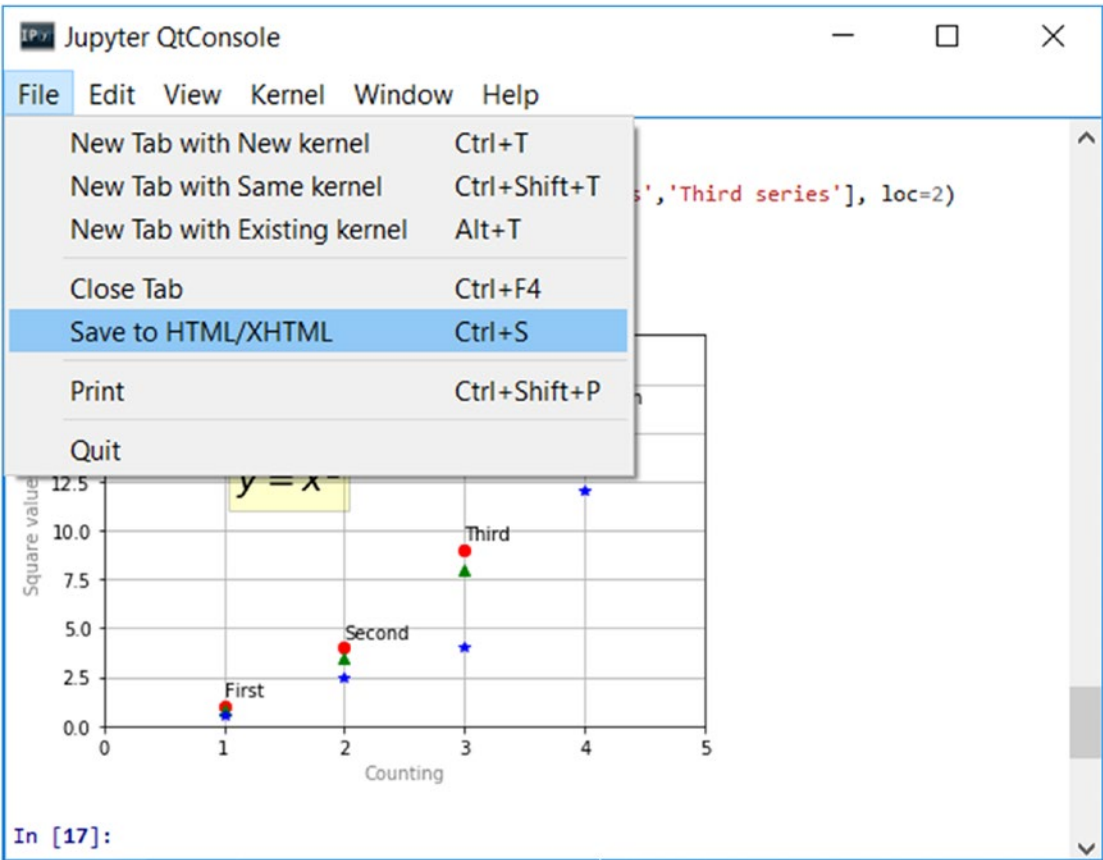


Figure 7-21. You can save your current session as a web page

You will be asked to save your session in two different formats: HTML and XHTML. The difference between the two formats is based on the image conversion type. If you select HTML as the output file format, the images contained in your session will be converted to PNG format. If you select XHTML as the output file format instead, the images will be converted to SVG format.

In this example, save your session as an HTML file and name it `my_session.html`, as shown in Figure 7-22.

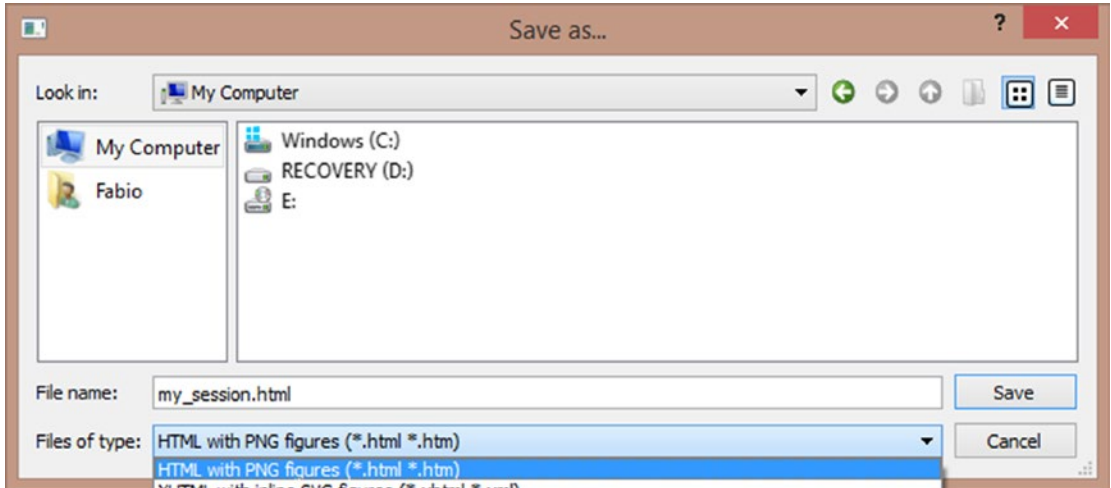


Figure 7-22. You can select the type of file between HTML and XHTML

At this point, you will be asked if you want to save your images in an external directory or inline (see Figure 7-23).

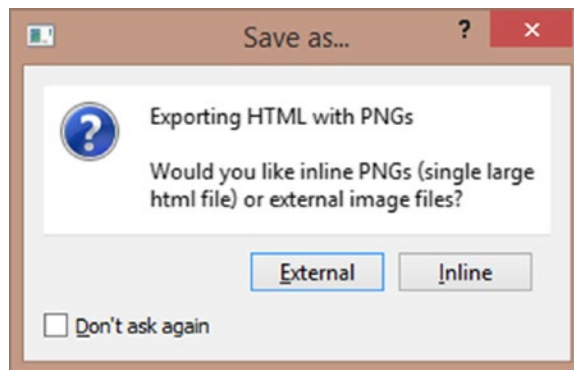


Figure 7-23. You can choose between creating external image files and embedding the PNG format directly into the HTML page

By choosing the external option, the images will be collected into a directory called `my_session_files`. By choosing inline, the graphical information concerning the image is embedded into the HTML code.

Saving Your Chart Directly as an Image

If you are interested in saving only the figure of a chart as an image file, ignoring all the code you've written during the session, this is also possible. In fact, thanks to the `savefig()` function, you can directly save the chart in a PNG format, although you should take care to add this function to the end of the same series of commands (otherwise you'll get a blank PNG file).

```
In [ ]: plt.axis([0,5,0,20])
...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
...: plt.xlabel('Counting',color='gray')
...: plt.ylabel('Square values',color='gray')
...: plt.text(1,1.5,'First')
...: plt.text(2,4.5,'Second')
...: plt.text(3,9.5,'Third')
...: plt.text(4,16.5,'Fourth')
...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow',
    'alpha':0.2})
...: plt.grid(True)
...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
...: plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
...: plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
...: plt.legend(['First series','Second series','Third series'],loc=2)
...: plt.savefig('my_chart.png')
```

Executing the previous code, a new file will be created in your working directory. This file will be named `my_chart.png` and will contain the image of your chart.

Handling Date Values

One of the most common problems encountered when doing data analysis is handling data of the date-time type. Displaying that data along an axis (normally the x-axis) can be problematic, especially when managing ticks (see Figure 7-24).

Take for example the display of a linear chart with a dataset of eight points in which you have to represent date values on the x-axis with the following format: day-month-year.

```
In [ ]: import datetime
...: import numpy as np
...: import matplotlib.pyplot as plt
...: events = [datetime.date(2015,1,23),datetime.
    date(2015,1,28),datetime.date(2015,2,3),datetime.
    date(2015,2,21),datetime.date(2015,3,15),datetime.
    date(2015,3,24),datetime.date(2015,4,8),datetime.date(2015,4,24)]
...: readings = [12,22,25,20,18,15,17,14]
...: plt.plot(events,readings)
Out[83]: [<matplotlib.lines.Line2D at 0x12666400>]
```

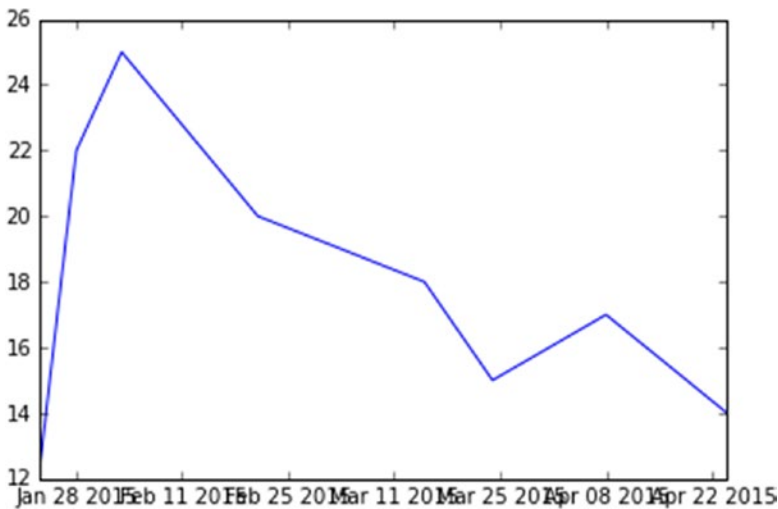


Figure 7-24. *If not handled, displaying date-time values can be problematic*

As you can see in Figure 7-24, automatic management of ticks, and especially the tick labels, can be a disaster. The dates expressed in this way are difficult to read, there are no clear time intervals elapsed between one point and another, and there is also overlap.

To manage dates it is therefore advisable to define a time scale with appropriate objects. First you need to import `matplotlib.dates`, a module specialized for this type of data. Then you define the scales of the times, as in this case, a scale of days and one of the months, through the `MonthLocator()` and `DayLocator()` functions. In these cases, the formatting is also very important, and to avoid overlap or unnecessary references, you have to limit the tick labels to the essential, which in this case is year-month. This format can be passed as an argument to the `DateFormatter()` function.

After you defined the two scales, one for the days and one for the months, you can set two different kinds of ticks on the x-axis, using the `set_major_locator()` and `set_minor_locator()` functions on the `xaxis` object. Instead, to set the text format of the tick labels referred to the months you have to use the `set_major_formatter()` function.

Changing all these settings you finally obtain the plot as shown in Figure 7-25.

```
In [ ]: import datetime
...: import numpy as np
...: import matplotlib.pyplot as plt
...: import matplotlib.dates as mdates
...: months = mdates.MonthLocator()
...: days = mdates.DayLocator()
...: timeFmt = mdates.DateFormatter('%Y-%m')
...: events = [datetime.date(2015,1,23),datetime.
...:             date(2015,1,28),datetime.date(2015,2,3),datetime.
...:             date(2015,2,21),datetime.date(2015,3,15),datetime.
...:             date(2015,3,24),datetime.date(2015,4,8),datetime.date(2015,4,24)]
readings = [12,22,25,20,18,15,17,14]
...: fig, ax = plt.subplots()
...: plt.plot(events,readings)
...: ax.xaxis.set_major_locator(months)
...: ax.xaxis.set_major_formatter(timeFmt)
...: ax.xaxis.set_minor_locator(days)
```

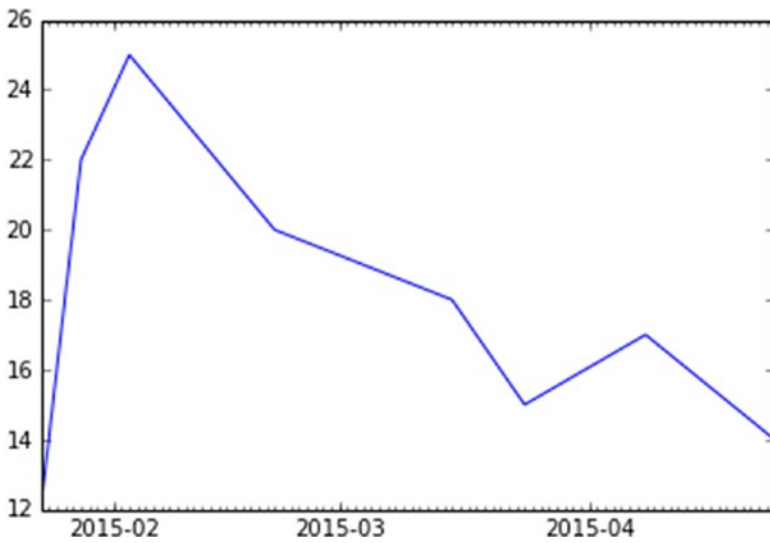


Figure 7-25. Now the tick labels of the x-axis refer only to the months, making the plot more readable

Chart Typology

In the previous sections you saw a number of examples relating to the architecture of the matplotlib library. Now that you are familiar with the use of the main graphic elements in a chart, it is time to see a series of examples treating different types of charts, starting from the most common ones such as linear charts, bar charts, and pie charts, up to a discussion about some that are more sophisticated but commonly used nonetheless.

This part of the chapter is very important since the purpose of this library is the visualization of the results produced by data analysis. Thus, knowing how to choose the proper type of chart is a fundamental choice. Remember that excellent data analysis represented incorrectly can lead to a wrong interpretation of the experimental results.

Line Charts

Among all the chart types, the linear chart is the simplest. A line chart is a sequence of data points connected by a line. Each data point consists of a pair of values (x,y), which will be reported in the chart according to the scale of values of the two axes (x and y).

By way of example, you can begin to plot the points generated by a mathematical function. Then, you can consider a generic mathematical function such as this:

$$y = \sin(3 * x) / x$$

Therefore, if you want to create a sequence of data points, you need to create two NumPy arrays. First you create an array containing the x values to be referred to the x-axis. In order to define a sequence of increasing values you will use the `np.arange()` function. Since the function is sinusoidal you should refer to values that are multiples and submultiples of the Greek pi (`np.pi`). Then, using these sequence of values, you can obtain the y values applying the `np.sin()` function directly to these values (thanks to NumPy!).

After all this, you have only to plot them by calling the `plot()` function. You will obtain a line chart, as shown in Figure 7-26.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi,2*np.pi,0.01)
...: y = np.sin(3*x)/x
...: plt.plot(x,y)
Out[393]: [<matplotlib.lines.Line2D at 0x22404358>]
```

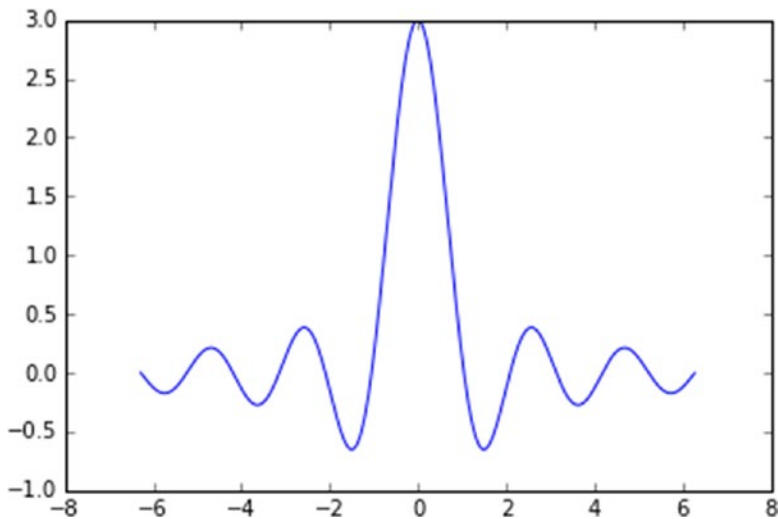


Figure 7-26. A mathematical function represented in a line chart

Now you can extend the case in which you want to display a family of functions, such as this:

$$y = \sin(n * x) / x$$

varying the parameter n .

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi,2*np.pi,0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(3*x)/x
...: plt.plot(x,y)
...: plt.plot(x,y2)
...: plt.plot(x,y3)
```

As you can see in Figure 7-27, a different color is automatically assigned to each line. All the plots are represented on the same scale; that is, the data points of each series refer to the same x-axis and y-axis. This is because each call of the `plot()` function takes into account the previous calls to same function, so the Figure applies the changes keeping memory of the previous commands until the Figure is not displayed (using `show()` with Python and Enter with the IPython QtConsole).

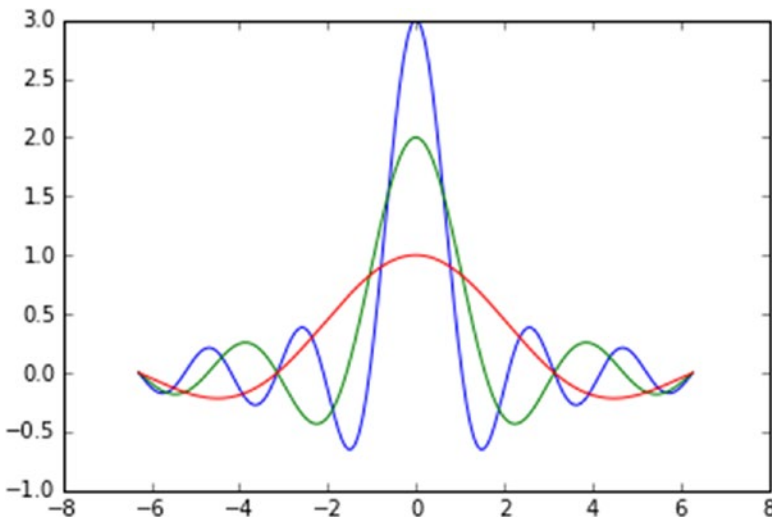


Figure 7-27. Three different series are drawn with different colors in the same chart

As you saw in the previous sections, regardless of the default settings, you can select the type of stroke, color, etc. As the third argument of the `plot()` function you can specify some codes that correspond to the color (see Table 7-2) and other codes that correspond to line styles, all included in the same string. Another possibility is to use two kwargs separately, `color` to define the color, and `linestyle` to define the stroke (see Figure 7-28).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi,2*np.pi,0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(3*x)/x
...: plt.plot(x,y,'k--',linewidth=3)
...: plt.plot(x,y2,'m-.')
...: plt.plot(x,y3,color='#87a3cc',linestyle='--')
```

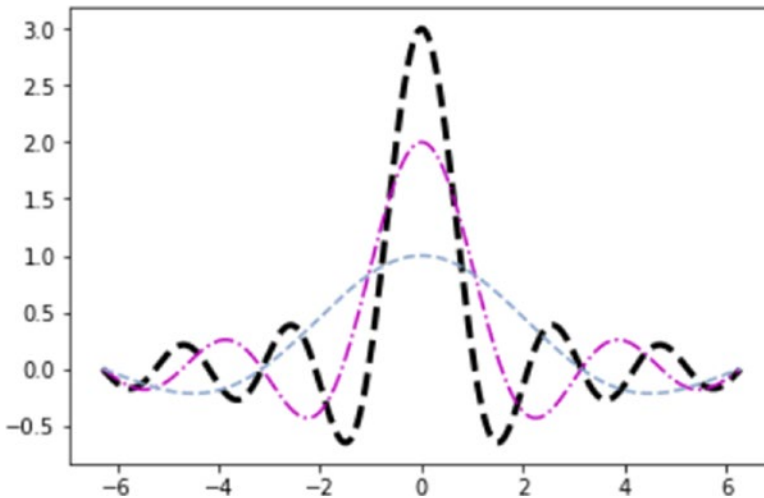


Figure 7-28. You can define colors and line styles using character codes

Table 7-2. *Color Codes*

Code	Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

You have just defined a range from -2π to 2π on the x-axis, but by default, values on ticks are shown in numerical form. Therefore you need to replace the numerical values with multiple of π . You can also replace the ticks on the y-axis. To do all this, you have to use `xticks()` and `yticks()` functions, passing to each of them two lists of values. The first list contains values corresponding to the positions where the ticks are to be placed, and the second contains the tick labels. In this particular case, you have to use strings containing LaTeX format in order to correctly display the symbol π . Remember to define them within two `$` characters and to add a `r` as the prefix.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi, 2*np.pi, 0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(x)/x
...: plt.plot(x, y, color='b')
...: plt.plot(x, y2, color='r')
...: plt.plot(x, y3, color='g')
...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
...:             [r'$-2\pi$', r'$-\pi$', r'$0$', r'$+\pi$', r'$+2\pi$'])
```

```
...: plt.yticks([-1,0,1,2,3],
               [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
```

Out[423]:

```
([<matplotlib.axis.YTick at 0x26877ac8>,
  <matplotlib.axis.YTick at 0x271d26d8>,
  <matplotlib.axis.YTick at 0x273c7f98>,
  <matplotlib.axis.YTick at 0x273cc470>,
  <matplotlib.axis.YTick at 0x273cc9e8>],
  <a list of 5 Text yticklabel objects>)
```

In the end, you will get a clean and pleasant line chart showing Greek characters, as in Figure 7-29.

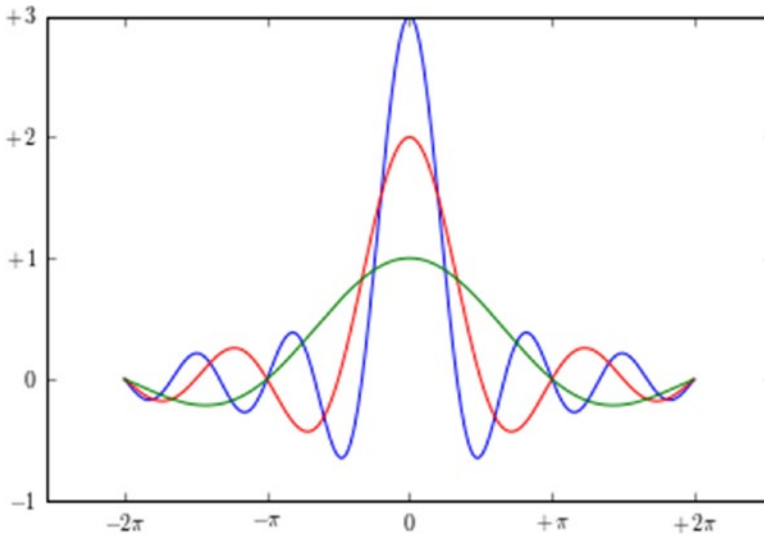


Figure 7-29. The tick label can be improved adding text with LaTeX format

In all the linear charts you have seen so far, you always have the x-axis and y-axis placed at the edge of the figure (corresponding to the sides of the bounding border box). Another way of displaying axes is to have the two axes passing through the origin (0, 0), i.e., the two Cartesian axes.

To do this, you must first capture the Axes object through the `gca()` function. Then through this object, you can select each of the four sides making up the bounding box, specifying for each one its position: right, left, bottom, and top. Crop the sides that do not match any axis (right and bottom) using the `set_color()` function and indicating none for color. Then, the sides corresponding to the x- and y-axes are moved to pass through the origin (0,0) with the `set_position()` function.

```
In [ ]: import matplotlib.pyplot as plt
....: import numpy as np
....: x = np.arange(-2*np.pi,2*np.pi,0.01)
....: y = np.sin(3*x)/x
....: y2 = np.sin(2*x)/x
....: y3 = np.sin(x)/x
....: plt.plot(x,y,color='b')
....: plt.plot(x,y2,color='r')
....: plt.plot(x,y3,color='g')
....: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
....:             [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
....: plt.yticks([-1,0,+1,+2,+3],
....:             [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
....: ax = plt.gca()
....: ax.spines['right'].set_color('none')
....: ax.spines['top'].set_color('none')
....: ax.xaxis.set_ticks_position('bottom')
....: ax.spines['bottom'].set_position(('data',0))
....: ax.yaxis.set_ticks_position('left')
....: ax.spines['left'].set_position(('data',0))
```

Now the chart will show the two axes crossing in the middle of the figure, that is, the origin of the Cartesian axes, as shown in Figure 7-30.

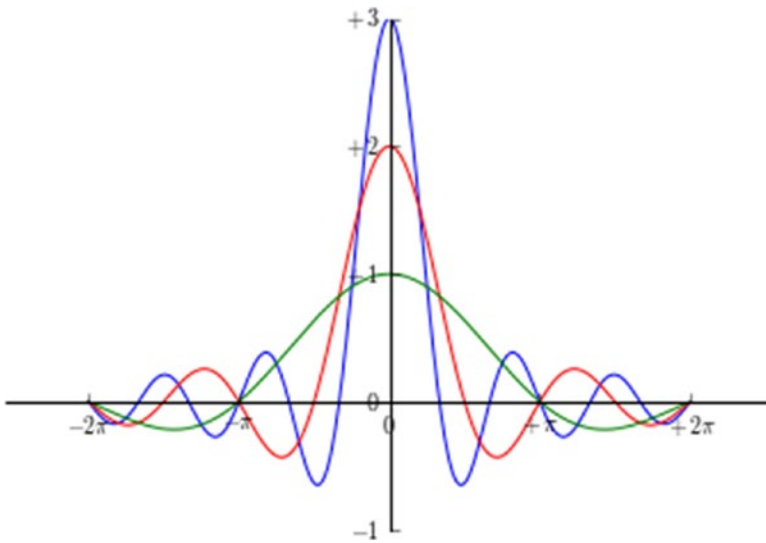


Figure 7-30. *The chart shows two Cartesian axes*

Often, it is very useful to be able to specify a particular point of the line using a notation and optionally add an arrow to better indicate the position of the point. For example, this notation may be a LaTeX expression, such as the formula for the limit of the function $\sin x/x$ with x tends to 0.

In this regard, matplotlib provides a function called `annotate()`, which is especially useful in these cases, even if the numerous kwargs needed to obtain a good result can make its settings quite complex. The first argument is the string to be represented containing the expression in LaTeX; then you can add the various kwargs. The point of the chart to note is indicated by a list containing the coordinates of the point $[x, y]$ passed to the `xy` kwarg. The distance of the textual notation from the point to be highlighted is defined by the `xytext` kwarg and represented by means of a curved arrow whose characteristics are defined in the `arrowprops` kwarg.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: x = np.arange(-2*np.pi, 2*np.pi, 0.01)
...: y = np.sin(3*x)/x
...: y2 = np.sin(2*x)/x
...: y3 = np.sin(x)/x
...: plt.plot(x, y, color='b')
...: plt.plot(x, y2, color='r')
```

```

...: plt.plot(x,y3,color='g')
...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
    [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
...: plt.yticks([-1,0,+1,+2,+3],
    [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
...: plt.annotate(r'$\lim_{x\to 0}\frac{\sin(x)}{x}= 1$', xy=[0,1],
    xycoords='data',xytext=[30,30],fontsize=16,textcoords='offset points',
    arrowprops=dict(arrowstyle="->",connectionstyle="arc3,rad=.2"))
...: ax = plt.gca()
...: ax.spines['right'].set_color('none')
...: ax.spines['top'].set_color('none')
...: ax.xaxis.set_ticks_position('bottom')
...: ax.spines['bottom'].set_position(('data',0))
...: ax.yaxis.set_ticks_position('left')
...: ax.spines['left'].set_position(('data',0))

```

Running this code, you will get the chart with the mathematical notation of the limit, which is the point shown by the arrow in Figure 7-31.

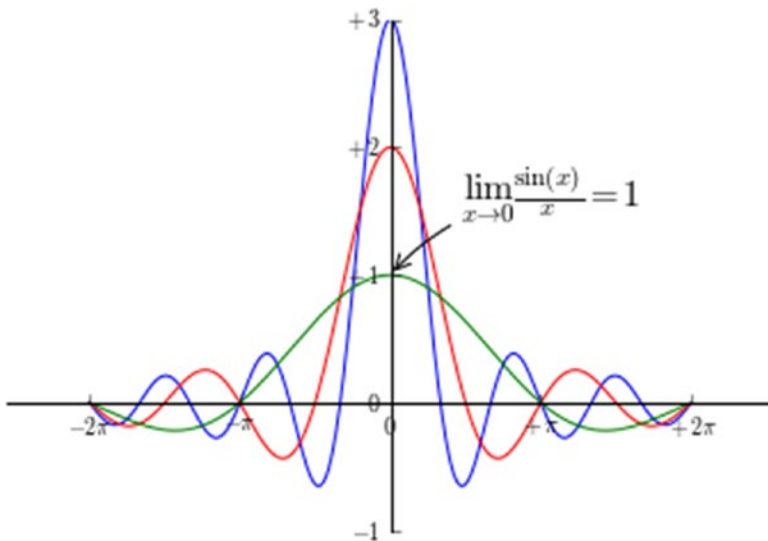


Figure 7-31. Mathematical expressions can be added to a chart with the `annotate()` function

Line Charts with pandas

Moving to more practical cases, or at least more closely related to data analysis, now is the time to see how easy it is to apply the matplotlib library to the dataframes of the pandas library. The visualization of the data in a dataframe as a linear chart is a very simple operation. It is sufficient to pass the dataframe as an argument to the `plot()` function to obtain a multiseries linear chart (see Figure 7-32).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: import pandas as pd
...: data = {'series1':[1,3,4,3,5],
...:         'series2':[2,4,5,2,4],
...:         'series3':[3,2,3,1,3]}
...: df = pd.DataFrame(data)
...: x = np.arange(5)
...: plt.axis([0,5,0,7])
...: plt.plot(x,df)
...: plt.legend(data, loc=2)
```

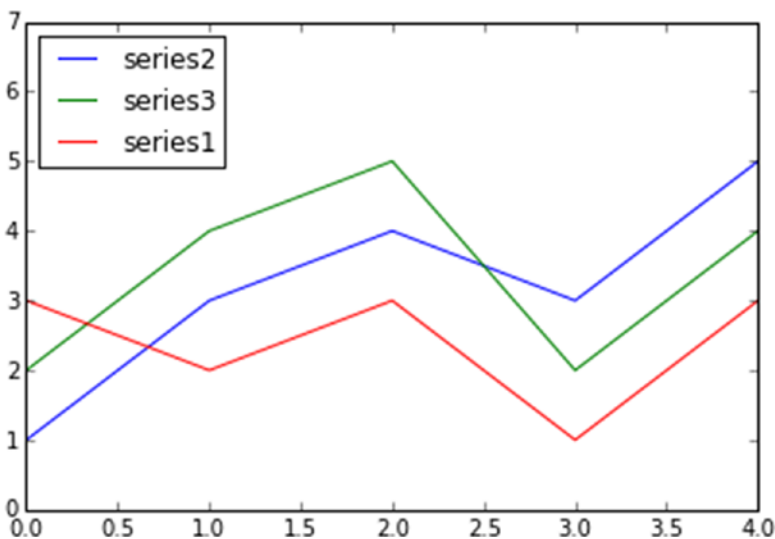


Figure 7-32. The multiseries line chart displays the data within a pandas dataframe

Histograms

A *histogram* consists of adjacent rectangles erected on the x-axis, split into discrete intervals called *bins*, and with an area proportional to the frequency of the occurrences for that bin. This kind of visualization is commonly used in statistical studies about distribution of samples.

In order to represent a histogram, pyplot provides a special function called `hist()`. This graphic function also has a feature that other functions producing charts do not have. The `hist()` function, in addition to drawing the histogram, returns a tuple of values that are the results of the calculation of the histogram. In fact the `hist()` function can also implement the calculation of the histogram, that is, it is sufficient to provide a series of samples of values as an argument and the number of bins in which to be divided, and it will take care of dividing the range of samples in many intervals (bins), and then calculate the occurrences for each bin. The result of this operation, in addition to being shown in graphical form (see Figure 7-33), will be returned in the form of a tuple.

(n, bins, patches)

To understand this operation, a practical example is best. Then you can generate a population of 100 random values from 0 to 100 using the `random.randint()` function.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: pop = np.random.randint(0,100,100)
...: pop
Out[ ]:
array([32, 14, 55, 33, 54, 85, 35, 50, 91, 54, 44, 74, 77,  6, 77, 74,  2,
       54, 14, 30, 80, 70,  6, 37, 62, 68, 88,  4, 35, 97, 50, 85, 19, 90,
       65, 86, 29, 99, 15, 48, 67, 96, 81, 34, 43, 41, 21, 79, 96, 56, 68,
       49, 43, 93, 63, 26,  4, 21, 19, 64, 16, 47, 57,  5, 12, 28,  7, 75,
        6, 33, 92, 44, 23, 11, 61, 40,  5, 91, 34, 58, 48, 75, 10, 39, 77,
       70, 84, 95, 46, 81, 27,  6, 83,  9, 79, 39, 90, 77, 94, 29])
```

Now, create the histogram of these samples by passing as an argument the `hist()` function. For example, you want to divide the occurrences in 20 bins (if not specified, the default value is 10 bins) and to do that you have to use the kwarg `bin` (as shown in Figure 7-33).

```
In [ ]: n,bins,patches = plt.hist(pop,bins=20)
```

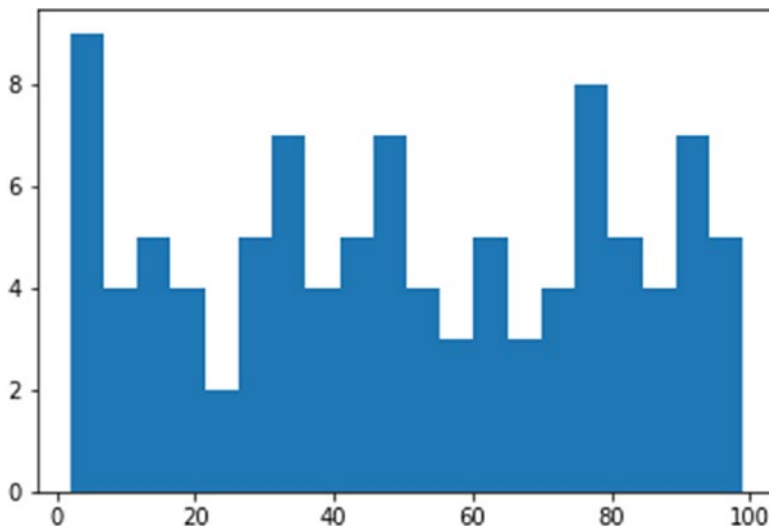


Figure 7-33. The histogram shows the occurrences in each bin

Bar Charts

Another very common type of chart is the bar chart. It is very similar to a histogram but in this case the x-axis is not used to reference numerical values but categories. The realization of the bar chart is very simple with matplotlib, using the `bar()` function.

```
In [ ]: import matplotlib.pyplot as plt
...: index = [0,1,2,3,4]
...: values = [5,7,3,4,6]
...: plt.bar(index,values)
Out[15]: <Container object of 5 artists>
```

With this few rows of code, you will obtain a bar chart as shown in [Figure 7-34](#).

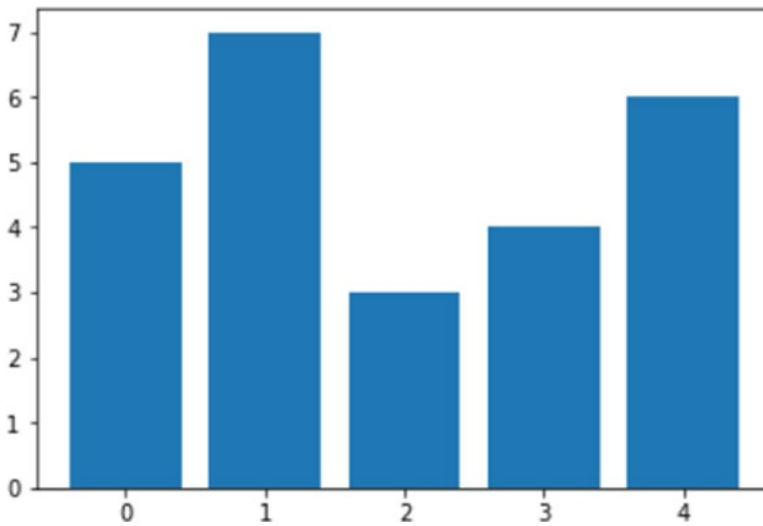


Figure 7-34. *The simplest bar chart with matplotlib*

If you look at Figure 7-34 you can see that the indices are drawn on the x-axis at the beginning of each bar. Actually, because each bar corresponds to a category, it would be better if you specify the categories through the tick label, defined by a list of strings passed to the `xticks()` function. As for the location of these tick labels, you have to pass a list containing the values corresponding to their positions on the x-axis as the first argument of the `xticks()` function. At the end you will get a bar chart, as shown in Figure 7-35.

```
In [ ]: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: plt.bar(index,values1)
...: plt.xticks(index+0.4,['A','B','C','D','E'])
```

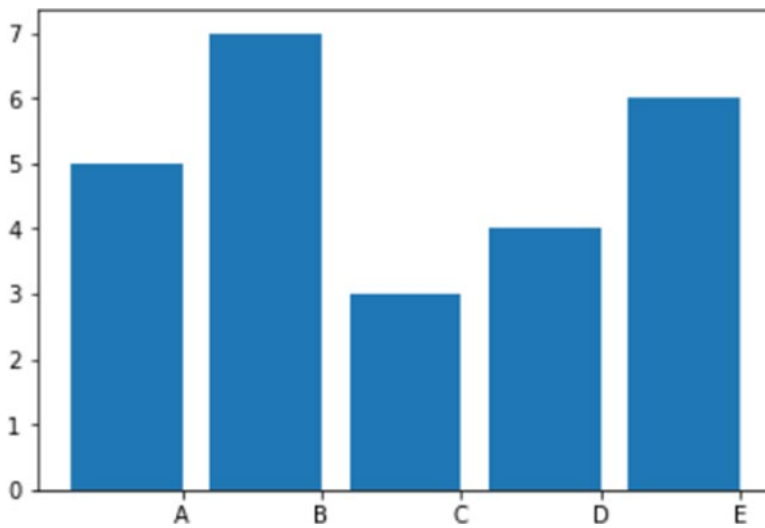


Figure 7-35. A simple bar chart with categories on the x-axis

Actually there are many other steps you can take to further refine the bar chart. Each of these finishes is set by adding a specific kwarg as an argument in the `bar()` function. For example, you can add the standard deviation values of the bar through the `yerr` kwarg along with a list containing the standard deviations. This kwarg is usually combined with another kwarg called `error_kw`, which, in turn, accepts other kwargs specialized for representing error bars. Two very specific kwargs used in this case are `eColor`, which specifies the color of the error bars, and `capsize`, which defines the width of the transverse lines that mark the ends of the error bars.

Another kwarg that you can use is `alpha`, which indicates the degree of transparency of the colored bar. Alpha is a value ranging from 0 to 1. When this value is 0 the object is completely transparent to become gradually more significant with the increase of the value, until arriving at 1, at which the color is fully represented.

As usual, the use of a legend is recommended, so in this case you should use a kwarg called `label` to identify the series that you are representing.

At the end you will get a bar chart with error bars, as shown in Figure 7-36.

```
In [ ]: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: std1 = [0.8,1,0.4,0.9,1.3]
...: plt.title('A Bar Chart')
```



```

...: plt.bar(index,values1,yerr=std1,error_kw={'ecolor':'0.1',
        'capsize':6},alpha=0.7,label='First')
...: plt.xticks(index+0.4,['A','B','C','D','E'])
...: plt.legend(loc=2)

```

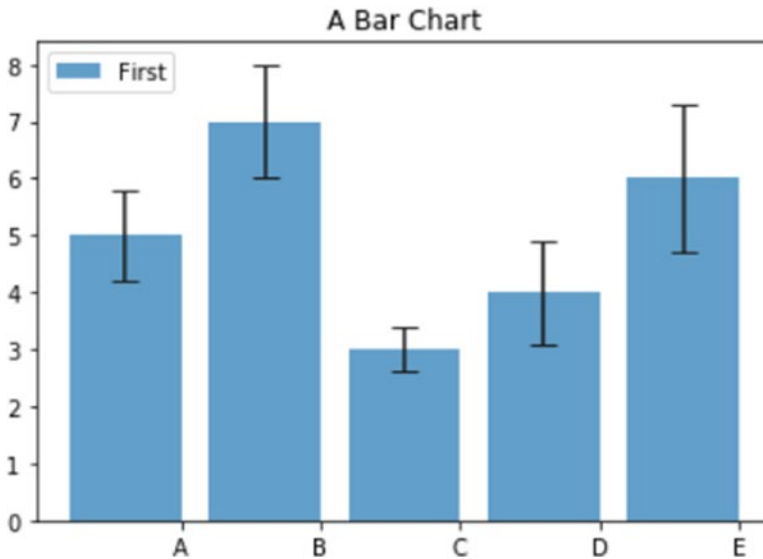


Figure 7-36. A bar chart with error bars

Horizontal Bar Charts

So far you have seen the bar chart oriented vertically. There are also bar charts oriented horizontally. This mode is implemented by a special function called `barh()`. The arguments and the kwargs valid for the `bar()` function remain the same for this function. The only change that you have to take into account is that the roles of the axes are reversed. Now, the categories are represented on the y-axis and the numerical values are shown on the x-axis (see Figure 7-37).

```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: std1 = [0.8,1,0.4,0.9,1.3]
...: plt.title('A Horizontal Bar Chart')

```

```

...: plt.barh(index, values1, xerr=std1, error_kw={'ecolor': '0.1',
...:      'capsize': 6}, alpha=0.7, label='First')
...: plt.yticks(index+0.4, ['A', 'B', 'C', 'D', 'E'])
...: plt.legend(loc=5)

```

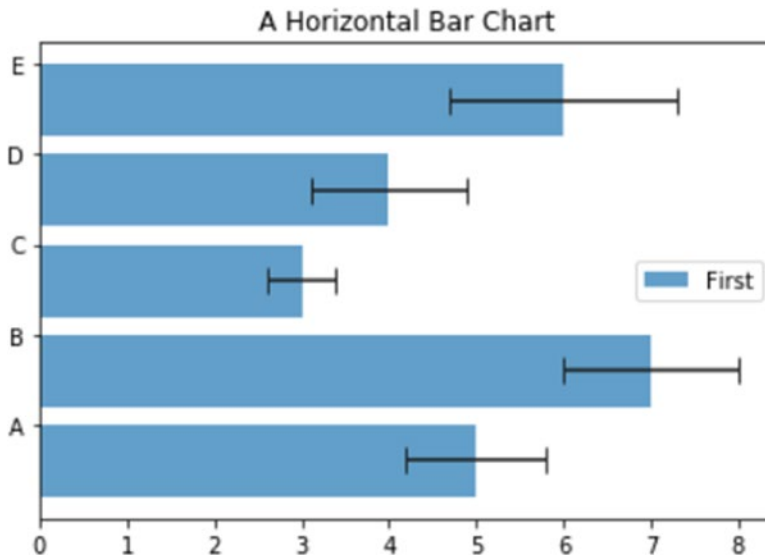


Figure 7-37. A simple horizontal bar chart

Multiserial Bar Charts

As line charts, bar charts also generally are used to simultaneously display larger series of values. But in this case it is necessary to make some clarifications on how to structure a multiseries bar chart. So far you have defined a sequence of indexes, each corresponding to a bar, to be assigned to the x-axis. These indices should represent categories. In this case, however, you have more bars that must share the same category.

One approach used to overcome this problem is to divide the space occupied by an index (for convenience its width is 1) in as many parts as are the bars sharing that index and that we want to display. Moreover, it is advisable to add space, which will serve as a gap to separate a category with respect to the next (as shown in Figure 7-38).

```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: values2 = [6,6,4,5,7]
...: values3 = [5,6,5,4,6]
...: bw = 0.3
...: plt.axis([0,5,0,8])
...: plt.title('A Multiseries Bar Chart',fontsize=20)
...: plt.bar(index,values1,bw,color='b')
...: plt.bar(index+bw,values2,bw,color='g')
...: plt.bar(index+2*bw,values3,bw,color='r')
...: plt.xticks(index+1.5*bw,['A','B','C','D','E'])

```

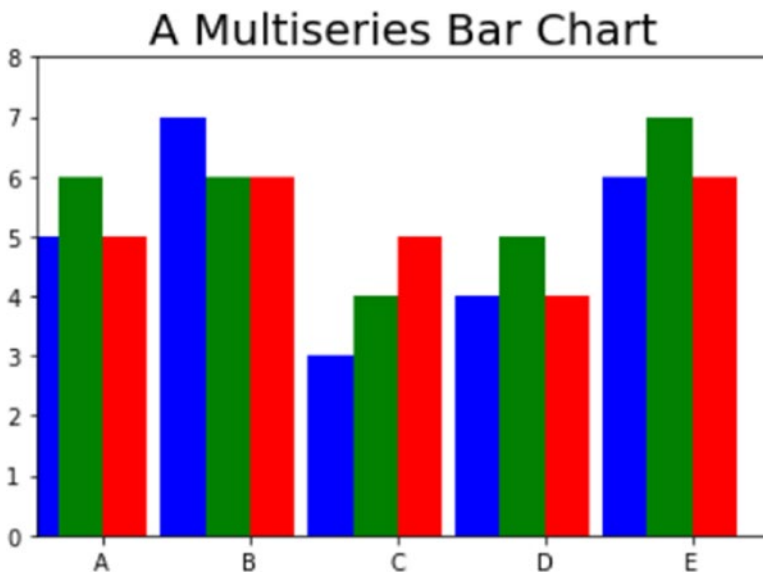


Figure 7-38. A multiseries bar chart displaying three series

Regarding the multiseried horizontal bar chart (see Figure 7-39), things are very similar. You have to replace the `bar()` function with the corresponding `barh()` function and remember to replace the `xticks()` function with the `yticks()` function. You need to reverse the range of values covered by the axes in the `axis()` function.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(5)
...: values1 = [5,7,3,4,6]
...: values2 = [6,6,4,5,7]
...: values3 = [5,6,5,4,6]
...: bw = 0.3
...: plt.axis([0,8,0,5])
...: plt.title('A Multiseried Horizontal Bar Chart',fontsize=20)
...: plt.barh(index,values1,bw,color='b')
...: plt.barh(index+bw,values2,bw,color='g')
...: plt.barh(index+2*bw,values3,bw,color='r')
...: plt.yticks(index+0.4,['A','B','C','D','E'])
```

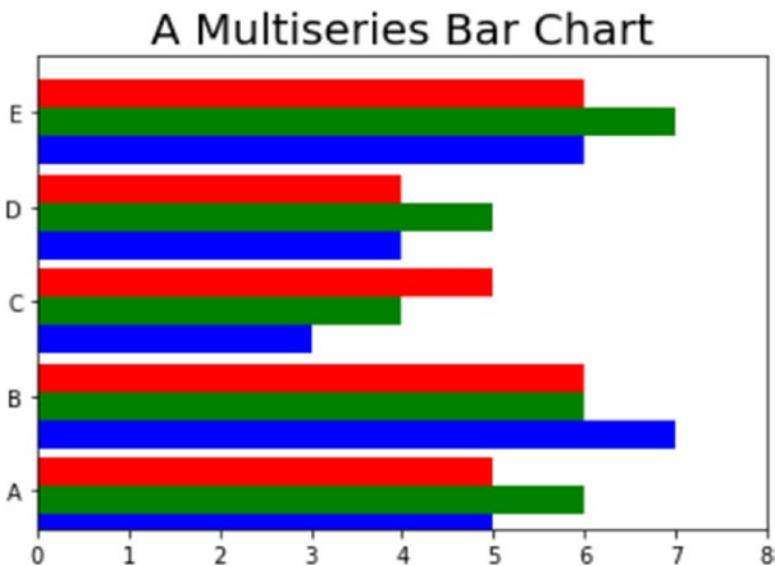


Figure 7-39. A multiseried horizontal bar chart

Multiseries Bar Charts with pandas Dataframe

As you saw in the line charts, the matplotlib library also provides the ability to directly represent the dataframe objects containing the results of data analysis in the form of bar charts. And even here it does it quickly, directly, and automatically. The only thing you need to do is use the `plot()` function applied to the dataframe object and specify inside a kwarg called `kind` to which you have to assign the type of chart you want to represent, which in this case is `bar`. Thus, without specifying any other settings, you will get the bar chart shown in Figure 7-40.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: import pandas as pd
...: data = {'series1':[1,3,4,3,5],
...:         'series2':[2,4,5,2,4],
...:         'series3':[3,2,3,1,3]}
...: df = pd.DataFrame(data)
...: df.plot(kind='bar')
```

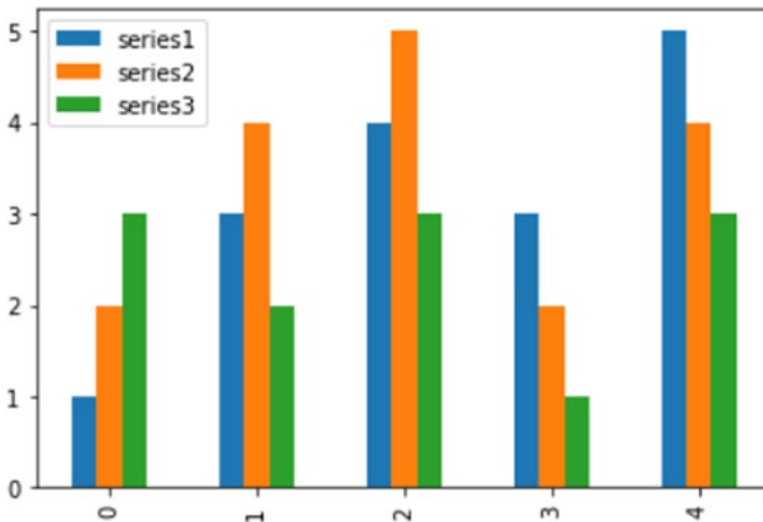


Figure 7-40. The values in a dataframe can be directly displayed as a bar chart

However, if you want to get more control, or if your case requires it, you can still extract portions of the dataframe as NumPy arrays and use them as illustrated in the previous examples in this section. That is, by passing them separately as arguments to the matplotlib functions.

Moreover, regarding the horizontal bar chart, the same rules can be applied, but remember to set `barh` as the value of the `kind` kwarg. You'll get a multiseries horizontal bar chart, as shown in Figure 7-41.

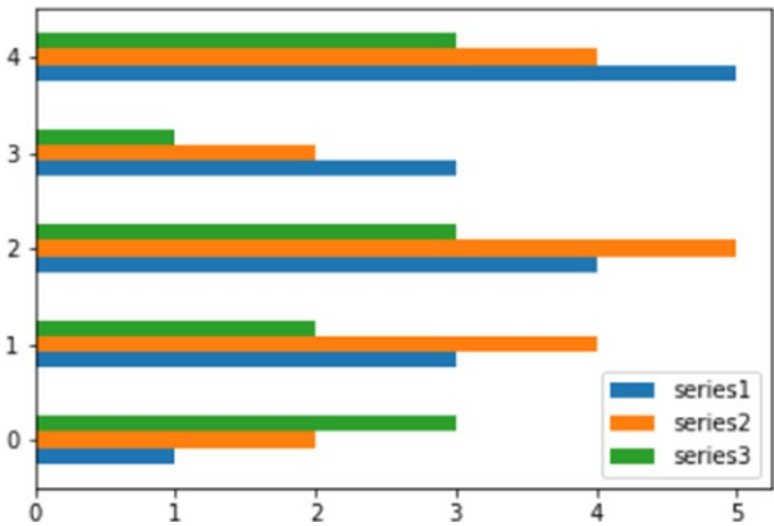


Figure 7-41. A horizontal bar chart could be a valid alternative to visualize your dataframe values

Multiseries Stacked Bar Charts

Another form to represent a multiseries bar chart is in the stacked form, in which the bars are stacked one on the other. This is especially useful when you want to show the total value obtained by the sum of all the bars.

To transform a simple multiseries bar chart in a stacked one, you add the `bottom` kwarg to each `bar()` function. Each series must be assigned to the corresponding `bottom` kwarg. At the end you will obtain the stacked bar chart, as shown in Figure 7-42.

```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: series1 = np.array([3,4,5,3])
...: series2 = np.array([1,2,2,5])
...: series3 = np.array([2,3,3,4])
...: index = np.arange(4)
...: plt.axis([-0.5,3.5,0,15])
...: plt.title('A Multiseries Stacked Bar Chart')
...: plt.bar(index,series1,color='r')
...: plt.bar(index,series2,color='b',bottom=series1)
...: plt.bar(index,series3,color='g',bottom=(series2+series1))
...: plt.xticks(index+0.4,['Jan18','Feb18','Mar18','Apr18'])

```

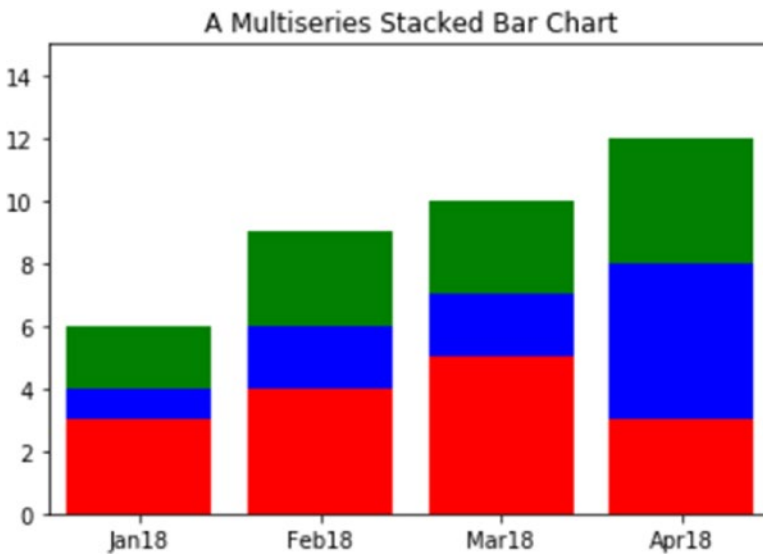


Figure 7-42. *A multiseries stacked bar*

Here too, in order to create the equivalent horizontal stacked bar chart, you need to replace the `bar()` function with `barh()` function, being careful to change the other parameters as well. Indeed the `xticks()` function should be replaced with the `yticks()` function because the labels of the categories must now be reported on the y-axis. After making all these changes, you will obtain the horizontal stacked bar chart as shown in Figure 7-43.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(4)
...: series1 = np.array([3,4,5,3])
...: series2 = np.array([1,2,2,5])
...: series3 = np.array([2,3,3,4])
...: plt.axis([0,15,-0.5,3.5])
...: plt.title('A Multiseries Horizontal Stacked Bar Chart')
...: plt.barh(index,series1,color='r')
...: plt.barh(index,series2,color='g',left=series1)
...: plt.barh(index,series3,color='b',left=(series1+series2))
...: plt.yticks(index+0.4,['Jan18','Feb18','Mar18','Apr18'])
```



Figure 7-43. A multiseries horizontal stacked bar chart

So far the various series have been distinguished by using different colors. Another mode of distinction between the various series is to use hatches that allow you to fill the various bars with strokes drawn in a different way. To do this, you have first to set the color of the bar as white and then you have to use the hatch kwarg to define how the hatch is to be set. The various hatches have codes distinguishable among these characters (|, /, -, \, *, -) corresponding to the line style filling the bar. The more a symbol is replicated, the denser the lines forming the hatch will be. For example, /// is more dense than //, which is more dense than / (see Figure 7-44).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: index = np.arange(4)
...: series1 = np.array([3,4,5,3])
...: series2 = np.array([1,2,2,5])
...: series3 = np.array([2,3,3,4])
...: plt.axis([0,15,-0.5,3.5])
...: plt.title('A Multiseries Horizontal Stacked Bar Chart')
...: plt.barh(index,series1,color='w',hatch='xx')
...: plt.barh(index,series2,color='w',hatch='///', left=series1)
...: plt.barh(index,series3,color='w',hatch='\\\\\\\\',left=(series1+series2))
...: plt.yticks(index+0.4,['Jan18','Feb18','Mar18','Apr18'])
```

```
Out[453]:
([<matplotlib.axis.YTick at 0x2a9f0748>,
 <matplotlib.axis.YTick at 0x2a9e1f98>,
 <matplotlib.axis.YTick at 0x2ac06518>,
 <matplotlib.axis.YTick at 0x2ac52128>],
 <a list of 4 Text yticklabel objects>)
```

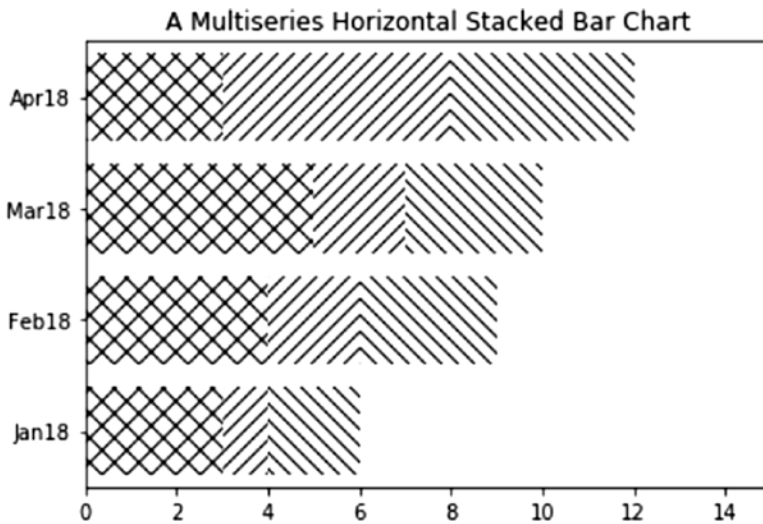


Figure 7-44. The stacked bars can be distinguished by their hatches

Stacked Bar Charts with a pandas Dataframe

Also with regard to stacked bar charts, it is very simple to directly represent the values contained in the dataframe object by using the `plot()` function. You need only to add as an argument the `stacked` kwarg set to `True` (see Figure 7-45).

```
In [ ]: import matplotlib.pyplot as plt
...: import pandas as pd
...: data = {'series1':[1,3,4,3,5],
...:         'series2':[2,4,5,2,4],
...:         'series3':[3,2,3,1,3]}
...: df = pd.DataFrame(data)
...: df.plot(kind='bar', stacked=True)
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0xcda8f98>
```

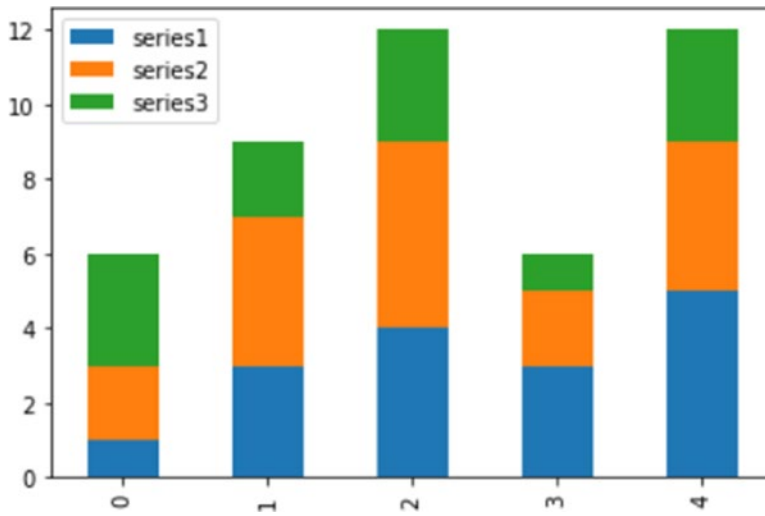


Figure 7-45. The values of a dataframe can be directly displayed as a stacked bar chart

Other Bar Chart Representations

Another type of very useful representation is that of a bar chart for comparison, where two series of values sharing the same categories are compared by placing the bars in opposite directions along the y-axis. In order to do this, you have to put the y values of one of the two series in a negative form. Also in this example, you will see the possibility of coloring the inner color of the bars in a different way. In fact, you can do this by setting the two different colors on a specific kwarg: `facecolor`.

Furthermore, in this example, you will see how to add the y value with a label at the end of each bar. This could be useful to increase the readability of the bar chart. You can do this using a for loop in which the `text()` function will show the y value. You can adjust the label position with the two kwargs called `ha` and `va`, which control the horizontal and vertical alignment, respectively. The result will be the chart shown in Figure 7-46.

```
In [ ]: import matplotlib.pyplot as plt
...: x0 = np.arange(8)
...: y1 = np.array([1,3,4,6,4,3,2,1])
...: y2 = np.array([1,2,5,4,3,3,2,1])
...: plt.ylim(-7,7)
```

```

...: plt.bar(x0,y1,0.9,facecolor='r')
...: plt.bar(x0,-y2,0.9,facecolor='b')
...: plt.xticks(())
...: plt.grid(True)
...: for x, y in zip(x0, y1):
...:     plt.text(x + 0.4, y + 0.05, '%d' % y, ha='center', va= 'bottom')
...:
...: for x, y in zip(x0, y2):
...:     plt.text(x + 0.4, -y - 0.05, '%d' % y, ha='center', va= 'top')

```

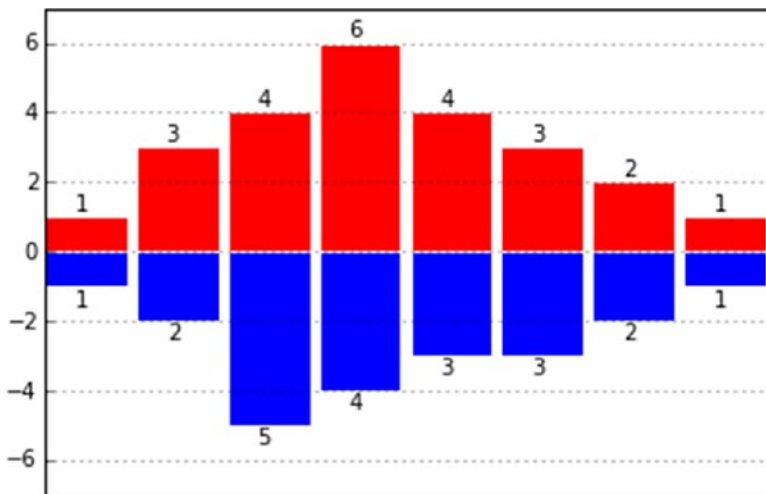


Figure 7-46. Two series can be compared using this kind of bar chart

Pie Charts

An alternative way to display data to the bar charts is the pie chart, easily obtainable using the `pie()` function.

Even for this type of function, you pass as the main argument a list containing the values to be displayed. I chose the percentages (their sum is 100), but you can use any kind of value. It will be up to the `pie()` function to inherently calculate the percentage occupied by each value.

Also with this type of representation, you need to define some key features making use of the kwargs. For example, if you want to define the sequence of the colors, which will be assigned to the sequence of input values correspondingly, you have to use the `colors` kwarg. Therefore, you have to assign a list of strings, each containing the name of the desired color. Another important feature is to add labels to each slice of the pie. To do this, you have to use the `labels` kwarg to which you will assign a list of strings containing the labels to be displayed in sequence.

In addition, in order to draw the pie chart in a perfectly spherical way, you have to add the `axis()` function to the end, specifying the string `'equal'` as an argument. You will get a pie chart as shown in Figure 7-47.

```
In [ ]: import matplotlib.pyplot as plt
...: labels = ['Nokia', 'Samsung', 'Apple', 'Lumia']
...: values = [10, 30, 45, 15]
...: colors = ['yellow', 'green', 'red', 'blue']
...: plt.pie(values, labels=labels, colors=colors)
...: plt.axis('equal')
```

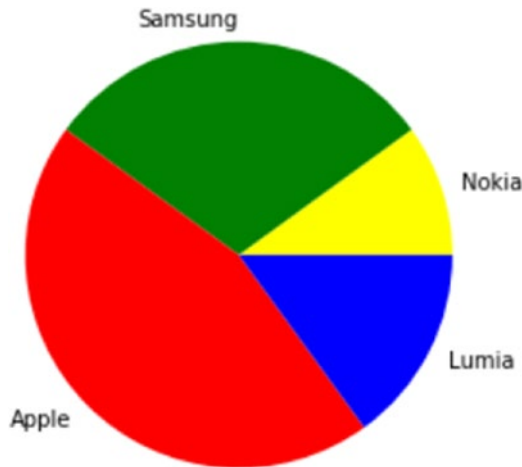


Figure 7-47. A very simple pie chart

To add complexity to the pie chart, you can draw it with a slice extracted from the pie. This is usually done when you want to focus on a specific slice. In this case, for example, you would highlight the slice referring to Nokia. In order to do this, there is a special kwarg named `explode`. It is nothing but a sequence of float values of 0 or 1, where 1 corresponds to the fully extended slice and 0 corresponds to slices completely in the pie. All intermediate values correspond to an intermediate degree of extraction (see Figure 7-48).

You can also add a title to the pie chart with the `title()` function. You can also adjust the angle of rotation of the pie by adding the `startangle` kwarg, which takes an integer value between 0 and 360, which are the degrees of rotation precisely (0 is the default value).

The modified chart should appear as shown in Figure 7-48.

```
In [ ]: import matplotlib.pyplot as plt
...: labels = ['Nokia','Samsung','Apple','Lumia']
...: values = [10,30,45,15]
...: colors = ['yellow','green','red','blue']
...: explode = [0.3,0,0,0]
...: plt.title('A Pie Chart')
...: plt.pie(values,labels=labels,colors=colors,explode=explode,
startangle=180)
...: plt.axis('equal')
```

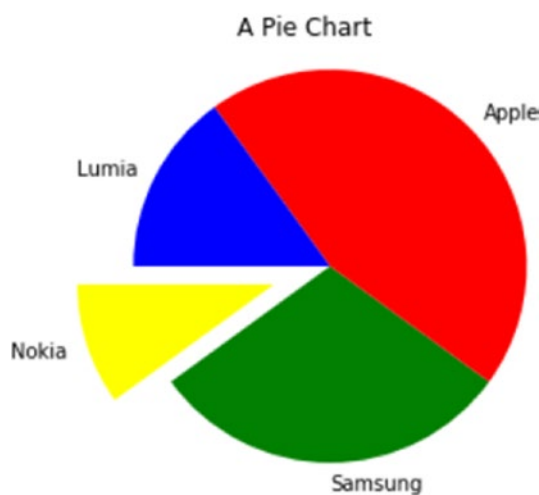


Figure 7-48. A more advanced pie chart

But the possible additions that you can insert in a pie chart do not end here. For example, a pie chart does not have axes with ticks and so it is difficult to imagine the perfect percentage represented by each slice. To overcome this, you can use the `autopct` kwarg, which adds to the center of each slice a text label showing the corresponding value.

If you want to make it an even more appealing image, you can add a shadow with the `shadow` kwarg setting it to `True`. In the end you will get a pie chart as shown in Figure 7-49.

```
In [ ]: import matplotlib.pyplot as plt
...: labels = ['Nokia', 'Samsung', 'Apple', 'Lumia']
...: values = [10, 30, 45, 15]
...: colors = ['yellow', 'green', 'red', 'blue']
...: explode = [0.3, 0, 0, 0]
...: plt.title('A Pie Chart')
...: plt.pie(values, labels=labels, colors=colors, explode=explode,
...:         shadow=True, autopct='%1.1f%%', startangle=180)
...: plt.axis('equal')
```

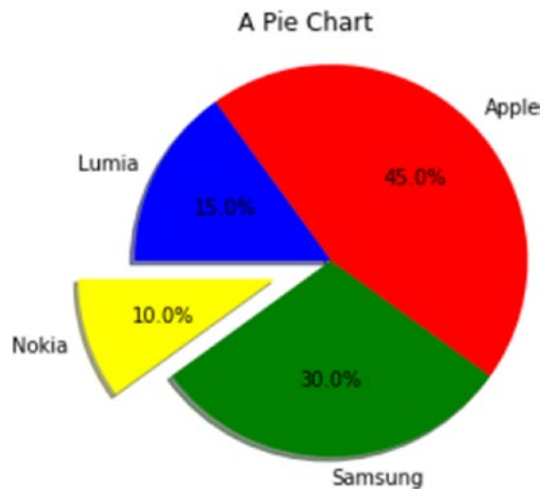


Figure 7-49. An even more advanced pie chart

Pie Charts with a pandas Dataframe

Even for the pie chart, you can represent the values contained within a dataframe object. In this case, however, the pie chart can represent only one series at a time, so in this example you will display only the values of the first series specifying `df['series1']`. You have to specify the type of chart you want to represent through the `kind` kwarg in the `plot()` function, which in this case is `pie`. Furthermore, because you want to represent a pie chart as perfectly circular, it is necessary that you add the `figsize` kwarg. At the end you will obtain a pie chart as shown in Figure 7-50.

```
In [ ]: import matplotlib.pyplot as plt
...: import pandas as pd
...: data = {'series1':[1,3,4,3,5],
...:         'series2':[2,4,5,2,4],
...:         'series3':[3,2,3,1,3]}
...: df = pd.DataFrame(data)
...: df['series1'].plot(kind='pie',figsize=(6,6))
Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0xe1ba710>
```

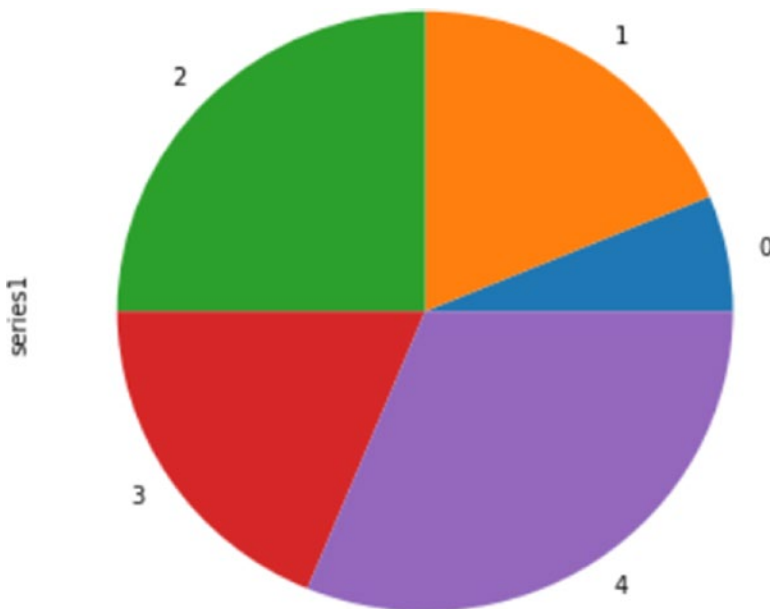


Figure 7-50. The values in a pandas dataframe can be directly drawn as a pie chart

Advanced Charts

In addition to the more classical charts such as bar charts or pie charts, you might want to represent your results in an alternative ways. On the Internet and in various publications there are many examples in which many alternative graphics solutions are discussed and proposed, some really brilliant and captivating. This section only shows some graphic representations; a more detailed discussion about this topic is beyond the purpose of this book. You can use this section as an introduction to a world that is constantly expanding: data visualization.

Contour Plots

A quite common type of chart in the scientific world is the *contour plot* or *contour map*. This visualization is in fact suitable for displaying three-dimensional surfaces through a contour map composed of curves closed showing the points on the surface that are located at the same level, or that have the same z value.

Although visually the contour plot is a very complex structure, its implementation is not so difficult, thanks to the matplotlib library. First, you need the function $z = f(x, y)$ for generating a three-dimensional surface. Then, once you have defined a range of values x, y that will define the area of the map to be displayed, you can calculate the z values for each pair (x, y) , applying the function $f(x, y)$ just defined in order to obtain a matrix of z values. Finally, thanks to the `contour()` function, you can generate the contour map of the surface. It is often desirable to add also a color map along with a contour map. That is, the areas delimited by the curves of level are filled by a color gradient, defined by a color map. For example, as in Figure 7-51, you may indicate negative values with increasingly dark shades of blue, and move to yellow and then red with the increase of positive values.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: dx = 0.01; dy = 0.01
...: x = np.arange(-2.0, 2.0, dx)
...: y = np.arange(-2.0, 2.0, dy)
...: X, Y = np.meshgrid(x, y)
...: def f(x, y):
...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
```

```

...: C = plt.contour(X,Y,f(X,Y),8,colors='black')
...: plt.contourf(X,Y,f(X,Y),8)
...: plt.clabel(C, inline=1, fontsize=10)

```

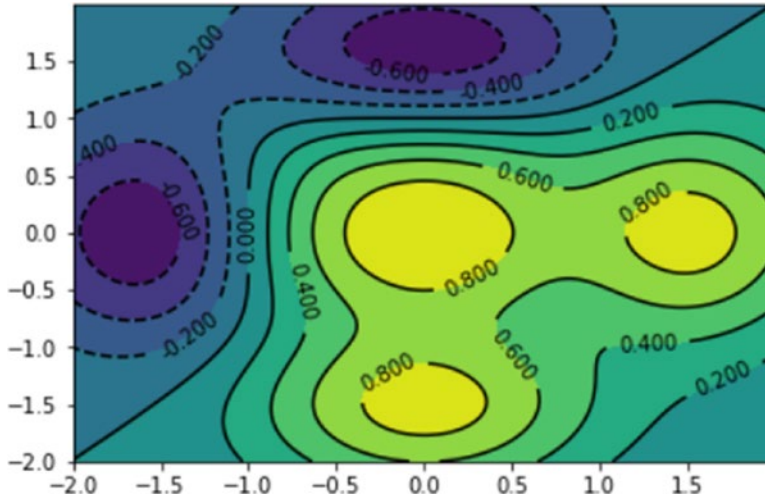


Figure 7-51. A contour map can describe the z values of a surface

The standard color gradient (color map) is represented in Figure 7-51. Actually you choose among a large number of color maps available just specifying them with the `cmap` kwarg.

Furthermore, when you have to deal with this kind of representation, adding a color scale as a reference to the side of the graph is almost a must. This is possible by simply adding the `colorbar()` function at the end of the code. In Figure 7-52 you can see another example of color map that starts from black, passes through red, then turns yellow until reaching white for the highest values. This color map is `plt.cm.hot`.

```

In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: dx = 0.01; dy = 0.01
...: x = np.arange(-2.0,2.0,dx)
...: y = np.arange(-2.0,2.0,dy)
...: X,Y = np.meshgrid(x,y)
...: def f(x,y):
...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)

```

```

...: C = plt.contour(X,Y,f(X,Y),8,colors='black')
...: plt.contourf(X,Y,f(X,Y),8,cmap=plt.cm.hot)
...: plt.clabel(C, inline=1, fontsize=10)
...: plt.colorbar()

```

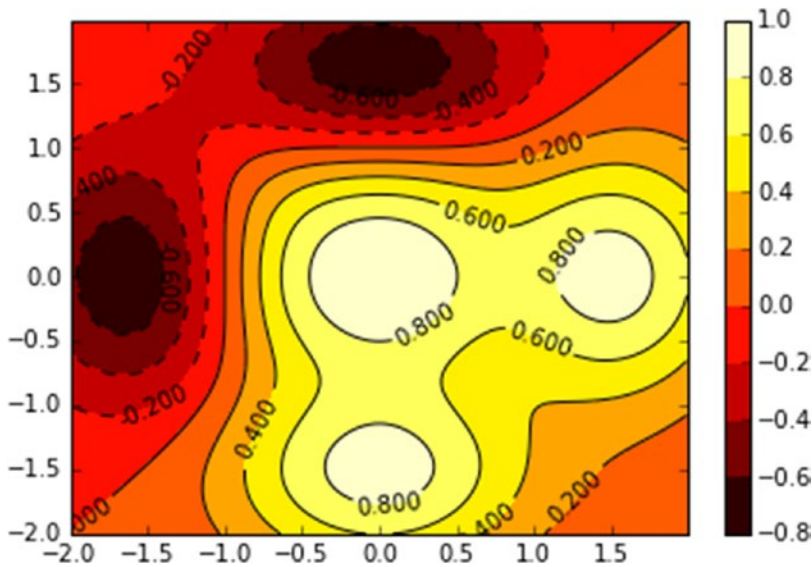


Figure 7-52. The “hot” color map gradient gives an attractive look to the contour map

Polar Charts

Another type of advanced chart that is popular is the *polar chart*. This type of chart is characterized by a series of sectors that extend radially; each of these areas will occupy a certain angle. Thus you can display two different values assigning them to the magnitudes that characterize the polar chart: the extension of the radius r and the angle θ occupied by the sector. These in fact are the polar coordinates (r, θ) , an alternative way of representing functions at the coordinate axes. From the graphical point of view, you could imagine it as a kind of chart that has characteristics both of the pie chart and of the bar chart. In fact as the pie chart, the angle of each sector gives percentage information represented by that category with respect to the total. As for the bar chart, the radial extension is the numerical value of that category.

So far we have used the standard set of colors using single characters as the color code (e.g., r to indicate red). In fact you can use any sequence of colors you want. You have to define a list of string values that contain RGB codes in the `#rrggbb` format corresponding to the colors you want.

Oddly, to get a polar chart you have to use the `bar()` function and pass the list containing the angles θ and a list of the radial extension of each sector. The result will be a polar chart, as shown in Figure 7-53.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: N = 8
...: theta = np.arange(0., 2 * np.pi, 2 * np.pi / N)
...: radii = np.array([4, 7, 5, 3, 1, 5, 6, 7])
...: plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
...: colors = np.array(['#4bb2c5', '#c5b47f', '#EAA228', '#579575',
...:                    '#839557', '#958c12', '#953579', '#4b5de4'])
...: bars = plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0,
...:                 color=colors)
```

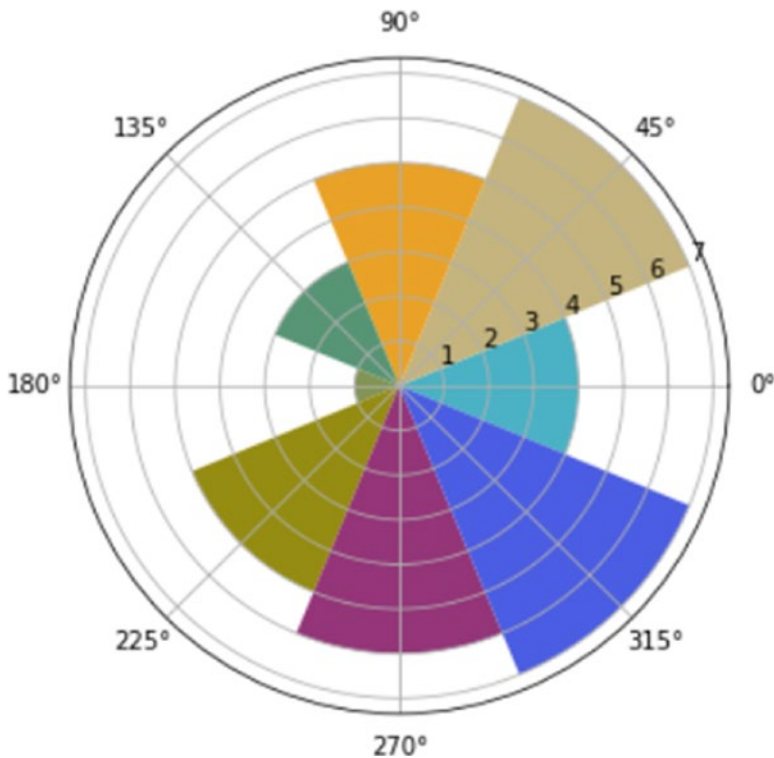


Figure 7-53. A polar chart

In this example, you have defined the sequence of colors using the format `#rrggbb`, but you can also specify a sequence of colors as strings with their actual name (see Figure 7-54).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: N = 8
...: theta = np.arange(0., 2 * np.pi, 2 * np.pi / N)
...: radii = np.array([4, 7, 5, 3, 1, 5, 6, 7])
...: plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
...: colors = np.array(['lightgreen', 'darkred', 'navy', 'brown',
...:                   'violet', 'plum', 'yellow', 'darkgreen'])
...: bars = plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0,
...:               color=colors)
```

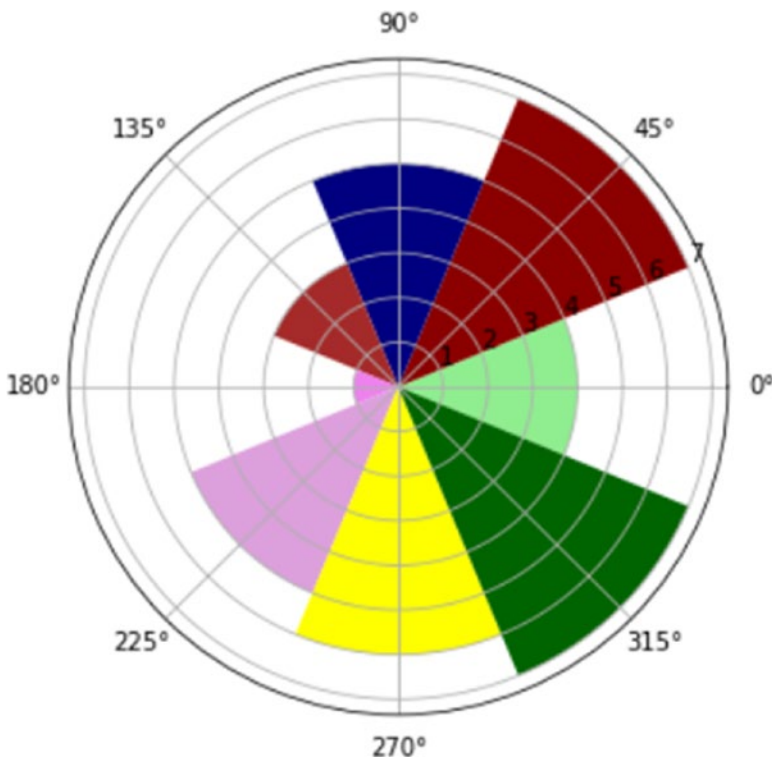


Figure 7-54. A polar chart with another sequence of colors

The mplot3d Toolkit

The `mplot3d` toolkit is included with all standard installations of `matplotlib` and allows you to extend the capabilities of visualization to 3D data. If the figure is displayed in a separate window, you can rotate the axes of the three-dimensional representation with the mouse.

With this package you are still using the `Figure` object, only that instead of the `Axes` object you will define a new kind of object, called `Axes3D`, and introduced by this toolkit. Thus, you need to add a new import to the code, if you want to use the `Axes3D` object.

```
from mpl_toolkits.mplot3d import Axes3D
```

3D Surfaces

In a previous section, you used the contour plot to represent the three-dimensional surfaces through the level lines. Using the `mplot3D` package, surfaces can be drawn directly in 3D. In this example, you will use the same function $z = f(x, y)$ you have used in the contour map.

Once you have calculated the `meshgrid`, you can view the surface with the `plot_surface()` function. A three-dimensional blue surface will appear, as in [Figure 7-55](#).

```
In [ ]: from mpl_toolkits.mplot3d import Axes3D
...: import matplotlib.pyplot as plt
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: X = np.arange(-2,2,0.1)
...: Y = np.arange(-2,2,0.1)
...: X,Y = np.meshgrid(X,Y)
...: def f(x,y):
...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
...: ax.plot_surface(X,Y,f(X,Y), rstride=1, cstride=1)
```

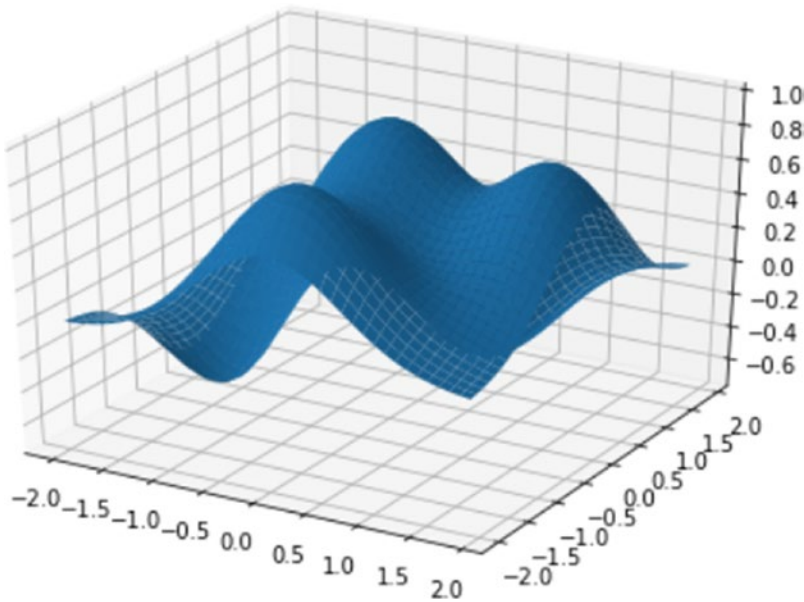


Figure 7-55. A 3D surface can be represented with the *mplot3d* toolkit

A 3D surface stands out most by changing the color map, for example by setting the `cmap` kwarg. You can also rotate the surface using the `view_init()` function. In fact, this function adjusts the view point from which you see the surface, changing the two kwargs called `elev` and `azim`. Through their combination you can get the surface displayed from any angle. The first kwarg adjusts the height at which the surface is seen, while `azim` adjusts the angle of rotation of the surface.

For instance, you can change the color map using `plt.cm.hot` and moving the view point to `elev=30` and `azim=125`. The result is shown in Figure 7-56.

```
In [ ]: from mpl_toolkits.mplot3d import Axes3D
...: import matplotlib.pyplot as plt
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: X = np.arange(-2,2,0.1)
...: Y = np.arange(-2,2,0.1)
...: X,Y = np.meshgrid(X,Y)
...: def f(x,y):
...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
...: ax.plot_surface(X,Y,f(X,Y), rstride=1, cstride=1, cmap=plt.cm.hot)
...: ax.view_init(elev=30,azim=125)
```

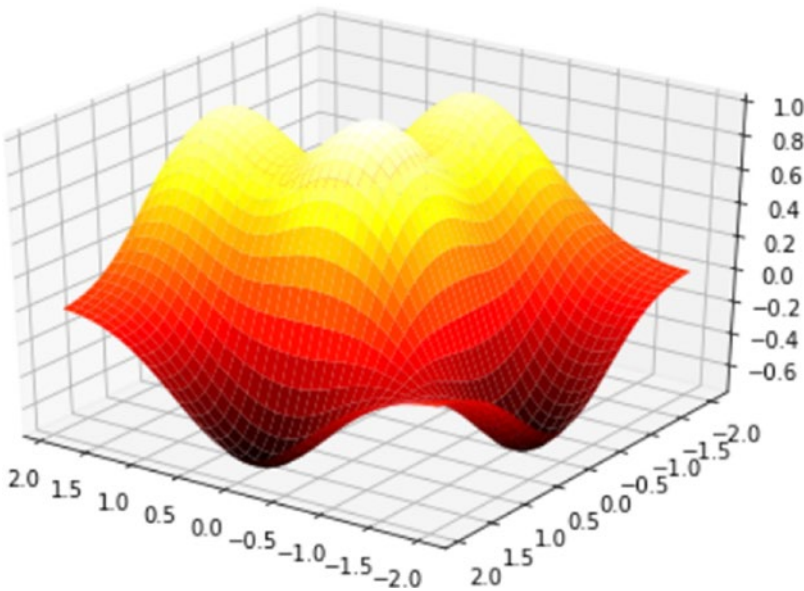



Figure 7-56. The 3D surface can be rotated and observed from a higher viewpoint

Scatter Plots in 3D

The mode most used among all 3D views remains the 3D scatter plot. With this type of visualization, you can identify if the points follow particular trends, but above all if they tend to cluster.

In this case, you will use the `scatter()` function as the 2D case but applied on the `Axes3D` object. By doing this, you can visualize different series, expressed by the calls to the `scatter()` function, all together in the same 3D representation (see Figure 7-57).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: from mpl_toolkits.mplot3d import Axes3D
...: xs = np.random.randint(30,40,100)
...: ys = np.random.randint(20,30,100)
...: zs = np.random.randint(10,20,100)
...: xs2 = np.random.randint(50,60,100)
...: ys2 = np.random.randint(30,40,100)
...: zs2 = np.random.randint(50,70,100)
...: xs3 = np.random.randint(10,30,100)
```



```

...: ys3 = np.random.randint(40,50,100)
...: zs3 = np.random.randint(40,50,100)
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: ax.scatter(xs,ys,zs)
...: ax.scatter(xs2,ys2,zs2,c='r',marker='^')
...: ax.scatter(xs3,ys3,zs3,c='g',marker='*')
...: ax.set_xlabel('X Label')
...: ax.set_ylabel('Y Label')
...: ax.set_zlabel('Z Label')
Out[34]: <matplotlib.text.Text at 0xe1c2438>

```

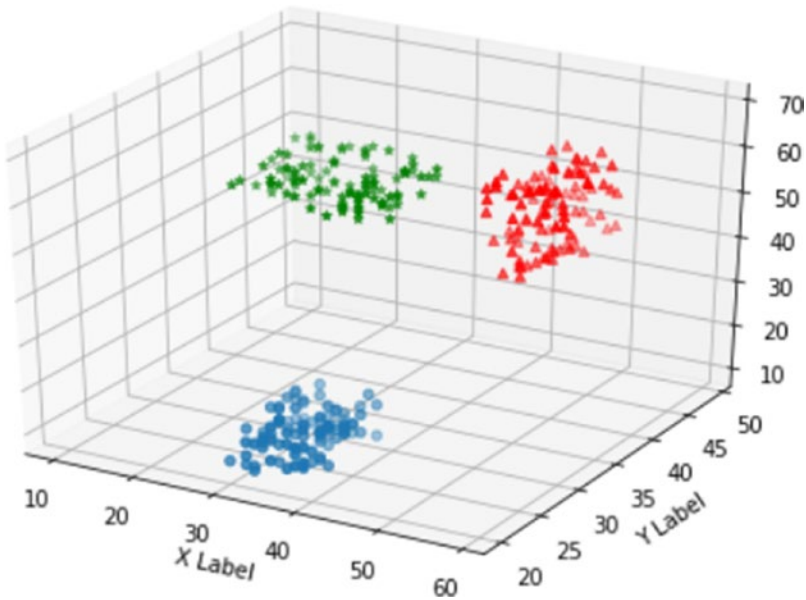


Figure 7-57. This 3D scatter plot shows three different clusters

Bar Charts in 3D

Another type of 3D plot widely used in data analysis is the 3D bar chart. Also in this case, you use the `bar()` function applied to the object `Axes3D`. If you define multiple series, you can accumulate several calls to the `bar()` function in the same 3D visualization (see Figure 7-58).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: from mpl_toolkits.mplot3d import Axes3D
...: x = np.arange(8)
...: y = np.random.randint(0,10,8)
...: y2 = y + np.random.randint(0,3,8)
...: y3 = y2 + np.random.randint(0,3,8)
...: y4 = y3 + np.random.randint(0,3,8)
...: y5 = y4 + np.random.randint(0,3,8)
...: clr = ['#4bb2c5', '#c5b47f', '#EAA228', '#579575', '#839557',
...:        '#958c12', '#953579', '#4b5de4']
...: fig = plt.figure()
...: ax = Axes3D(fig)
...: ax.bar(x,y,0,zdir='y',color=clr)
...: ax.bar(x,y2,10,zdir='y',color=clr)
...: ax.bar(x,y3,20,zdir='y',color=clr)
...: ax.bar(x,y4,30,zdir='y',color=clr)
...: ax.bar(x,y5,40,zdir='y',color=clr)
...: ax.set_xlabel('X Axis')
...: ax.set_ylabel('Y Axis')
...: ax.set_zlabel('Z Axis')
...: ax.view_init(elev=40)
```

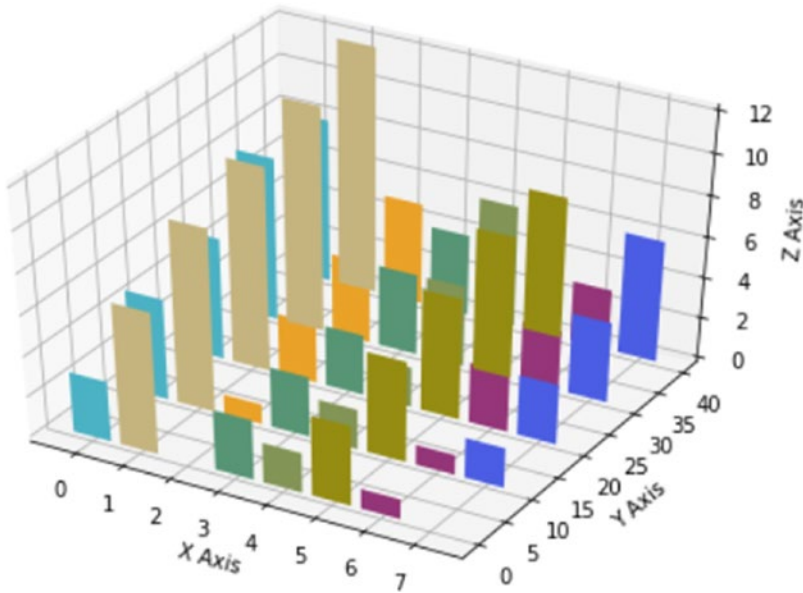


Figure 7-58. A 3D bar chart

Multi-Panel Plots

So far you’ve had the chance to see different ways of representing data through a chart. You saw how to see more charts in the same figure by separating them with subplots. In this section, you will deepen your understanding of this topic by analyzing more complex cases.

Display Subplots Within Other Subplots

Now an even more advanced method will be explained: the ability to view charts within others, enclosed within frames. Since we are talking of frames, i.e., Axes objects, you need to separate the main Axes (i.e., the general chart) from the frame you want to add that will be another instance of Axes. To do this, you use the `figures()` function to get the Figure object on which you will define two different Axes objects using the `add_axes()` function. See the result of this example in Figure 7-59.

```
In [ ]: import matplotlib.pyplot as plt
...: fig = plt.figure()
...: ax = fig.add_axes([0.1,0.1,0.8,0.8])
...: inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
```

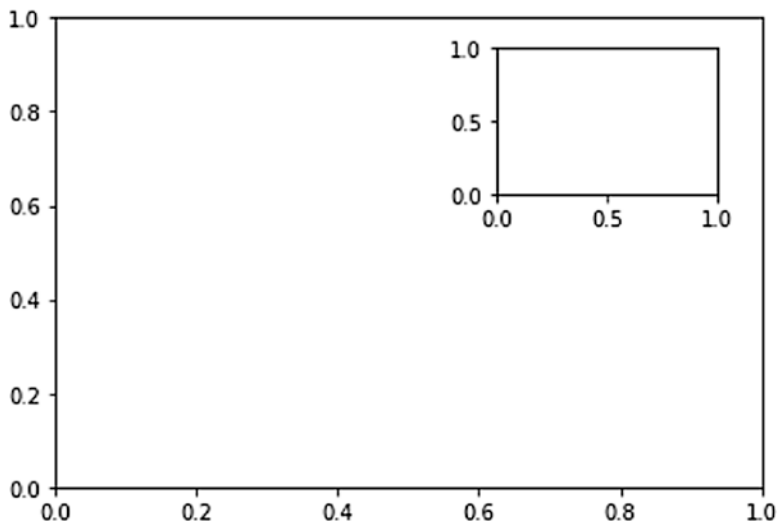


Figure 7-59. A subplot is displayed within another plot

To better understand the effect of this mode of display, you can fill the previous Axes with real data, as shown in Figure 7-60.

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: fig = plt.figure()
...: ax = fig.add_axes([0.1,0.1,0.8,0.8])
...: inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
...: x1 = np.arange(10)
...: y1 = np.array([1,2,7,1,5,2,4,2,3,1])
...: x2 = np.arange(10)
...: y2 = np.array([1,3,4,5,4,5,2,6,4,3])
...: ax.plot(x1,y1)
...: inner_ax.plot(x2,y2)
Out[95]: [<matplotlib.lines.Line2D at 0x14acf6d8>]
```

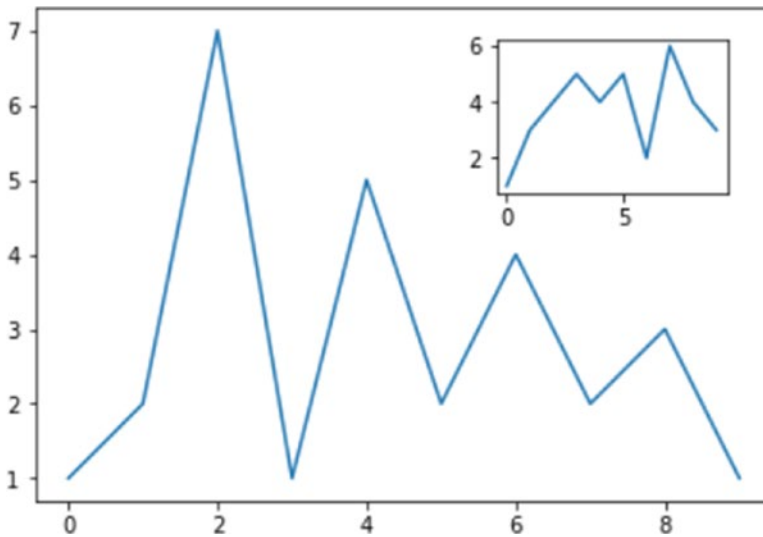


Figure 7-60. A more realistic visualization of a subplot within another plot

Grids of Subplots

You have already seen the creation of subplots. It is quite simple to add subplots using the `subplots()` function and by dividing a plot into sectors. `matplotlib` allows you to manage even more complex cases using another function called `GridSpec()`. This subdivision allows splitting the drawing area into a grid of sub-areas, to which you can assign one or more of them to each subplot, so that in the end you can obtain subplots with different sizes and orientations, as you can see in Figure 7-61.

```
In [ ]: import matplotlib.pyplot as plt
...: gs = plt.GridSpec(3,3)
...: fig = plt.figure(figsize=(6,6))
...: fig.add_subplot(gs[1,:2])
...: fig.add_subplot(gs[0,:2])
...: fig.add_subplot(gs[2,0])
...: fig.add_subplot(gs[:2,2])
...: fig.add_subplot(gs[2,1:])
Out[97]: <matplotlib.axes._subplots.AxesSubplot at 0x12717438>
```

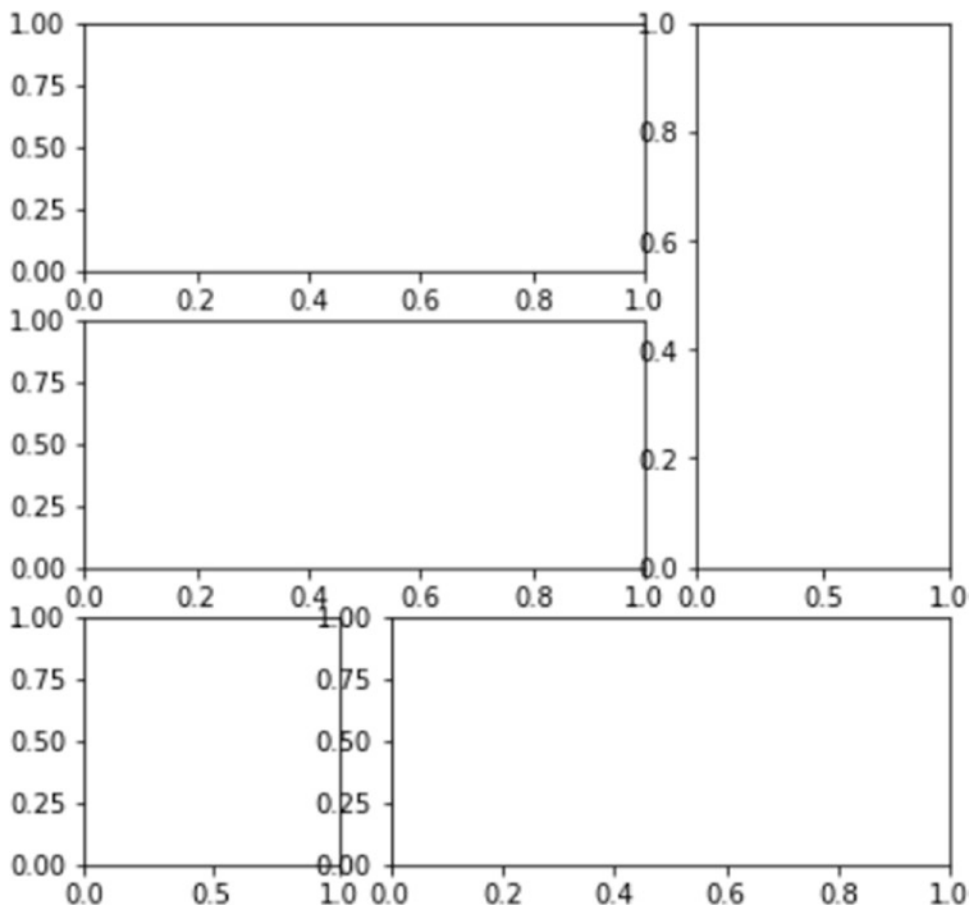


Figure 7-61. Subplots with different sizes can be defined on a grid of sub-areas

Now that it's clear to you how to manage the grid by assigning the various sectors to the subplot, it's time to see how to use these subplots. In fact, you can use the Axes object returned by each `add_subplot()` function to call the `plot()` function to draw the corresponding plot (see Figure 7-62).

```
In [ ]: import matplotlib.pyplot as plt
...: import numpy as np
...: gs = plt.GridSpec(3,3)
...: fig = plt.figure(figsize=(6,6))
...: x1 = np.array([1,3,2,5])
...: y1 = np.array([4,3,7,2])
...: x2 = np.arange(5)
```

```

...: y2 = np.array([3,2,4,6,4])
...: s1 = fig.add_subplot(gs[1,:2])
...: s1.plot(x,y,'r')
...: s2 = fig.add_subplot(gs[0,:2])
...: s2.bar(x2,y2)
...: s3 = fig.add_subplot(gs[2,0])
...: s3.barh(x2,y2,color='g')
...: s4 = fig.add_subplot(gs[:2,2])
...: s4.plot(x2,y2,'k')
...: s5 = fig.add_subplot(gs[2,1:])
...: s5.plot(x1,y1,'b^',x2,y2,'yo')

```

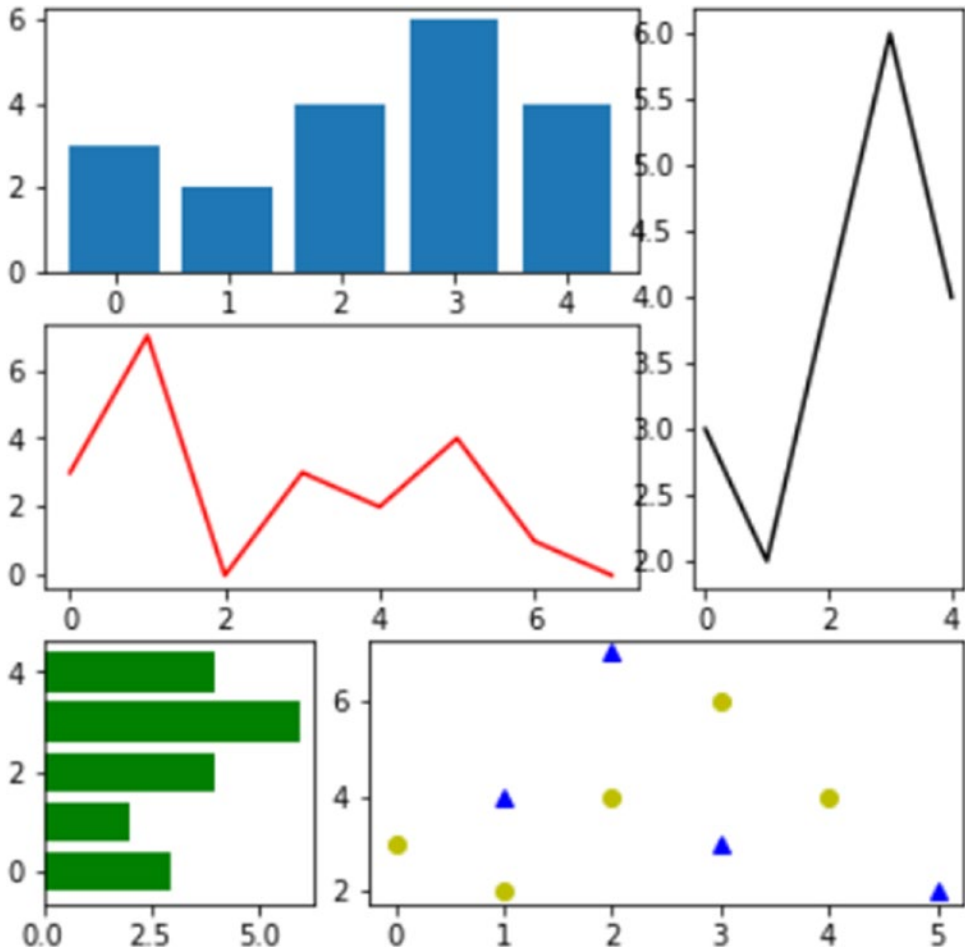


Figure 7-62. A grid of subplots can display many plots at the same time

Conclusions

In this chapter, you received all the fundamental aspects of the matplotlib library, and through a series of examples, you have learned about the basic tools for handling data visualization. You have become familiar with various examples of how to develop different types of charts with a few lines of code.

With this chapter, we conclude the part about the libraries that provides the basic tools to perform data analysis. In the next chapter, you will begin to treat topics most closely related to data analysis.