CHAPTER 3

The NumPy Library

NumPy is a basic package for scientific computing with Python and especially for data analysis. In fact, this library is the basis of a large amount of mathematical and scientific Python packages, and among them, as you will see later in the book, the pandas library. This library, specialized for data analysis, is fully developed using the concepts introduced by NumPy. In fact, the built-in tools provided by the standard Python library could be too simple or inadequate for most of the calculations in data analysis.

Having knowledge of the NumPy library is important to being able to use all scientific Python packages, and particularly, to use and understand the pandas library. The pandas library is the main subject of the following chapters.

If you are already familiar with this library, you can proceed directly to the next chapter; otherwise you may see this chapter as a way to review the basic concepts or to regain familiarity with it by running the examples in this chapter.

NumPy: A Little History

At the dawn of the Python language, the developers needed to perform numerical calculations, especially when this language was being used by the scientific community.

The first attempt was Numeric, developed by Jim Hugunin in 1995, which was followed by an alternative package called Numarray. Both packages were specialized for the calculation of arrays, and each had strengths depending on in which case they were used. Thus, they were used differently depending on the circumstances. This ambiguity led then to the idea of unifying the two packages. Travis Oliphant started to develop the NumPy library for this purpose. Its first release (v 1.0) occurred in 2006.

From that moment on, NumPy proved to be the extension library of Python for scientific computing, and it is currently the most widely used package for the calculation of multidimensional arrays and large arrays. In addition, the package comes with a range of functions that allow you to perform operations on arrays in a highly efficient way and perform high-level mathematical calculations.

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_3

Currently, NumPy is open source and licensed under BSD. There are many contributors who have expanded the potential of this library.

The NumPy Installation

Generally, this module is present as a basic package in most Python distributions; however, if not, you can install it later.

On Linux (Ubuntu and Debian), use:

```
sudo apt-get install python-numpy
```

On Linux (Fedora)

```
sudo yum install numpy scipy
```

On Windows with Anaconda, use:

```
conda install numpy
```

Once NumPy is installed on your distribution, to import the NumPy module within your Python session, write the following:

```
>>> import numpy as np
```

Ndarray: The Heart of the Library

The NumPy library is based on one main object: *ndarray* (which stands for N-dimensional array). This object is a multidimensional homogeneous array with a predetermined number of items: homogeneous because virtually all the items in it are of the same type and the same size. In fact, the data type is specified by another NumPy object called *dtype* (data-type); each ndarray is associated with only one type of dtype.

The number of the dimensions and items in an array is defined by its *shape*, a tuple of *N*-positive integers that specifies the size for each dimension. The dimensions are defined as *axes* and the number of axes as *rank*.

Moreover, another peculiarity of NumPy arrays is that their size is fixed, that is, once you define their size at the time of creation, it remains unchanged. This behavior is different from Python lists, which can grow or shrink in size.

The easiest way to define a new ndarray is to use the `array()` function, passing a Python list containing the elements to be included in it as an argument.

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
```

You can easily check that a newly created object is an ndarray by passing the new variable to the `type()` function.

```
>>> type(a)
<type 'numpy.ndarray'>
```

In order to know the associated dtype to the newly created ndarray, you have to use the `dtype` attribute.

Note The result of `dtype`, `shape`, and other attributes can vary among different operating systems and Python distributions.

```
>>> a.dtype
dtype('int64')
```

The just-created array has one axis, and then its rank is 1, while its shape should be (3,1). To obtain these values from the corresponding array, it is sufficient to use the `ndim` attribute for getting the axes, the `size` attribute to know the array length, and the `shape` attribute to get its shape.

```
>>> a.ndim
1
>>> a.size
3
>>> a.shape
(3,)
```

What you have just seen is the simplest case of a one-dimensional array. But the use of arrays can be easily extended to several dimensions. For example, if you define a two-dimensional array 2x2:

```
>>> b = np.array([[1.3, 2.4],[0.3, 4.1]])
>>> b.dtype
dtype('float64')
>>> b.ndim
2
>>> b.size
4
>>> b.shape
(2, 2)
```

This array has rank 2, since it has two axes, each of length 2.

Another important attribute is `itemsize`, which can be used with ndarray objects. It defines the size in bytes of each item in the array, and `data` is the buffer containing the actual elements of the array. This second attribute is still not generally used, since to access the data within the array you will use the indexing mechanism that you will see in the next sections.

```
>>> b.itemsize
8
>>> b.data
<read-write buffer for 0x0000000002D34DF0, size 32, offset 0 at
0x0000000002D5FEA0>
```

Create an Array

To create a new array, you can follow different paths. The most common path is the one you saw in the previous section through a list or sequence of lists as arguments to the `array()` function.

```
>>> c = np.array([[1, 2, 3],[4, 5, 6]])
>>> c
array([[1, 2, 3],
       [4, 5, 6]])
```

The `array()` function, in addition to lists, can accept tuples and sequences of tuples.

```
>>> d = np.array(((1, 2, 3),(4, 5, 6)))
>>> d
array([[1, 2, 3],
       [4, 5, 6]])
```

It can also accept sequences of tuples and interconnected lists .

```
>>> e = np.array([(1, 2, 3), [4, 5, 6], (7, 8, 9)])
>>> e
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Types of Data

So far you have seen only simple integer and float numeric values, but NumPy arrays are designed to contain a wide variety of data types (see Table 3-1). For example, you can use the data type string:

```
>>> g = np.array([['a', 'b'],['c', 'd']])
>>> g
array([['a', 'b'],
            ['c', 'd']],
      dtype='|<U1')
>>> g.dtype
dtype('<U1')
>>> g.dtype.name
'str32'
```

Table 3-1. *Data Types Supported by NumPy*

Data Type	Description
`bool_`	Boolean (true or false) stored as a byte
`int_`	Default integer type (same as C long; normally either `int64` or `int32`)
`intc`	Identical to C int (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `size_t`; normally either `int32` or `int64`)
`int8`	Byte (–128 to 127)
`int16`	Integer (–32768 to 32767)
`int32`	Integer (–2147483648 to 2147483647)
`int64`	Integer (–9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`
`float16`	Half precision float: sign bit, 5-bit exponent, 10-bit mantissa
`float32`	Single precision float: sign bit, 8-bit exponent, 23-bit mantissa
`float64`	Double precision float: sign bit, 11-bit exponent, 52-bit mantissa
`complex_`	Shorthand for complex128
`complex64`	Complex number, represented by two 32-bit floats (real and imaginary components)
`complex128`	Complex number, represented by two 64-bit floats (real and imaginary components)

The dtype Option

The `array()` function does not accept a single argument. You have seen that each ndarray object is associated with a `dtype` object that uniquely defines the type of data that will occupy each item in the array. By default, the `array()` function can associate the most suitable type according to the values contained in the sequence of lists or tuples. Actually, you can explicitly define the dtype using the `dtype` option as argument of the function.

For example, if you want to define an array with complex values, you can use the `dtype` option as follows:

```
>>> f = np.array([[1, 2, 3],[4, 5, 6]], dtype=complex)
>>> f
array([[ 1.+0.j,  2.+0.j,  3.+0.j],
       [ 4.+0.j,  5.+0.j,  6.+0.j]])
```

Intrinsic Creation of an Array

The NumPy library provides a set of functions that generate ndarrays with initial content, created with different values depending on the function. Throughout the chapter, and throughout the book, you'll discover that these features will be very useful. In fact, they allow a single line of code to generate large amounts of data.

The `zeros()` function, for example, creates a full array of zeros with dimensions defined by the shape argument. For example, to create a two-dimensional array 3x3, you can use:

```
>>> np.zeros((3, 3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

While the `ones()` function creates an array full of ones in a very similar way.

```
>>> np.ones((3, 3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

By default, the two functions created arrays with the `float64` data type. A feature that will be particularly useful is `arange()`. This function generates NumPy arrays with numerical sequences that respond to particular rules depending on the passed arguments. For example, if you want to generate a sequence of values between 0 and 10, you will be passed only one argument to the function, that is the value with which you want to end the sequence.

```
>>> np.arange(0, 10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If instead of starting from zero you want to start from another value, you simply specify two arguments: the first is the starting value and the second is the final value.

```
>>> np.arange(4, 10)
array([4, 5, 6, 7, 8, 9])
```

It is also possible to generate a sequence of values with precise intervals between them. If the third argument of the `arange()` function is specified, this will represent the gap between one value and the next one in the sequence of values.

```
>>> np.arange(0, 12, 3)
array([0, 3, 6, 9])
```

In addition, this third argument can also be a float.

```
>>> np.arange(0, 6, 0.6)
array([ 0. ,  0.6,  1.2,  1.8,  2.4,  3. ,  3.6,  4.2,  4.8,  5.4])
```

So far you have only created one-dimensional arrays. To generate two-dimensional arrays you can still continue to use the `arange()` function but combined with the `reshape()` function. This function divides a linear array in different parts in the manner specified by the `shape` argument.

```
>>> np.arange(0, 12).reshape(3, 4)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Another function very similar to `arange()` is `linspace()`. This function still takes as its first two arguments the initial and end values of the sequence, but the third argument, instead of specifying the distance between one element and the next, defines the number of elements into which we want the interval to be split.

```
>>> np.linspace(0,10,5)
array([  0. ,   2.5,   5. ,   7.5,  10. ])
```

Finally, another method to obtain arrays already containing values is to fill them with random values. This is possible using the `random()` function of the `numpy.random` module. This function will generate an array with many elements as specified in the argument.

```
>>> np.random.random(3)
array([ 0.78610272,  0.90630642,  0.80007102])
```

The numbers obtained will vary with every run. To create a multidimensional array, you simply pass the size of the array as an argument.

```
>>> np.random.random((3,3))
array([[ 0.07878569,  0.7176506 ,  0.05662501],
       [ 0.82919021,  0.80349121,  0.30254079],
       [ 0.93347404,  0.65868278,  0.37379618]])
```

Basic Operations

So far you have seen how to create a new NumPy array and how items are defined in it. Now it is the time to see how to apply various operations to them.

Arithmetic Operators

The first operations that you will perform on arrays are the arithmetic operators. The most obvious are adding and multiplying an array by a scalar.

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])

>>> a+4
array([4, 5, 6, 7])
>>> a*2
array([0, 2, 4, 6])
```

These operators can also be used between two arrays. In NumPy, these operations are *element-wise,* that is, the operators are applied only between corresponding elements. These are objects that occupy the same position, so that the end result will be a new array containing the results in the same location of the operands (see Figure 3-1).

```
>>> b = np.arange(4,8)
>>> b
array([4, 5, 6, 7])

>>> a + b
array([ 4,  6,  8, 10])
>>> a - b
array([-4, -4, -4, -4])
>>> a * b
array([ 0,  5, 12, 21])
```

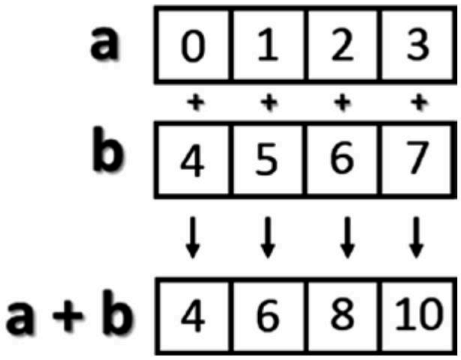


Figure 3-1. *Element-wise addition*

Moreover, these operators are also available for functions, provided that the value returned is a NumPy array. For example, you can multiply the array by the sine or the square root of the elements of array **b**.

```
>>> a * np.sin(b)
array([-0.        , -0.95892427, -0.558831  ,  1.9709598 ])
>>> a * np.sqrt(b)
array([ 0.        ,  2.23606798,  4.89897949,  7.93725393])
```

Moving on to the multidimensional case, even here the arithmetic operators continue to operate element-wise.

```
>>> A = np.arange(0, 9).reshape(3, 3)
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> B = np.ones((3, 3))
>>> B
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> A * B
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

The Matrix Product

The choice of operating element-wise is a peculiar aspect of the NumPy library. In fact, in many other tools for data analysis, the * operator is understood as a *matrix product* when it is applied to two matrices. Using NumPy, this kind of product is instead indicated by the `dot()` function. This operation is not element-wise.

```
>>> np.dot(A,B)
array([[  3.,   3.,   3.],
       [ 12.,  12.,  12.],
       [ 21.,  21.,  21.]])
```

The result at each position is the sum of the products of each element of the corresponding row of the first matrix with the corresponding element of the corresponding column of the second matrix. Figure 3-2 illustrates the process carried out during the matrix product (run for two elements).

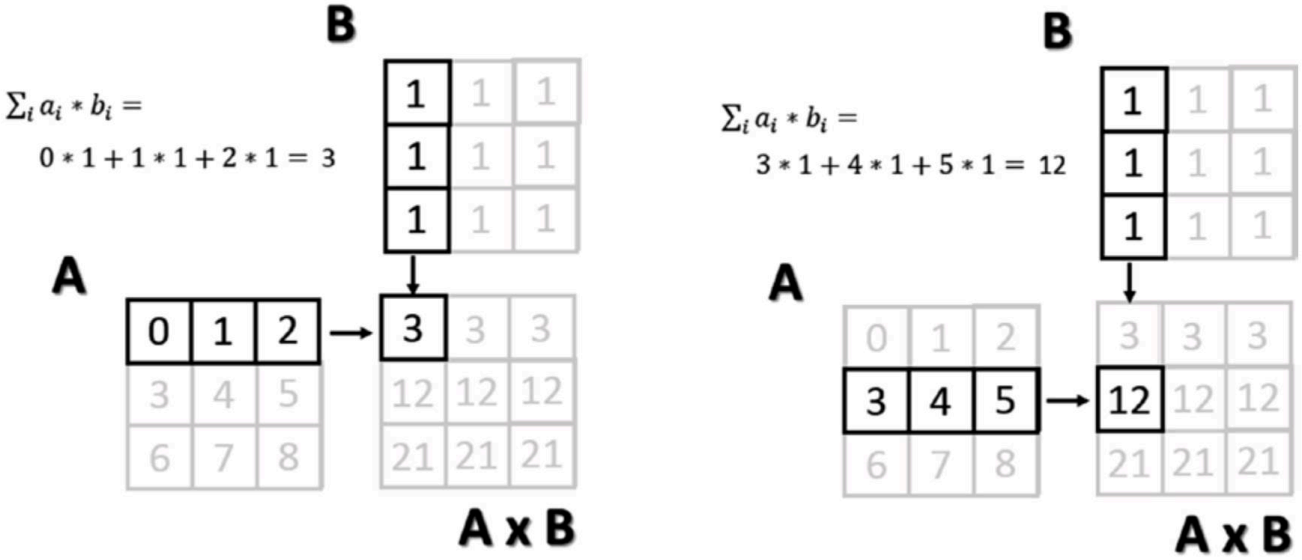


Figure 3-2. *Calculating matrix elements as a result of a matrix product*

An alternative way to write the matrix product is to see the `dot()` function as an object's function of one of the two matrices.

```
>>> A.dot(B)
array([[  3.,   3.,   3.],
       [ 12.,  12.,  12.],
       [ 21.,  21.,  21.]])
```

Note that since the matrix product is not a commutative operation, the order of the operands is important. Indeed, A * B is not equal to B * A.

```
>>> np.dot(B,A)
array([[  9.,  12.,  15.],
       [  9.,  12.,  15.],
       [  9.,  12.,  15.]])
```

Increment and Decrement Operators

Actually, there are no such operators in Python, since there are no operators called ++ or −−. To increase or decrease values, you have to use operators such as += and −=. These operators are not different from ones you saw earlier, except that instead of creating a new array with the results, they will reassign the results to the same array.

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> a += 1
>>> a
array([1, 2, 3, 4])
>>> a -= 1
>>> a
array([0, 1, 2, 3])
```

Therefore, using these operators is much more extensive than the simple incremental operators that increase the values by one unit, and they can be applied in many cases. For instance, you need them every time you want to change the values in an array without generating a new one.

```
array([0, 1, 2, 3])
>>> a += 4
>>> a
array([4, 5, 6, 7])
>>> a *= 2
>>> a
array([ 8, 10, 12, 14])
```

Universal Functions (ufunc)

A universal function, generally called `ufunc`, is a function operating on an array in an element-by-element fashion. This means that it acts individually on each single element of the input array to generate a corresponding result in a new output array. In the end, you obtain an array of the same size as the input.

There are many mathematical and trigonometric operations that meet this definition; for example, calculating the square root with `sqrt()`, the logarithm with `log()`, or the sin with `sin()`.

```
>>> a = np.arange(1, 5)
>>> a
array([1, 2, 3, 4])
```

```
>>> np.sqrt(a)
array([ 1.        ,  1.41421356,  1.73205081,  2.        ])
>>> np.log(a)
array([ 0.        ,  0.69314718,  1.09861229,  1.38629436])
>>> np.sin(a)
array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

Many functions are already implemented in the library NumPy.

Aggregate Functions

Aggregate functions perform an operation on a set of values, an array for example, and produce a single result. Therefore, the sum of all the elements in an array is an aggregate function. Many functions of this kind are implemented within the class ndarray.

```
>>> a = np.array([3.3, 4.5, 1.2, 5.7, 0.3])
>>> a.sum()
15.0
>>> a.min()
0.29999999999999999
>>> a.max()
5.7000000000000002
>>> a.mean()
3.0
>>> a.std()
2.0079840636817816
```

Indexing, Slicing, and Iterating

In the previous sections, you saw how to create an array and how to perform operations on it. In this section, you will see how to manipulate these objects. You'll learn how to select elements through indexes and slices, in order to obtain the values contained in them or to make assignments in order to change their values. Finally, you will also see how you can make iterations within them.

Indexing

Array indexing always uses square brackets ([ ]) to index the elements of the array so that the elements can then be referred individually for various, uses such as extracting a value, selecting items, or even assigning a new value.

When you create a new array, an appropriate scale index is also automatically created (see Figure 3-3).

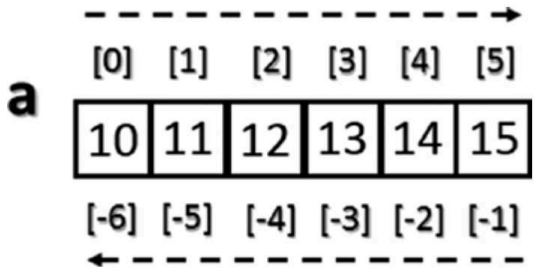


Figure 3-3. *Indexing a monodimensional ndarray*

In order to access a single element of an array, you can refer to its index.

```
>>> a = np.arange(10, 16)
>>> a
array([10, 11, 12, 13, 14, 15])
>>> a[4]
14
```

The NumPy arrays also accept negative indexes. These indexes have the same incremental sequence from 0 to –1, –2, and so on, but in practice they cause the final element to move gradually toward the initial element, which will be the one with the more negative index value.

```
>>> a[-1]
15
>>> a[-6]
10
```

To select multiple items at once, you can pass array of indexes in square brackets.

```
>>> a[[1, 3, 4]]
array([11, 13, 14])
```

Moving on to the two-dimensional case, namely the matrices, they are represented as rectangular arrays consisting of rows and columns, defined by two axes, where axis 0 is represented by the rows and axis 1 is represented by the columns. Thus, indexing in this case is represented by a pair of values: the first value is the index of the row and the second is the index of the column. Therefore, if you want to access the values or select elements in the matrix, you will still use square brackets, but this time there are two values [row index, column index] (see Figure 3-4).

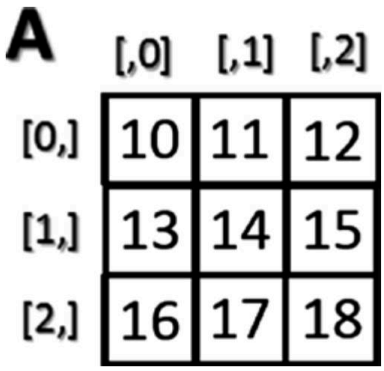


Figure 3-4. *Indexing a bidimensional array*

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

If you want to remove the element of the third column in the second row, you have to insert the pair [1, 2].

```
>>> A[1, 2]
15
```

Slicing

Slicing allows you to extract portions of an array to generate new arrays. When when you use the Python lists to slice arrays, the resulting arrays are copies, but in NumPy, the arrays are views of the same underlying buffer.

Depending on the portion of the array that you want to extract (or view), you must use the slice syntax; that is, you will use a sequence of numbers separated by colons (`:`) within square brackets.

If you want to extract a portion of the array, for example one that goes from the second to the sixth element, you have to insert the index of the starting element, that is 1, and the index of the final element, that is 5, separated by `:`.

```
>>> a = np.arange(10, 16)
>>> a
array([10, 11, 12, 13, 14, 15])
>>> a[1:5]
array([11, 12, 13, 14])
```

Now if you want to extract an item from the previous portion and skip a specific number of following items, then extract the next and skip again, you can use a third number that defines the gap in the sequence of the elements. For example, with a value of 2, the array will take the elements in an alternating fashion.

```
>>> a[1:5:2]
array([11, 13])
```

To better understand the slice syntax, you also should look at cases where you do not use explicit numerical values. If you omit the first number, NumPy implicitly interprets this number as 0 (i.e., the initial element of the array). If you omit the second number, this will be interpreted as the maximum index of the array; and if you omit the last number this will be interpreted as 1. All the elements will be considered without intervals.

```
>>> a[::2]
array([10, 12, 14])
>>> a[:5:2]
array([10, 12, 14])
>>> a[:5:]
array([10, 11, 12, 13, 14])
```

In the case of a two-dimensional array, the slicing syntax still applies, but it is separately defined for the rows and columns. For example, if you want to extract only the first row:

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
>>> A[0,:]
array([10, 11, 12])
```

As you can see in the second index, if you leave only the colon without defining a number, you will select all the columns. Instead, if you want to extract all the values of the first column, you have to write the inverse.

```
>>> A[:,0]
array([10, 13, 16])
```

Instead, if you want to extract a smaller matrix, you need to explicitly define all intervals with indexes that define them.

```
>>> A[0:2, 0:2]
array([[10, 11],
       [13, 14]])
```

If the indexes of the rows or columns to be extracted are not contiguous, you can specify an array of indexes.

```
>>> A[[0,2], 0:2]
array([[10, 11],
       [16, 17]])
```

Iterating an Array

In Python, the iteration of the items in an array is really very simple; you just need to use the `for` construct.

```
>>> for i in a:
...      print(i)
...
10
11
12
13
14
15
```

Of course, even here, moving to the two-dimensional case, you could think of applying the solution of two nested loops with the `for` construct. The first loop will scan the rows of the array, and the second loop will scan the columns. Actually, if you apply the `for` loop to a matrix, it will always perform a scan according to the first axis.

```
>>> for row in A:
...      print(row)
...
[10 11 12]
[13 14 15]
[16 17 18]
```

If you want to make an iteration element by element, you can use the following construct, using the `for` loop on `A.flat`.

```
>>> for item in A.flat:
...       print(item)
...
10
11
12
13
14
15
```

```
16
17
18
```

However, despite all this, NumPy offers an alternative and more elegant solution than the `for` loop. Generally, you need to apply an iteration to apply a function on the rows or on the columns or on an individual item. If you want to launch an aggregate function that returns a value calculated for every single column or on every single row, there is an optimal way that leaves it to NumPy to manage the iteration: the `apply_along_axis()` function.

This function takes three arguments: the aggregate function, the axis on which to apply the iteration, and the array. If the option `axis` equals 0, then the iteration evaluates the elements column by column, whereas if `axis` equals 1 then the iteration evaluates the elements row by row. For example, you can calculate the average values first by column and then by row.

```
>>> np.apply_along_axis(np.mean, axis=0, arr=A)
array([ 13.,  14.,  15.])
>>> np.apply_along_axis(np.mean, axis=1, arr=A)
array([ 11.,  14.,  17.])
```

In the previous case, you used a function already defined in the NumPy library, but nothing prevents you from defining your own functions. You also used an aggregate function. However, nothing forbids you from using an `ufunc`. In this case, iterating by column and by row produces the same result. In fact, using a `ufunc` performs one iteration element-by-element.

```
>>> def foo(x):
...     return x/2
...
>>> np.apply_along_axis(foo, axis=1, arr=A)
array([[5.,  5.5, 6. ],
       [6.5, 7.,  7.5],
       [8.,  8.5, 9. ]])
>>> np.apply_along_axis(foo, axis=0, arr=A)
array([[5.,  5.5, 6.],
       [6.5, 7.,  7.5],
       [8.,  8.5, 9.]])
```

As you can see, the ufunc function halves the value of each element of the input array regardless of whether the iteration is performed by row or by column.

Conditions and Boolean Arrays

So far you have used indexing and slicing to select or extract a subset of an array. These methods use numerical indexes. An alternative way to selectively extract the elements in an array is to use the conditions and Boolean operators.

Suppose you wanted to select all the values that are less than 0.5 in a 4x4 matrix containing random numbers between 0 and 1.

```
>>> A = np.random.random((4, 4))
>>> A
array([[ 0.03536295,  0.0035115 ,  0.54742404,  0.68960999],
       [ 0.21264709,  0.17121982,  0.81090212,  0.43408927],
       [ 0.77116263,  0.04523647,  0.84632378,  0.54450749],
       [ 0.86964585,  0.6470581 ,  0.42582897,  0.22286282]])
```

Once a matrix of random numbers is defined, if you apply an operator condition, you will receive as a return value a Boolean array containing true values in the positions in which the condition is satisfied. In this example, that is all the positions in which the values are less than 0.5.

```
>>> A < 0.5
array([[ True,  True, False, False],
       [ True,  True, False,  True],
       [False,  True, False, False],
       [False, False,  True,  True]], dtype=bool)
```

Actually, the Boolean arrays are used implicitly for making selections of parts of arrays. In fact, by inserting the previous condition directly inside the square brackets, you will extract all elements smaller than 0.5, so as to obtain a new array.

```
>>> A[A < 0.5]
array([ 0.03536295,  0.0035115 ,  0.21264709,  0.17121982,  0.43408927,
        0.04523647,  0.42582897,  0.22286282])
```

Shape Manipulation

You already saw, during the creation of a two-dimensional array, how it is possible to convert a one-dimensional array into a matrix, thanks to the `reshape()` function.

```
>>> a = np.random.random(12)
>>> a
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305,  0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642,  0.10862692,
        0.41894881,  0.73581471])
>>> A = a.reshape(3, 4)
>>> A
array([[ 0.77841574,  0.39654203,  0.38188665,  0.26704305],
       [ 0.27519705,  0.78115866,  0.96019214,  0.59328414],
       [ 0.52008642,  0.10862692,  0.41894881,  0.73581471]])
```

The `reshape()` function returns a new array and can therefore create new objects. However if you want to modify the object by modifying the shape, you have to assign a tuple containing the new dimensions directly to its `shape` attribute.

```
>>> a.shape = (3, 4)
>>> a
array([[ 0.77841574,  0.39654203,  0.38188665,  0.26704305],
       [ 0.27519705,  0.78115866,  0.96019214,  0.59328414],
       [ 0.52008642,  0.10862692,  0.41894881,  0.73581471]])
```

As you can see, this time it is the starting array to change shape and there is no object returned. The inverse operation is also possible, that is, you can convert a two-dimensional array into a one-dimensional array, by using the `ravel()` function.

```
>>> a = a.ravel()
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305,  0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642,  0.10862692,
        0.41894881,  0.73581471])
```

Or even here acting directly on the shape attribute of the array itself.

```
>>> a.shape = (12)
>>> a
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305,  0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642,  0.10862692,
        0.41894881,  0.73581471])
```

Another important operation is transposing a matrix, which is inverting the columns with the rows. NumPy provides this feature with the `transpose()` function.

```
>>> A.transpose()
array([[ 0.77841574,  0.27519705,  0.52008642],
       [ 0.39654203,  0.78115866,  0.10862692],
       [ 0.38188665,  0.96019214,  0.41894881],
       [ 0.26704305,  0.59328414,  0.73581471]])
```

Array Manipulation

Often you need to create an array using already created arrays. In this section, you will see how to create new arrays by joining or splitting arrays that are already defined.

Joining Arrays

You can merge multiple arrays to form a new one that contains all of the arrays. NumPy uses the concept of *stacking*, providing a number of functions in this regard. For example, you can perform vertical stacking with the `vstack()` function, which combines the second array as new rows of the first array. In this case, the array grows in a vertical direction. By contrast, the `hstack()` function performs horizontal stacking; that is, the second array is added to the columns of the first array.

```
>>> A = np.ones((3, 3))
>>> B = np.zeros((3, 3))
>>> np.vstack((A, B))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
```

```
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.hstack((A,B))
array([[ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.]])
```

Two other functions performing stacking between multiple arrays are `column_stack()` and `row_stack()`. These functions operate differently than the two previous functions. Generally these functions are used with one-dimensional arrays, which are stacked as columns or rows in order to form a new two-dimensional array.

```
>>> a = np.array([0, 1, 2])
>>> b = np.array([3, 4, 5])
>>> c = np.array([6, 7, 8])
>>> np.column_stack((a, b, c))
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
>>> np.row_stack((a, b, c))
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Splitting Arrays

In the previous section, you saw how to assemble multiple arrays through stacking. Now you will see how to divide an array into several parts. In NumPy, you use splitting to do this. Here too, you have a set of functions that work both horizontally with the `hsplit()` function and vertically with the `vsplit()` function.

```
>>> A = np.arange(16).reshape((4, 4))
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Thus, if you want to split the array horizontally, meaning the width of the array is divided into two parts, the 4x4 matrix A will be split into two 2x4 matrices.

```
>>> [B,C] = np.hsplit(A, 2)
>>> B
array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
>>> C
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

Instead, if you want to split the array vertically, meaning the height of the array is divided into two parts, the 4x4 matrix A will be split into two 4x2 matrices.

```
>>> [B,C] = np.vsplit(A, 2)
>>> B
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> C
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

A more complex command is the `split()` function, which allows you to split the array into nonsymmetrical parts. Passing the array as an argument, you have also to specify the indexes of the parts to be divided. If you use the option `axis = 1`, then the indexes will be columns; if instead the option is `axis = 0`, then they will be row indexes.

For example, if you want to divide the matrix into three parts, the first of which will include the first column, the second will include the second and the third column, and the third will include the last column, you must specify three indexes in the following way.

```
>>> [A1,A2,A3] = np.split(A,[1,3],axis=1)
>>> A1
array([[ 0],
       [ 4],
```

```
       [ 8],
       [12]])
>>> A2
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10],
       [13, 14]])
>>> A3
array([[ 3],
       [ 7],
       [11],
       [15]])
```

You can do the same thing by row.

```
>>> [A1,A2,A3] = np.split(A,[1,3],axis=0)
>>> A1
array([[0, 1, 2, 3]])
>>> A2
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> A3
array([[12, 13, 14, 15]])
```

This feature also includes the functionalities of the `vsplit()` and `hsplit()` functions.

General Concepts

This section describes the general concepts underlying the NumPy library. The difference between copies and views is when they return values. The mechanism of broadcasting, which occurs implicitly in many NumPy functions, is also covered in this section.

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.

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

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

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

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

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

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

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

Installation of pandas

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

Installation from Anaconda

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

```
conda list pandas
```

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

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

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

```
conda install pandas
```

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

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

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

   - pandas
```

The following new packages will be installed:

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

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

```
conda update pandas
```

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

Installation from PyPI

pandas can also be installed by PyPI using this command:

```
pip install pandas
```

Installation on Linux

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

On Debian and Ubuntu distributions, use this command:

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

While on OpenSuse and Fedora, enter the following command:

```
zypper in python-pandas
```

Installation from Source

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

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

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

A Module Repository for Windows

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

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

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

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

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

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

Testing Your pandas Installation

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

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

```
nosetests pandas
```

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

NOSE MODULE

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

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

Getting Started with pandas

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

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

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

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

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

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

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

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

Introduction to pandas Data Structures

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

- Series
- Dataframes

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

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

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

The Series

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

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

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

Declaring a Series

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

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

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

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

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

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

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

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

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

Selecting the Internal Elements

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

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

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

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

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

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

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

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

Assigning Values to the Elements

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

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

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

Defining a Series from NumPy Arrays and Other Series

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

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

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

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

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

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

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

Filtering Values

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

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

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

Operations and Mathematical Functions

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

You can simply write the arithmetic expression for the operators.

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

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

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

Evaluating Vales

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

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

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

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

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

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

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

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

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

NaN Values

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

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

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

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

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

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

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

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

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

Series as Dictionaries

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

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

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

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

Operations Between Series

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

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

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

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

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

The DataFrame

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

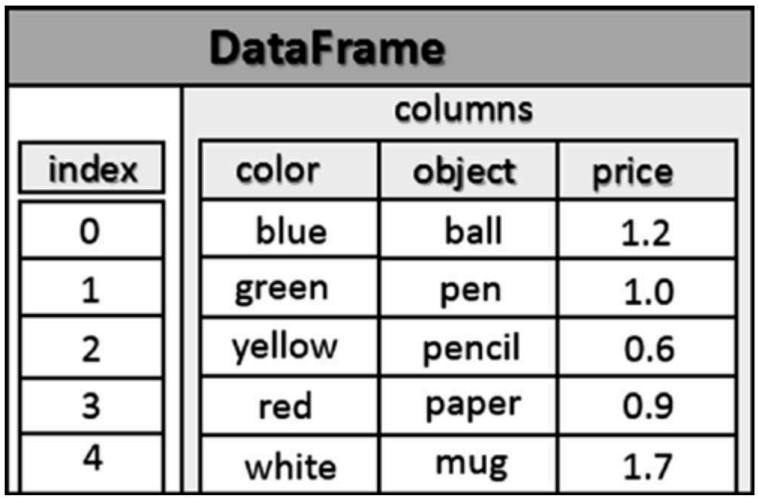


Figure 4-2. *The dataframe structure*

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

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

Defining a Dataframe

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

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

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

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

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

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

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

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

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

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

Selecting Elements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Assigning Values

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Membership of a Value

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

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

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

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

Deleting a Column

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

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

Filtering

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

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

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

DataFrame from Nested dict

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

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

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

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

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

Transposition of a Dataframe

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

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

The Index Objects

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

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

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

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

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

Methods on Index

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

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

Index with Duplicate Labels

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

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

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

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

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

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

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

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

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://matplolib.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

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_7

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.

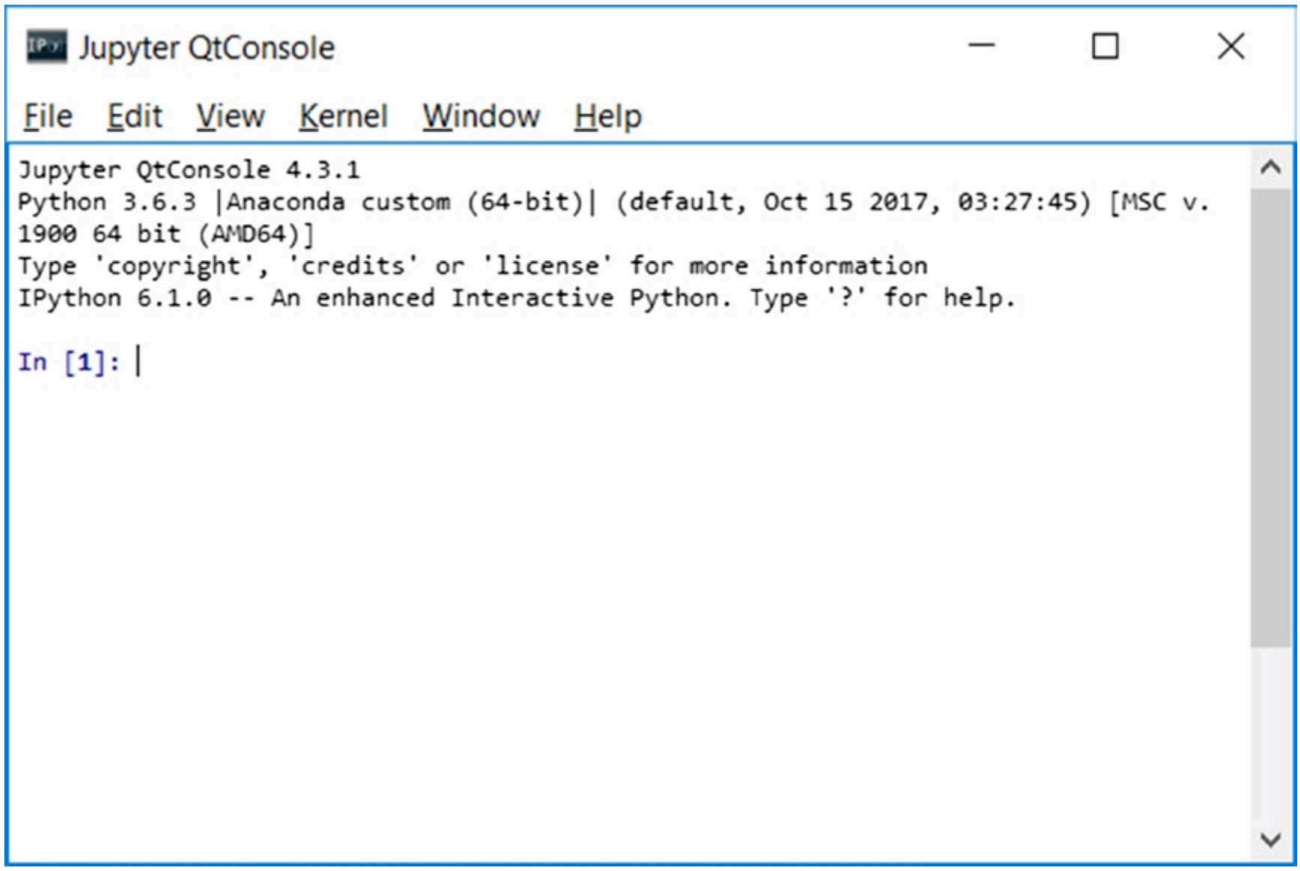


Figure 7-1. *The IPython QtConsole*

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.

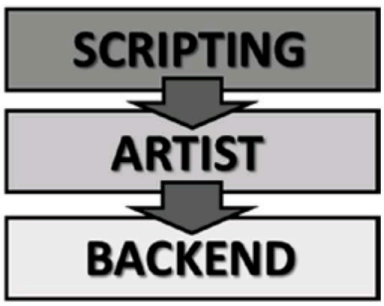


Figure 7-2. *The three layers of the matplotlib architecture*

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).

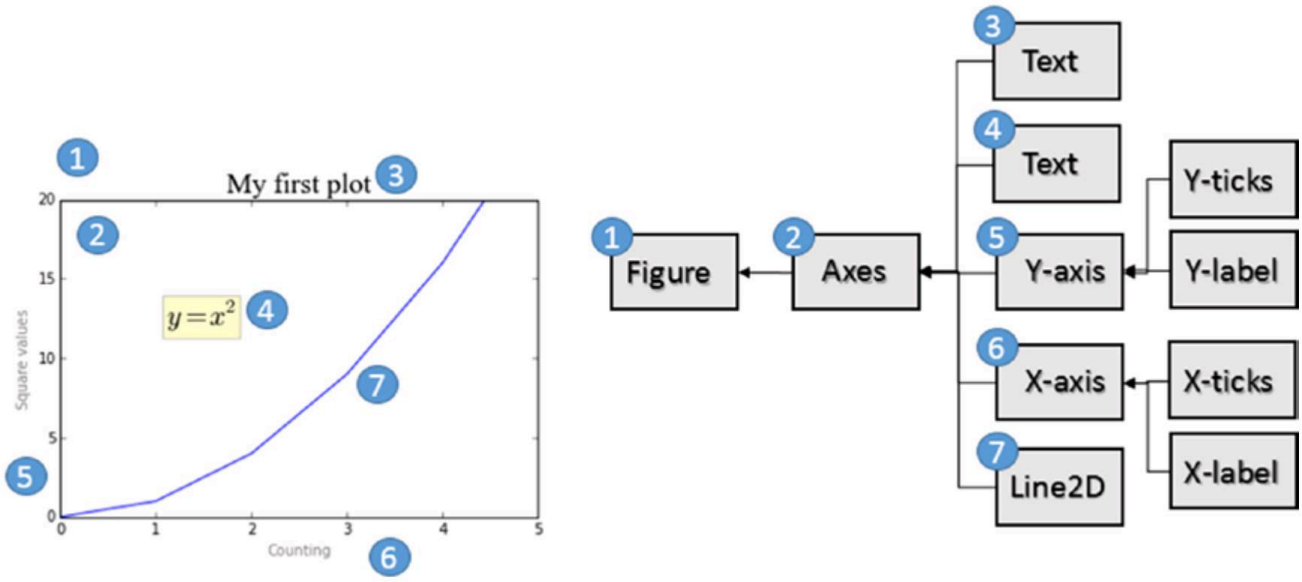


Figure 7-3. *Each element of a chart corresponds to an instance of Artist structured in a hierarchy*

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.

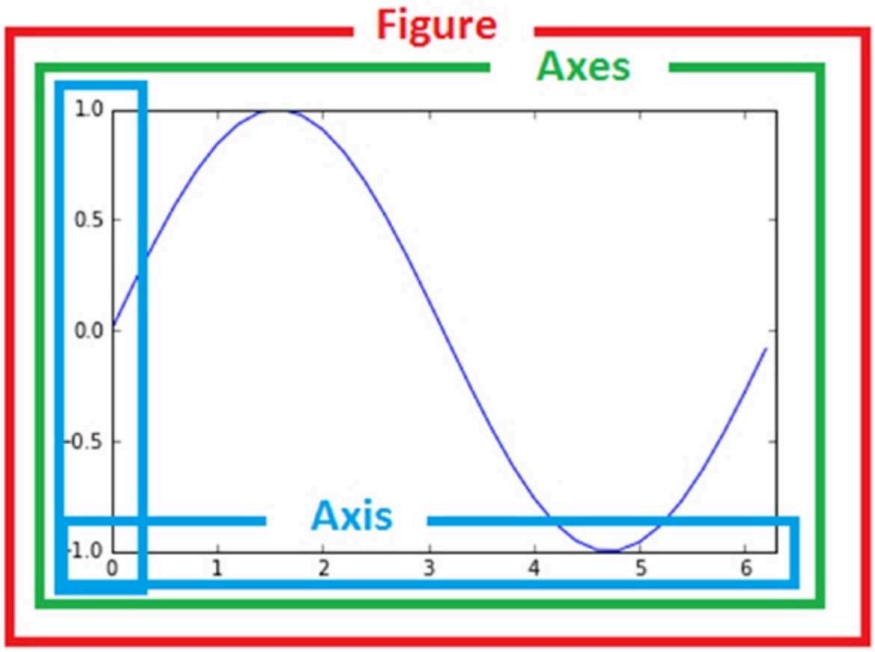


Figure 7-4. *The three main artist objects in the hierarchy of the Artist layer*

- *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.

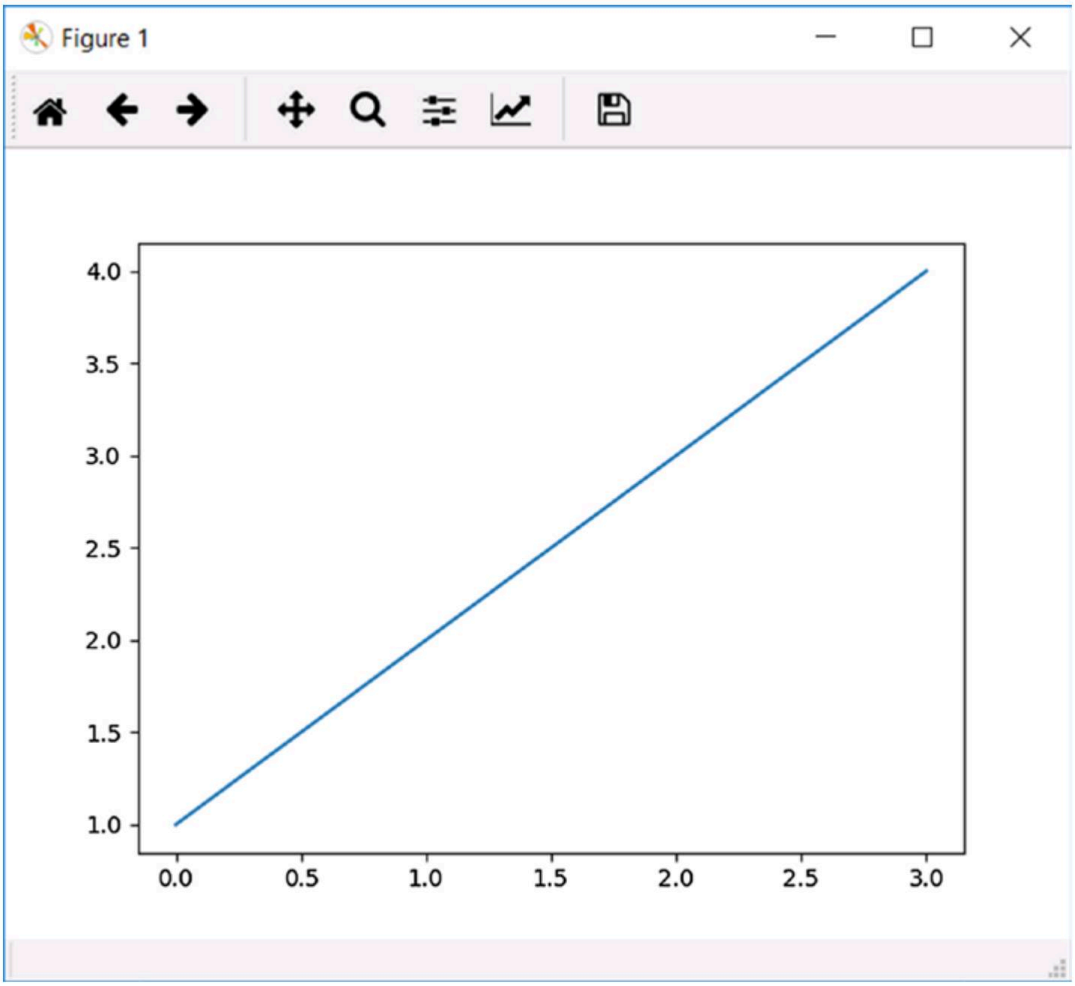
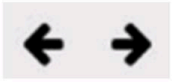


Figure 7-5. *The plotting window*

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).

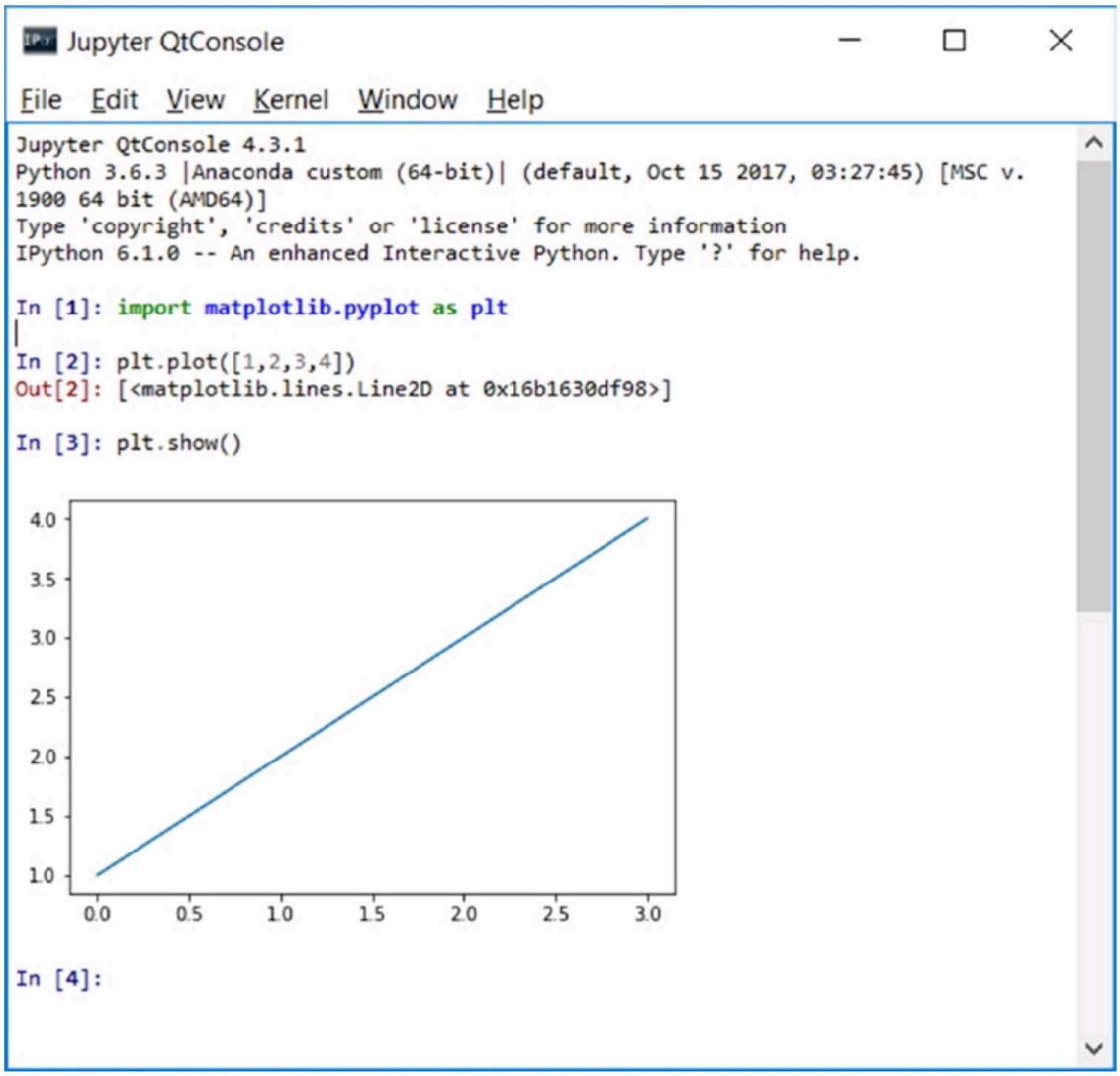


Figure 7-6. *The QtConsole shows the chart directly as output*

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.

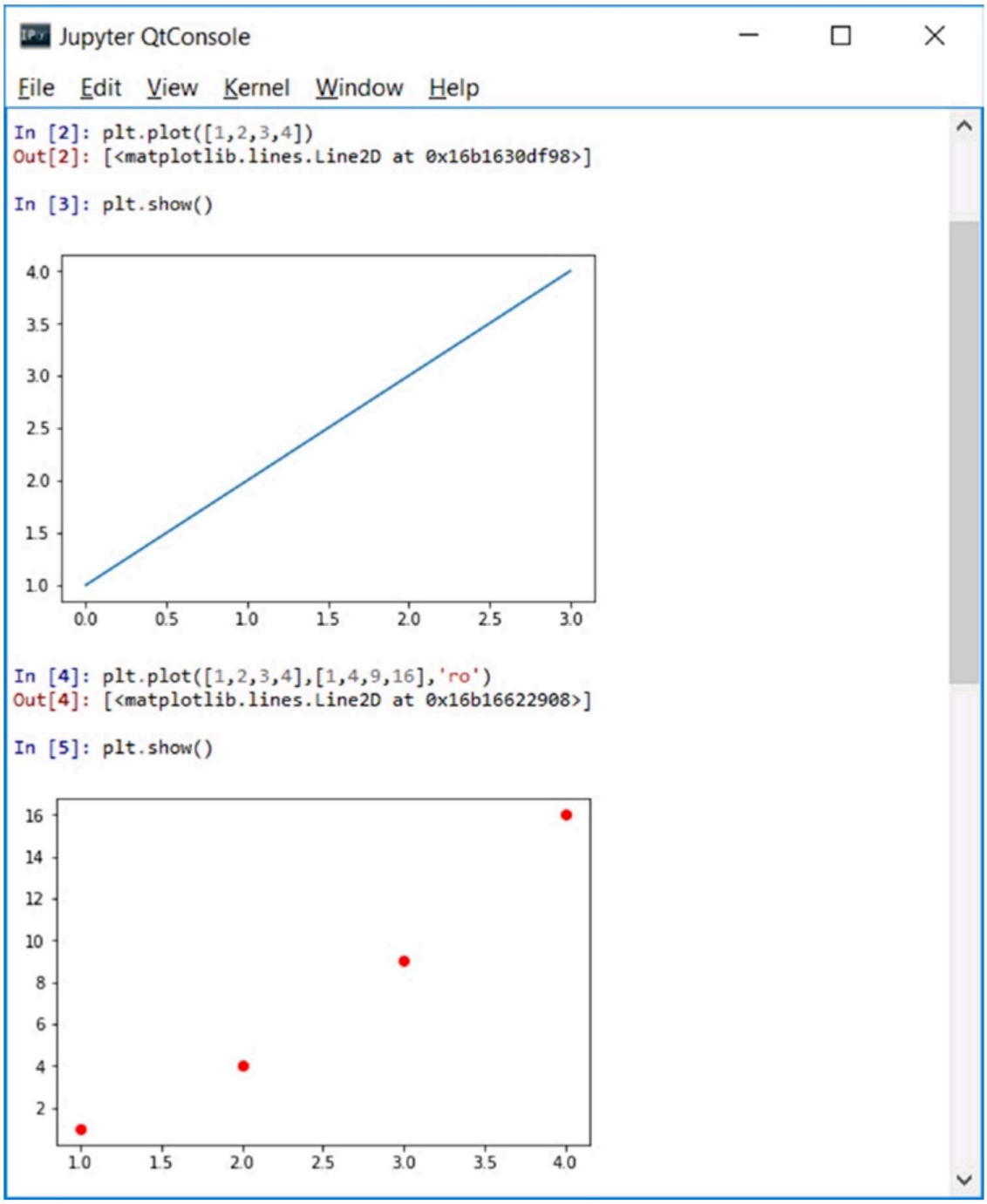


Figure 7-7. *The pairs of (x,y) values are represented in the plot by red circles*

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.

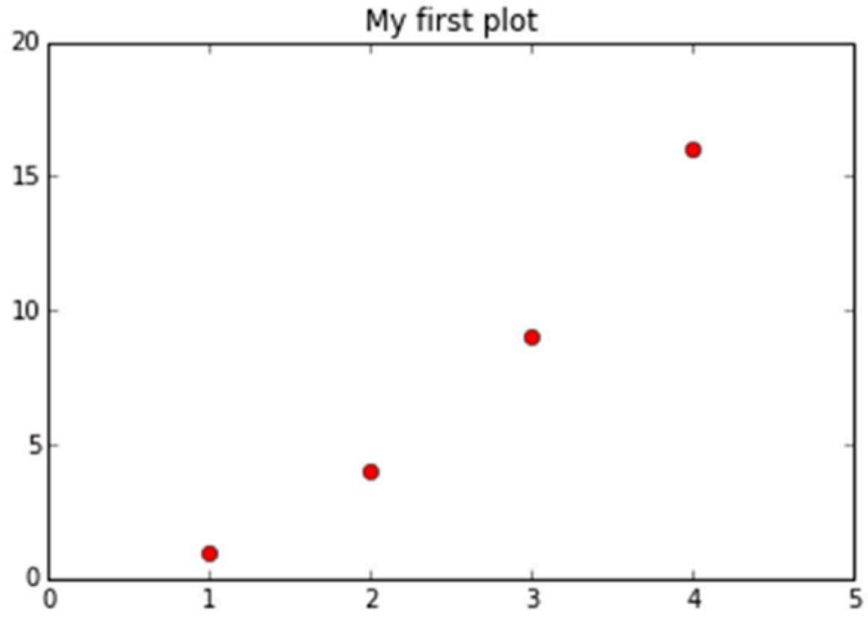


Figure 7-8. *The plot after the properties have been set*

matplotlib and NumPy

Even the matplot 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>]
```

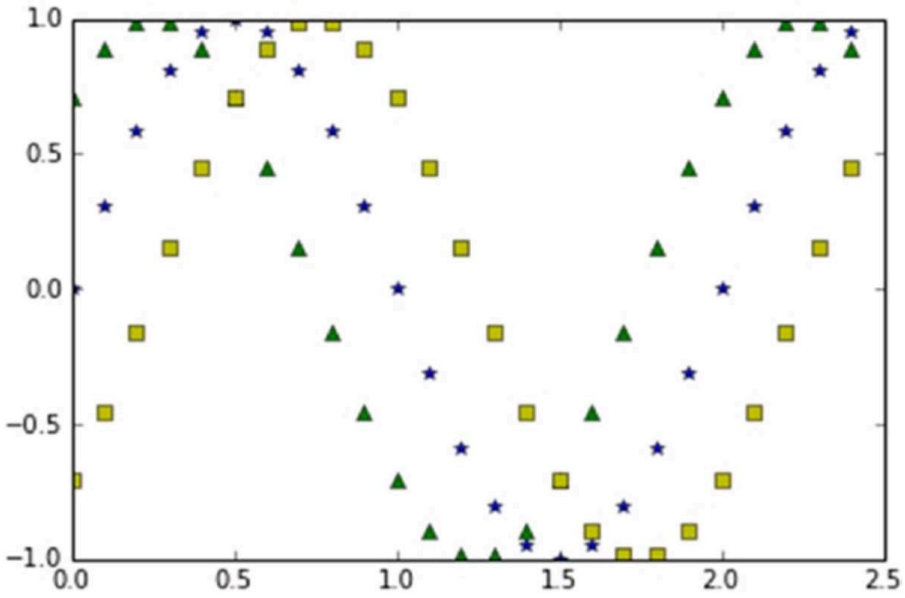


Figure 7-9. *Three sinusoidal trends phase-shifted by π / 4 represented by markers*

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.

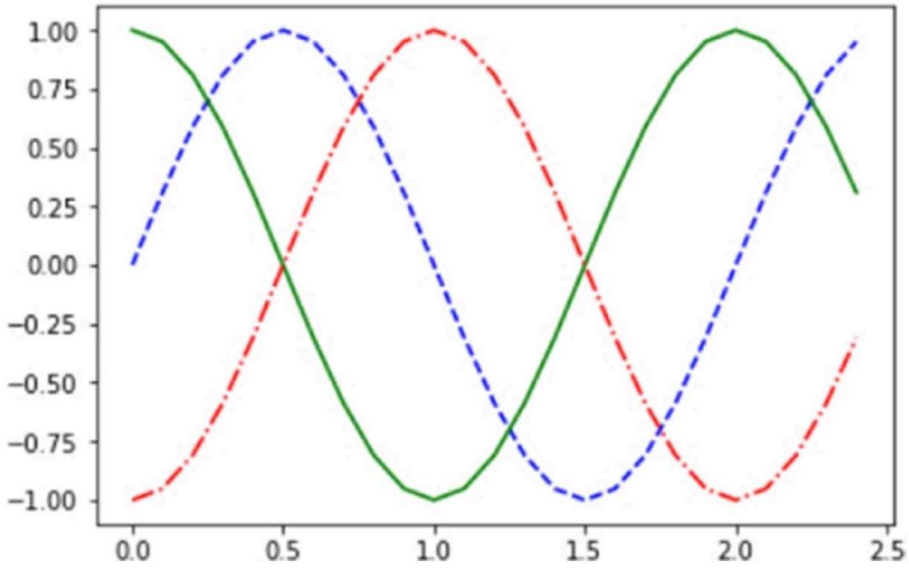


Figure 7-10. *This chart represents the three sinusoidal patterns with colored lines*

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>]
```

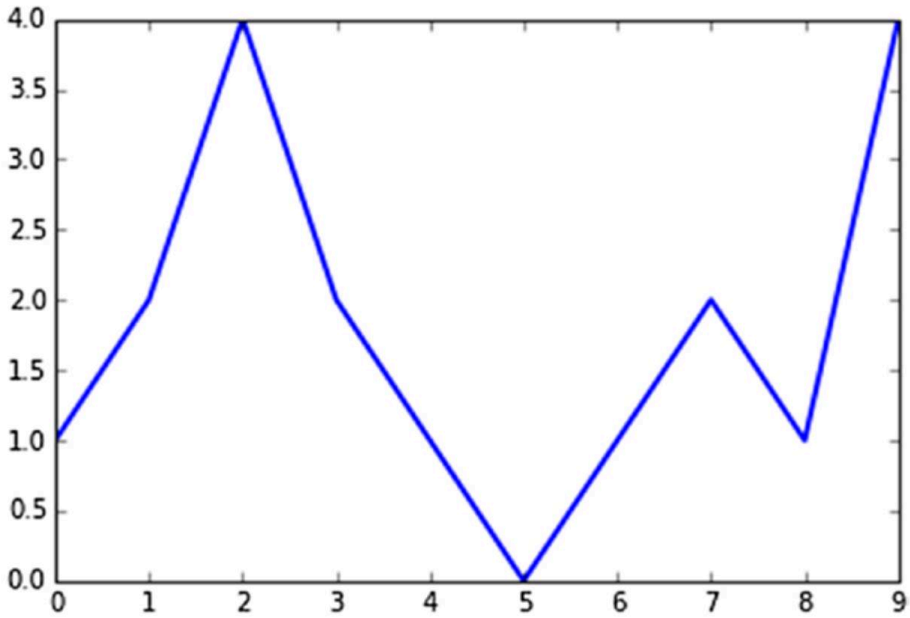


Figure 7-11. *The thickness of a line can be set directly from the plot() function*

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>]
```

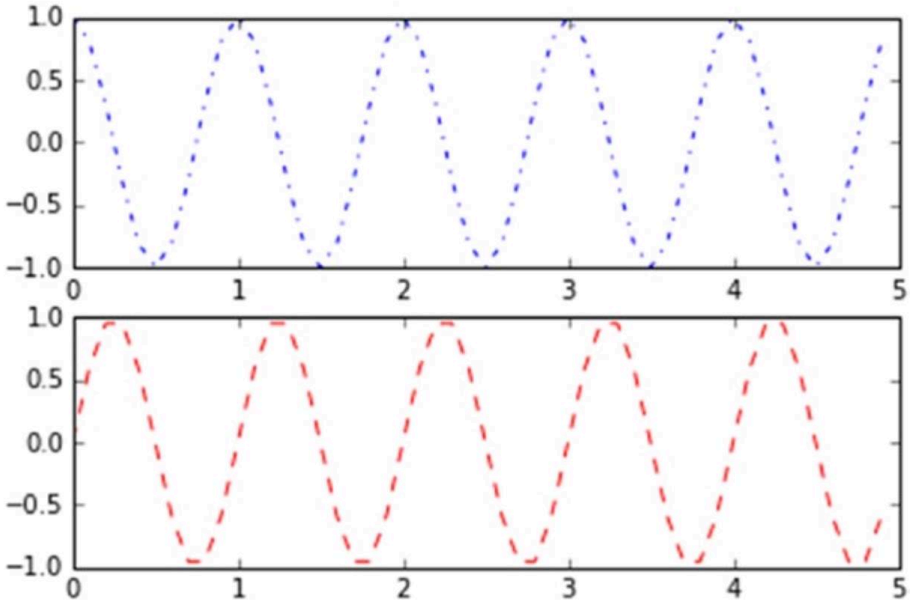


Figure 7-12. *The figure has been divided into two horizontal subplots*

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>]
```

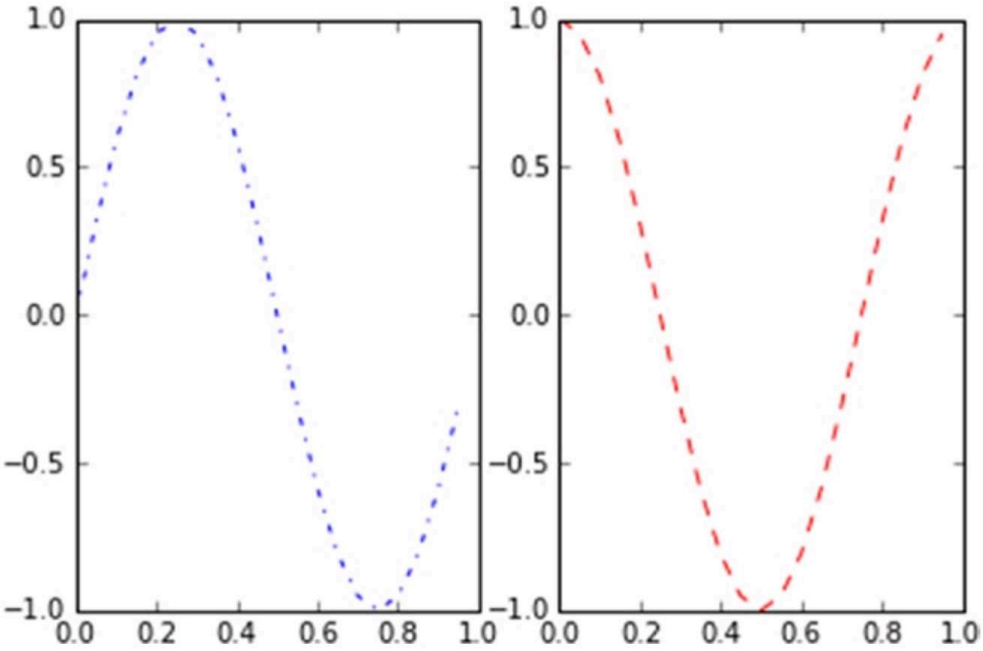


Figure 7-13. *The figure has been divided into two vertical subplots*

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>]
```

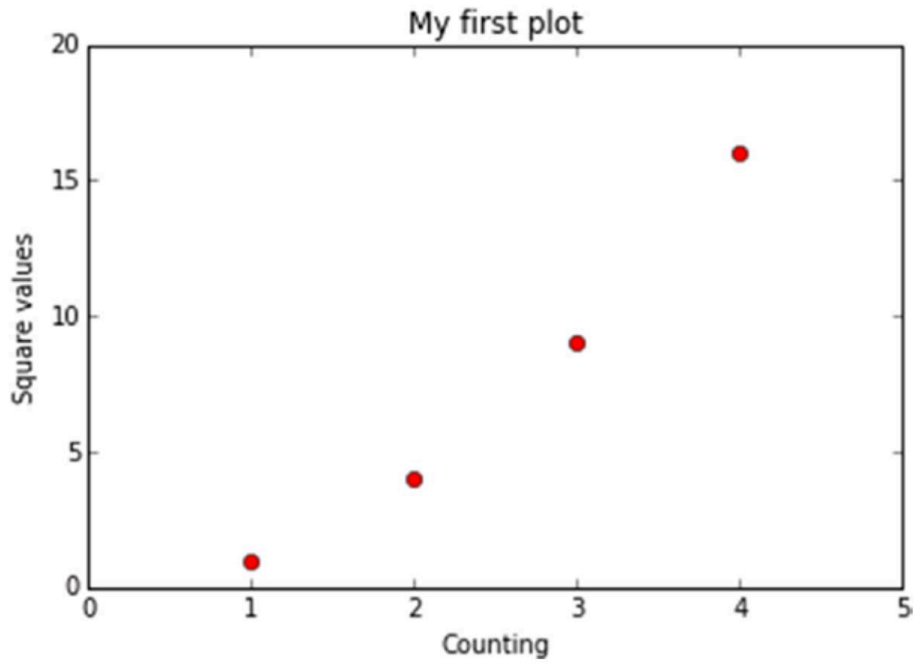


Figure 7-14. *A plot is more informative by adding axis labels*

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>]
```

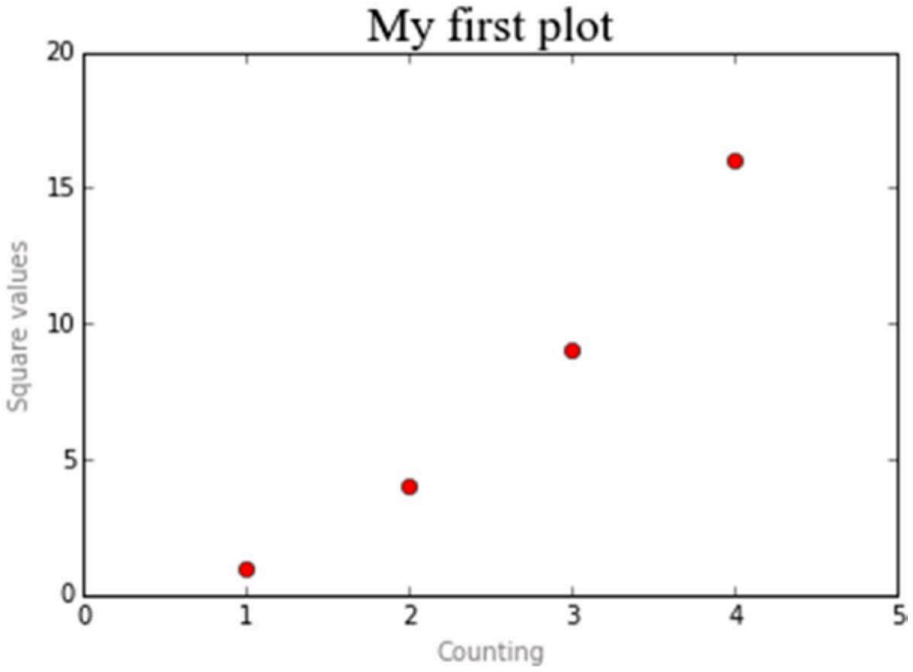


Figure 7-15. *The text can be modified by setting the keywords*

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>]
```

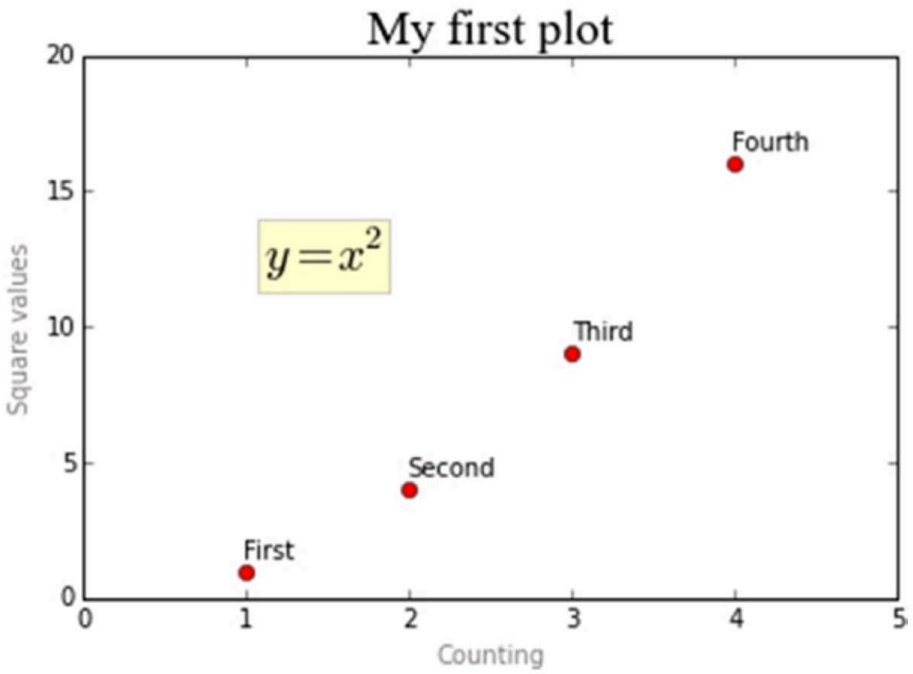


Figure 7-17. *Any mathematical expression can be seen in the context of a chart*

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>]
```

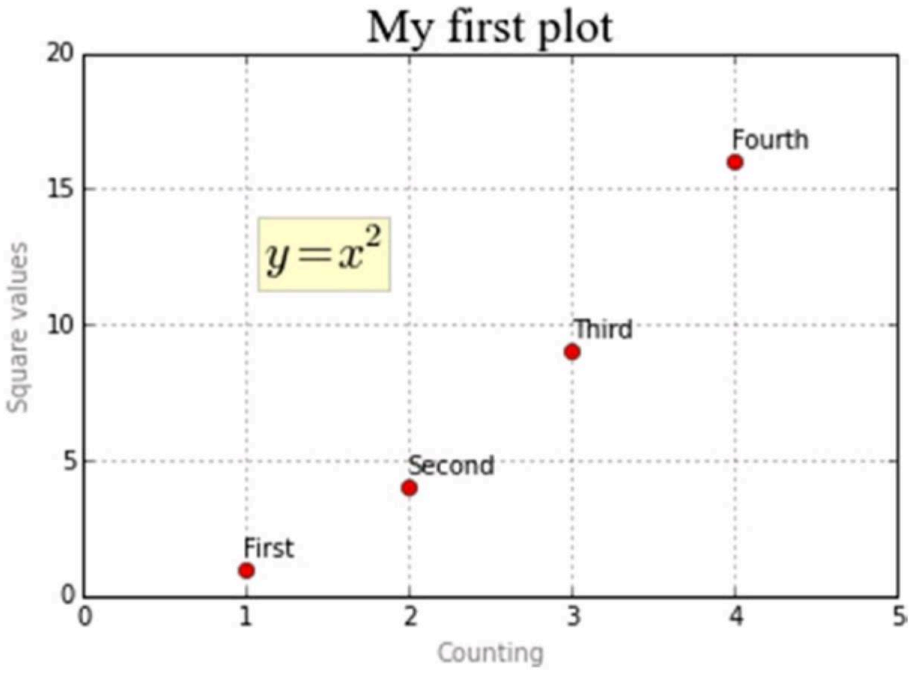


Figure 7-18. *A grid makes it easier to read the values of the data points represented on a chart*

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>
```

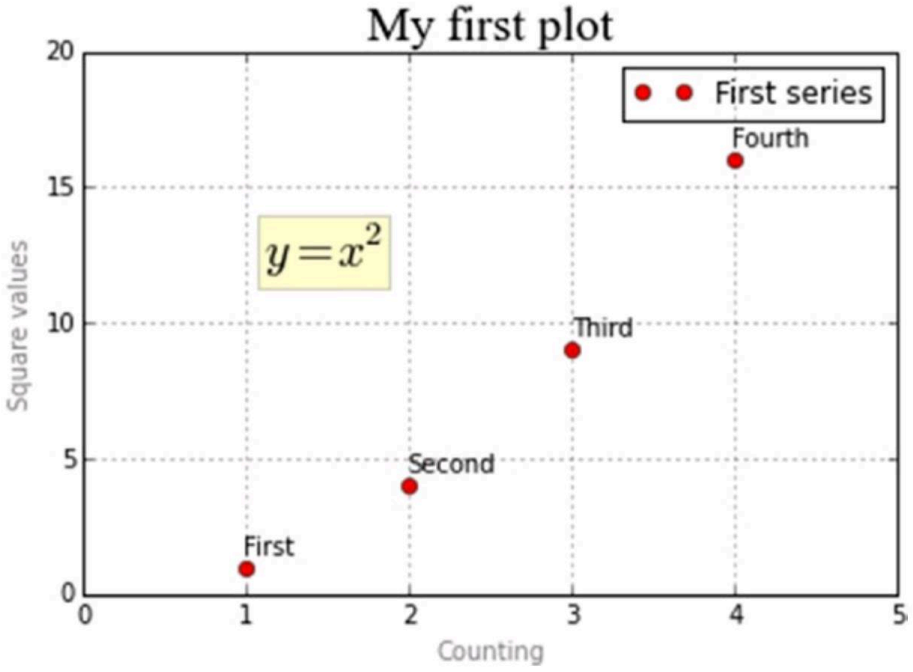


Figure 7-19. *A legend is added in the upper-right corner by default*

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>
```

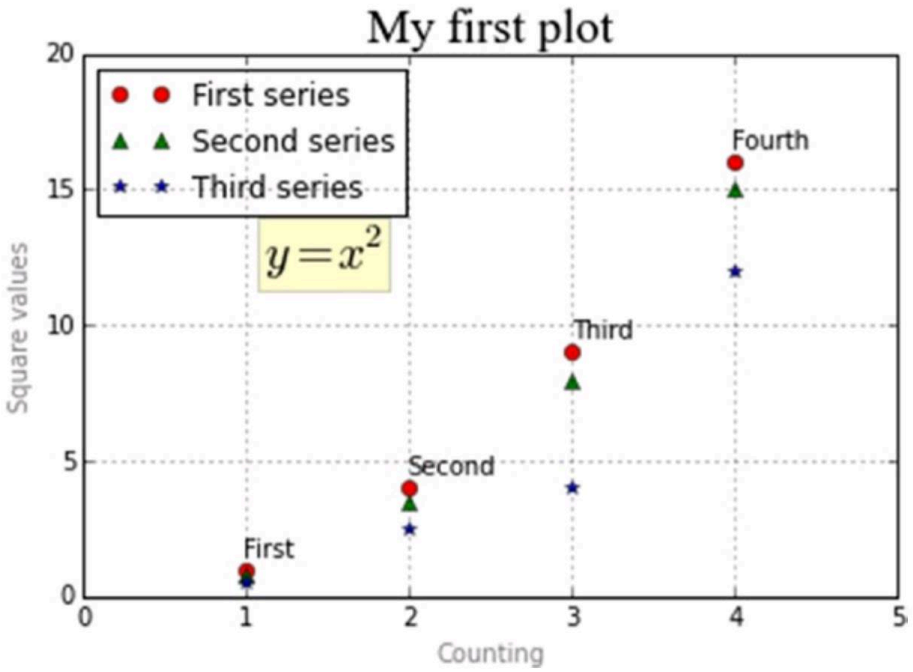


Figure 7-20. *A legend is necessary in every multiseries chart*

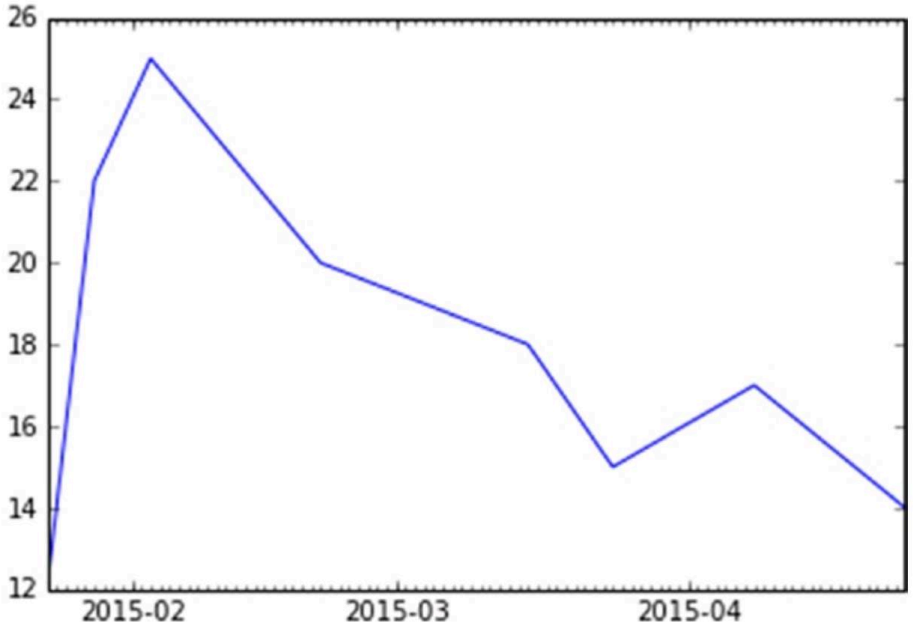


Figure 7-25. *Now the tick labels of the x-axis refer only to the months, making the plot more readable*

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>]
```

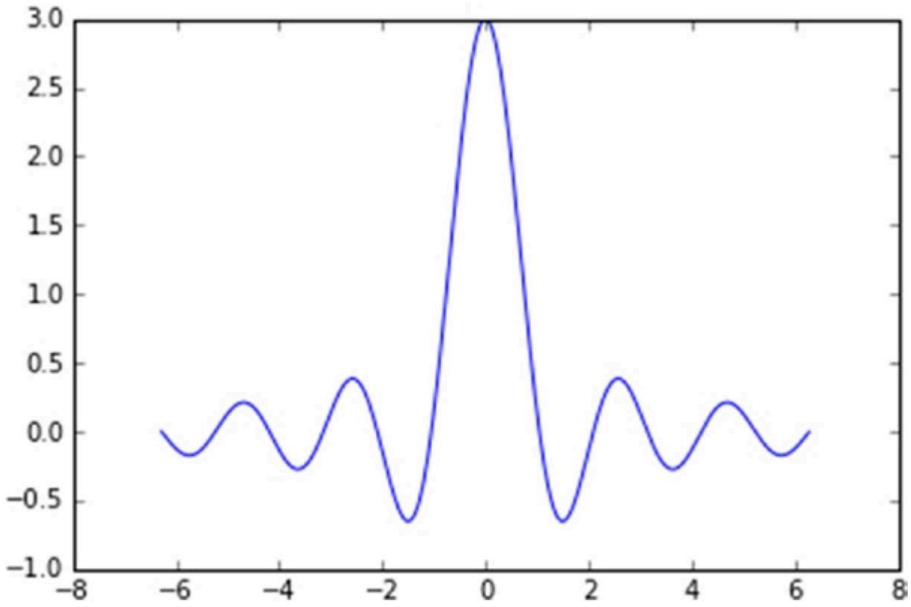


Figure 7-26. *A mathematical function represented in a line chart*

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).

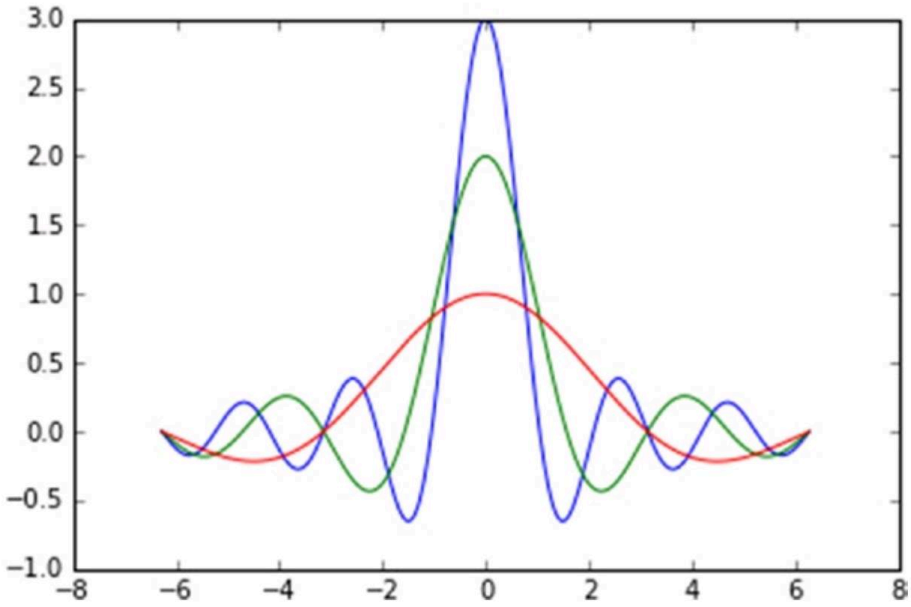


Figure 7-27. *Three different series are drawn with different colors in the same chart*

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='--')
```

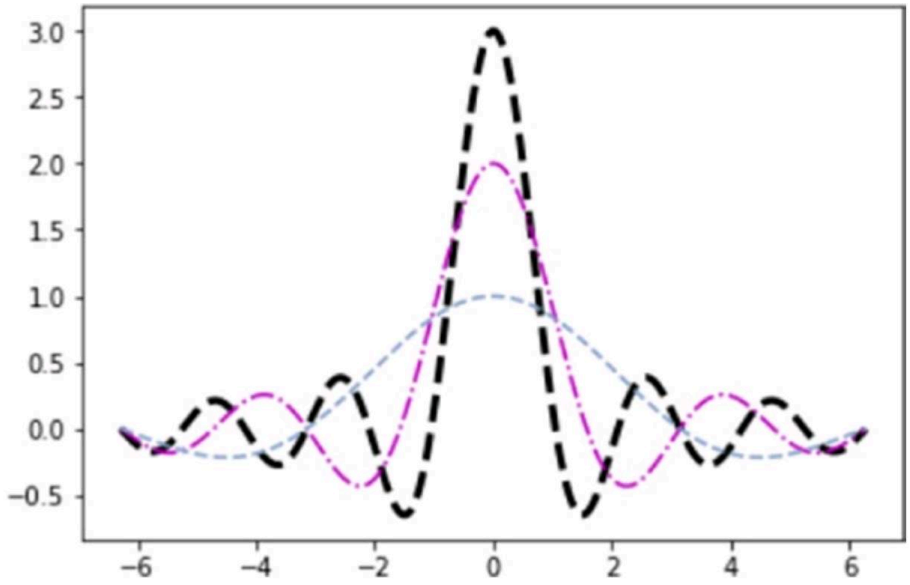


Figure 7-28. *You can define colors and line styles using character codes*

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.

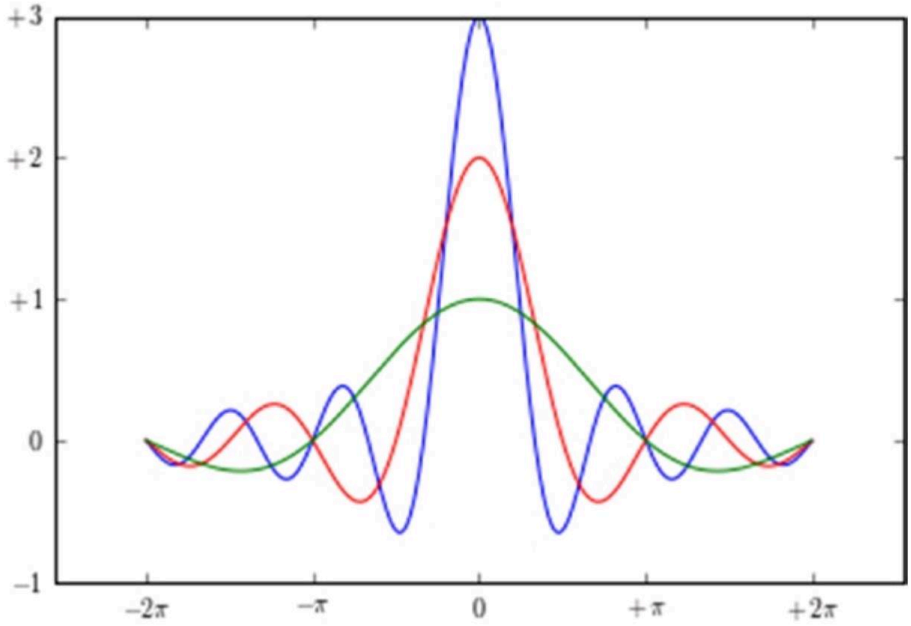


Figure 7-29. *The tick label can be improved adding text with LaTeX format*

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.

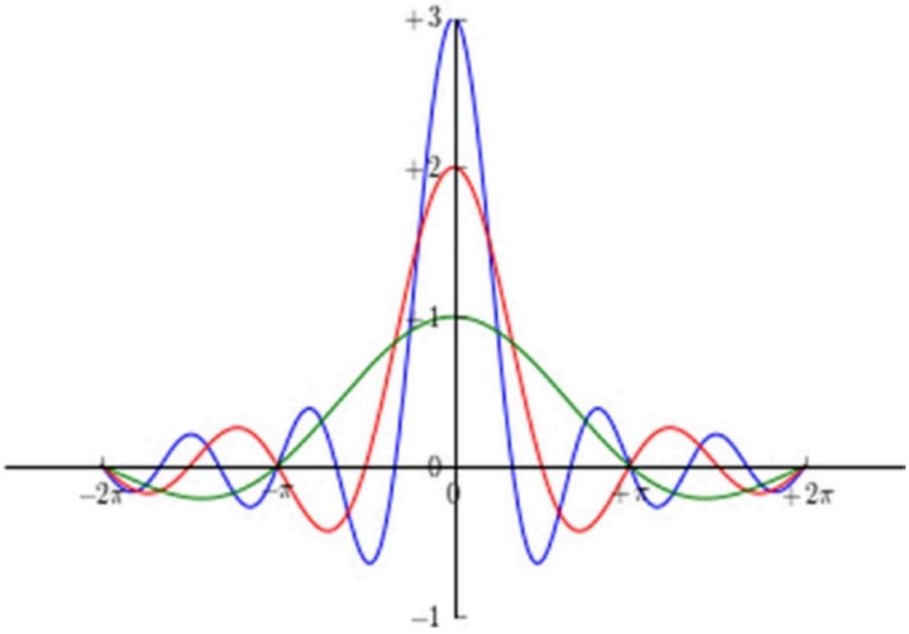


Figure 7-30. *The chart shows two Cartesian axes*

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 sinx/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.

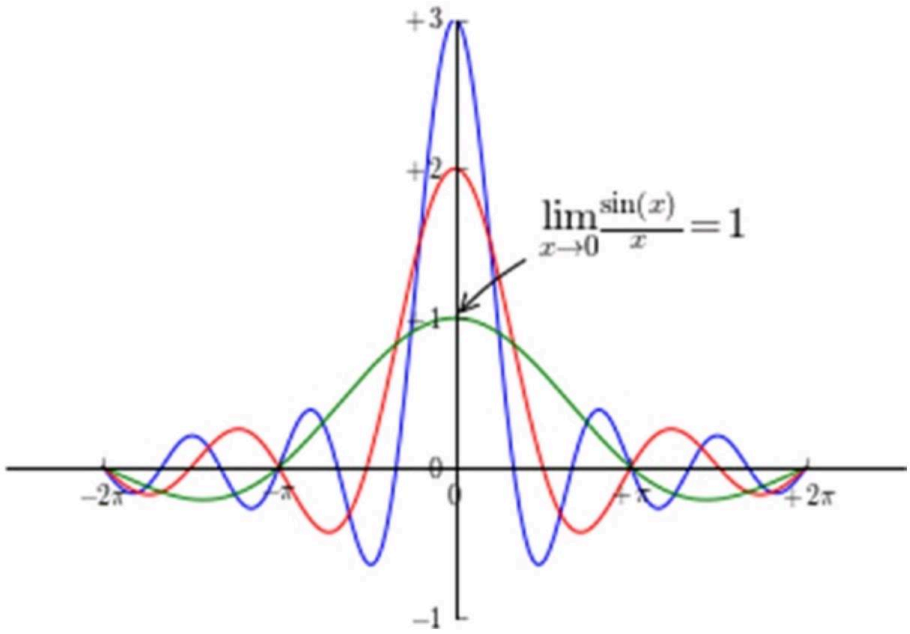


Figure 7-31. *Mathematical expressions can be added to a chart with the annotate() function*

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)
```

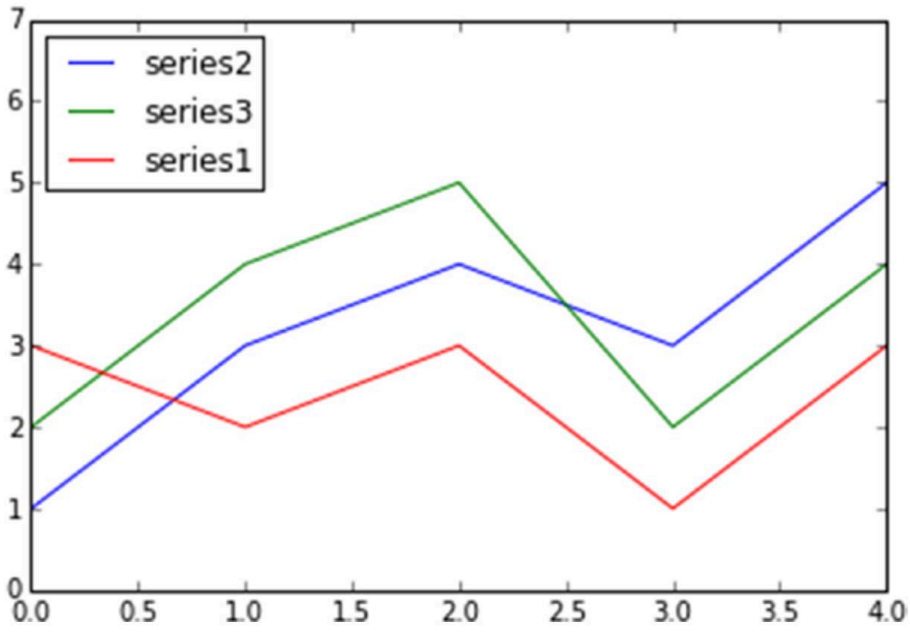


Figure 7-32. *The multiseries line chart displays the data within a pandas dataframe*

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)
```

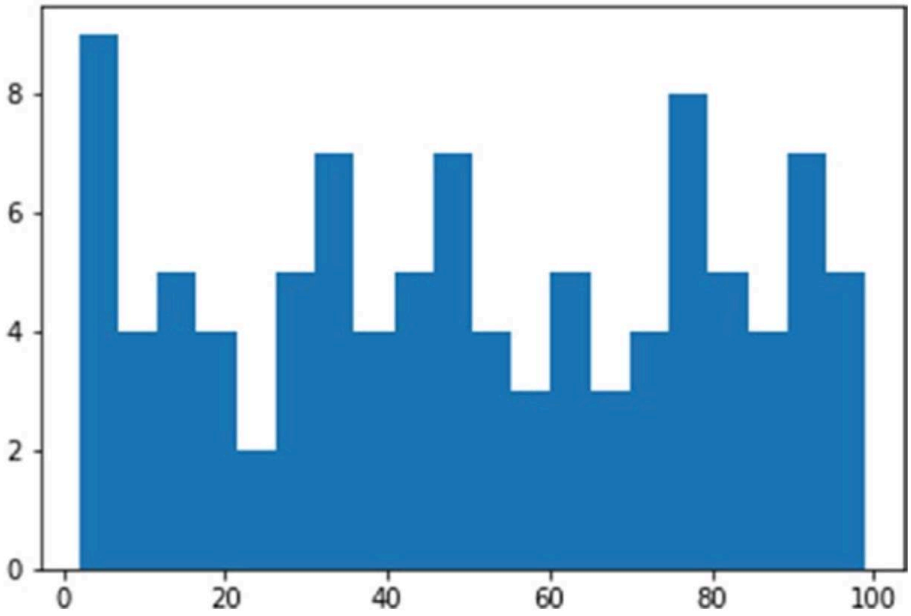


Figure 7-33. *The histogram shows the occurrences in each bin*

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.

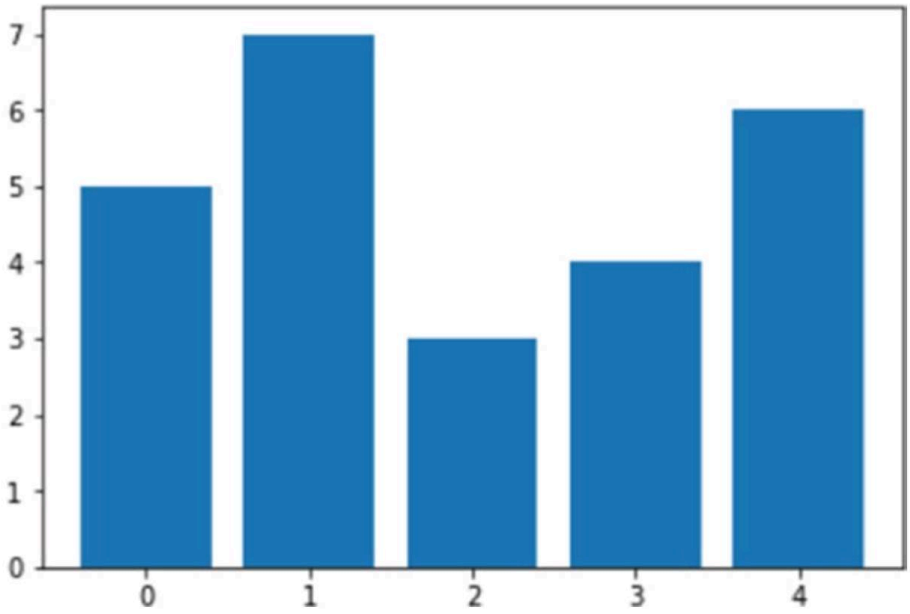


Figure 7-34. *The simplest bar chart with matplotlib*

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'])
```

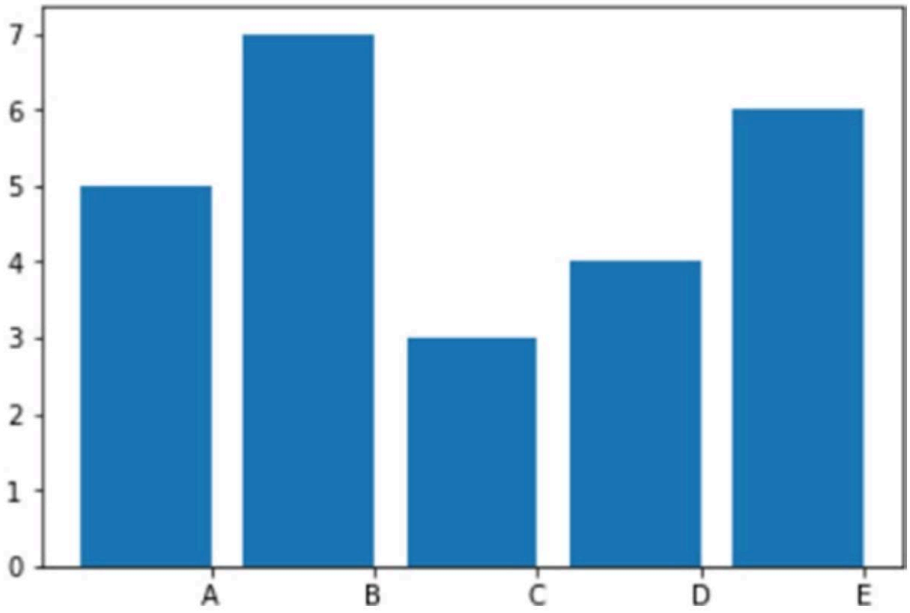


Figure 7-35. *A simple bar chart with categories on the x-axis*

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)
```

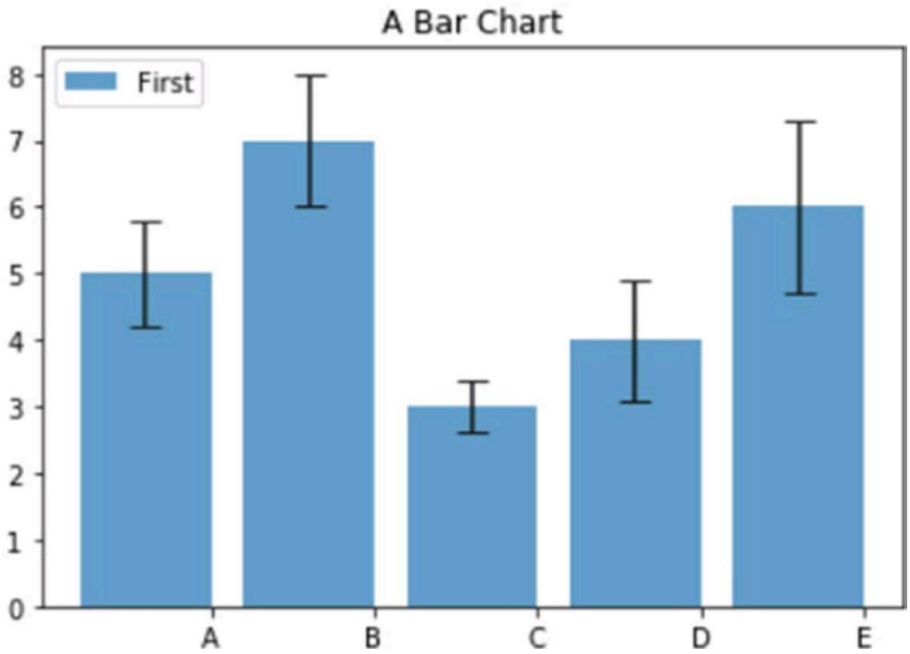


Figure 7-36. *A bar chart with error bars*

Horizontal Bar Charts

So far you have seen the bar chart oriented vertically. There are also bar chart 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)
```

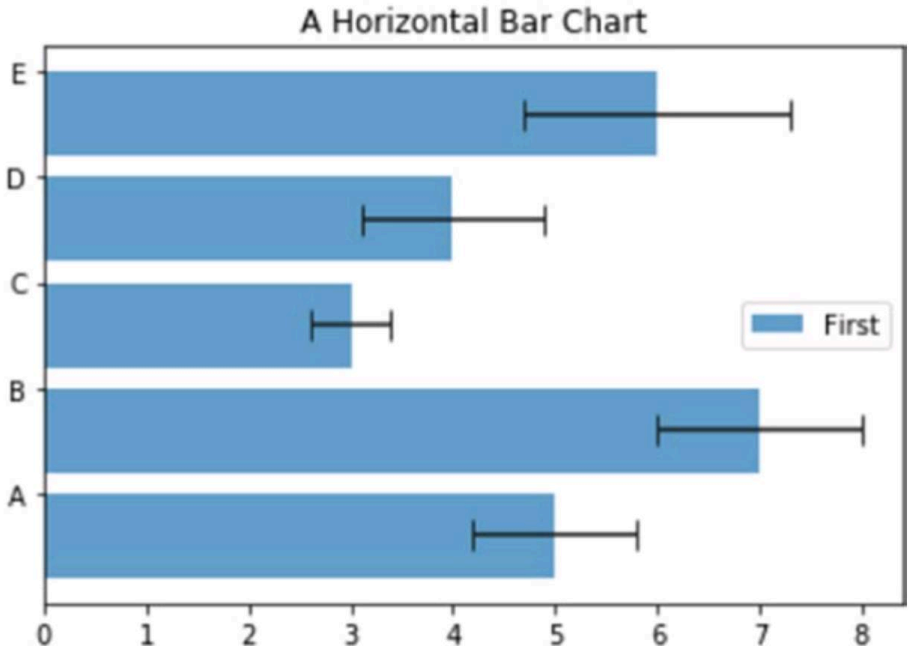


Figure 7-37. *A simple horizontal bar chart*

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'])
```

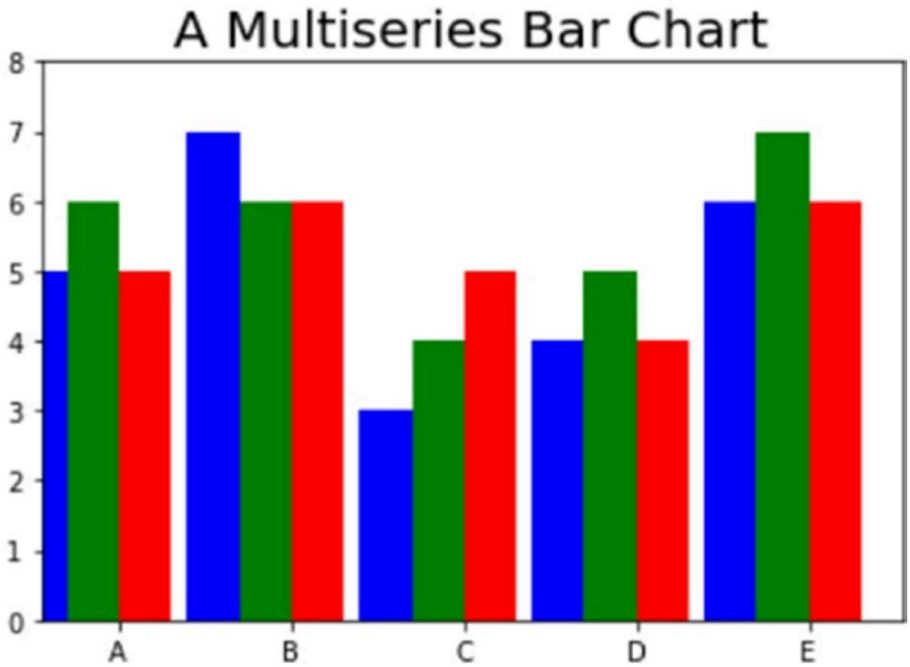


Figure 7-38. *A multiseries bar chart displaying three series*

Regarding the multiseries 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 Multiseries 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'])
```

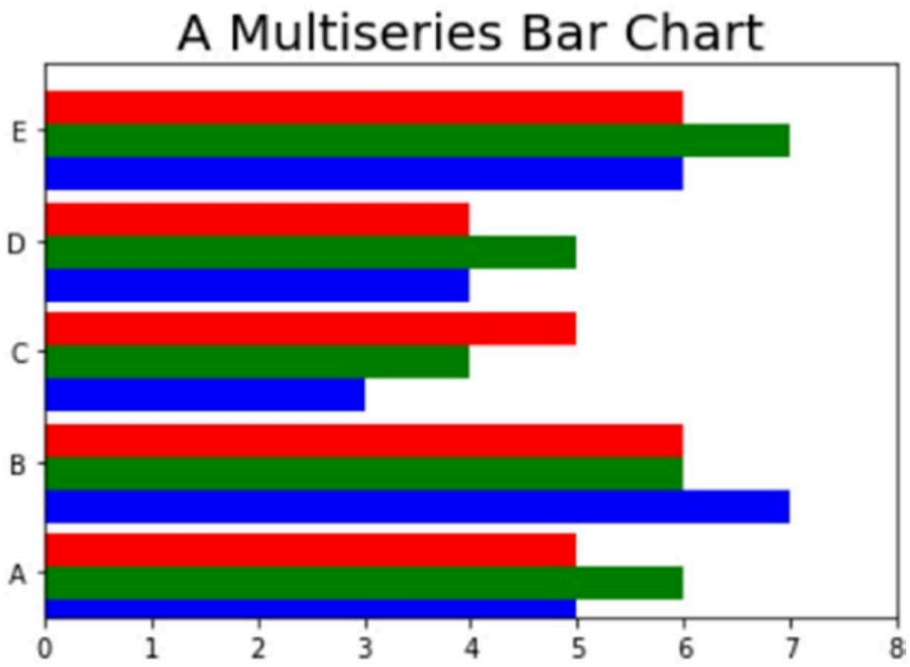


Figure 7-39. *A multiseries horizontal bar chart*

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')
```

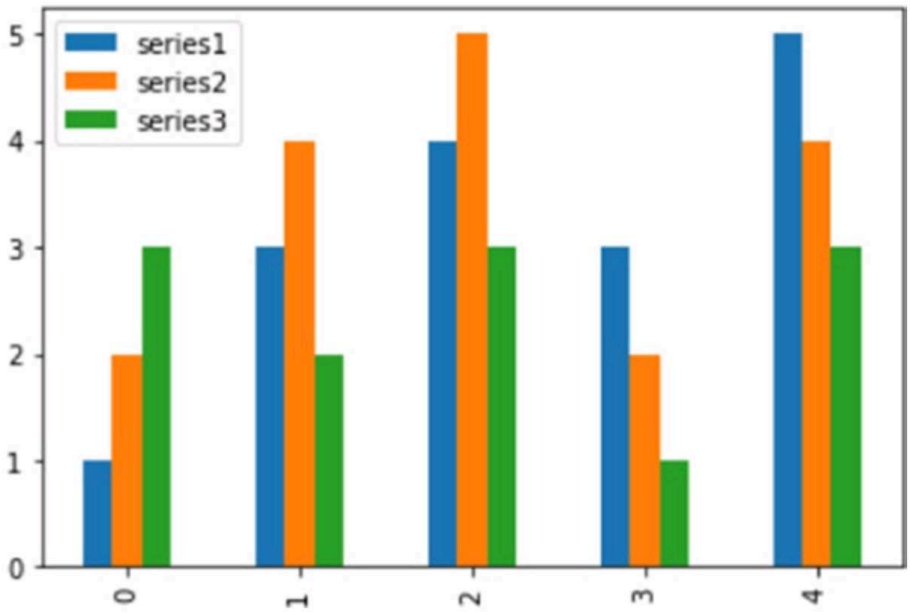


Figure 7-40. *The values in a dataframe can be directly displayed as a bar chart*

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.

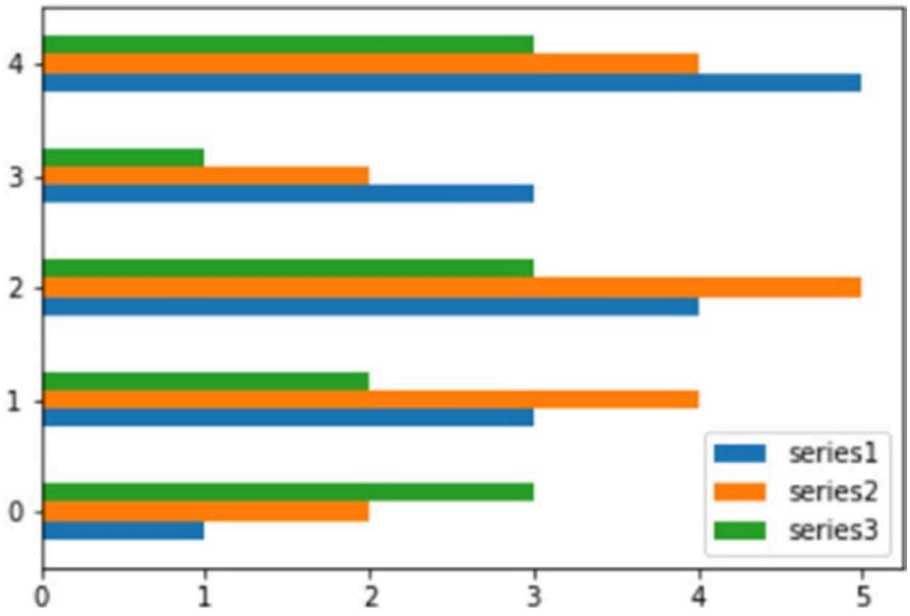


Figure 7-41. *A horizontal bar chart could be a valid alternative to visualize your dataframe values*

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'])
```

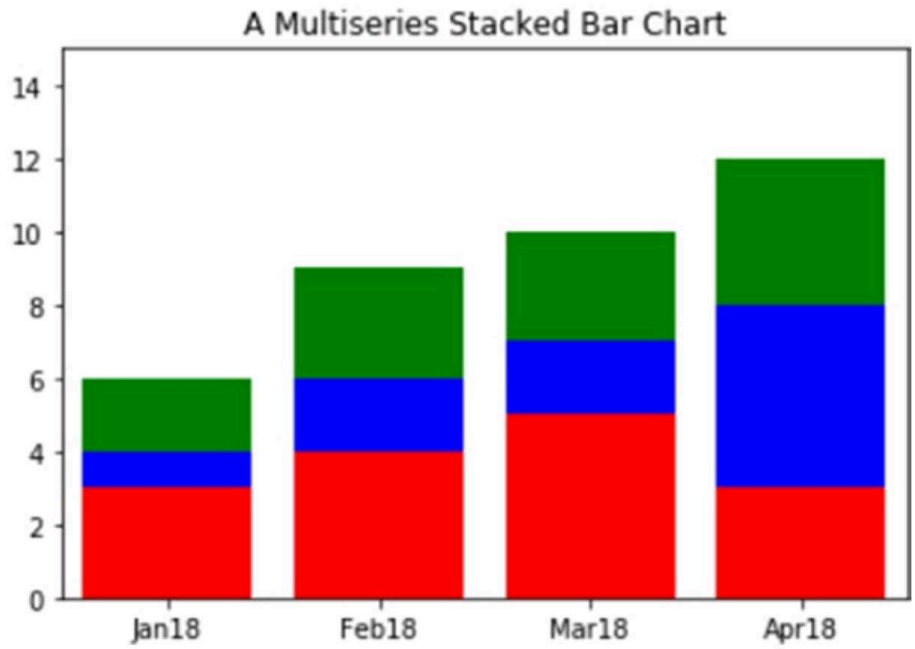


Figure 7-42. *A multiseries stacked bar*

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'])
```

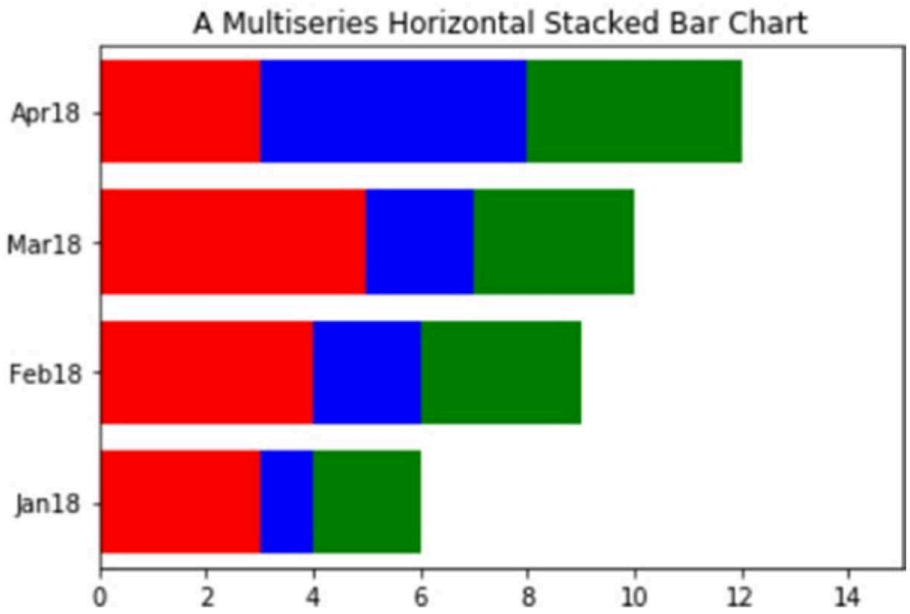


Figure 7-43. *A multiseries horizontal stacked bar chart*

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>)
```

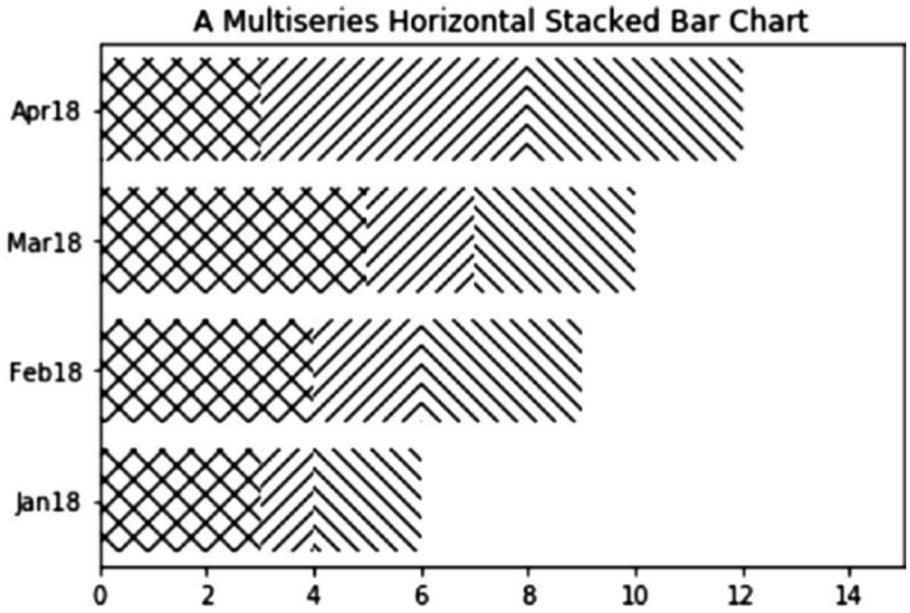


Figure 7-44. *The stacked bars can be distinguished by their hatches*

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>
```

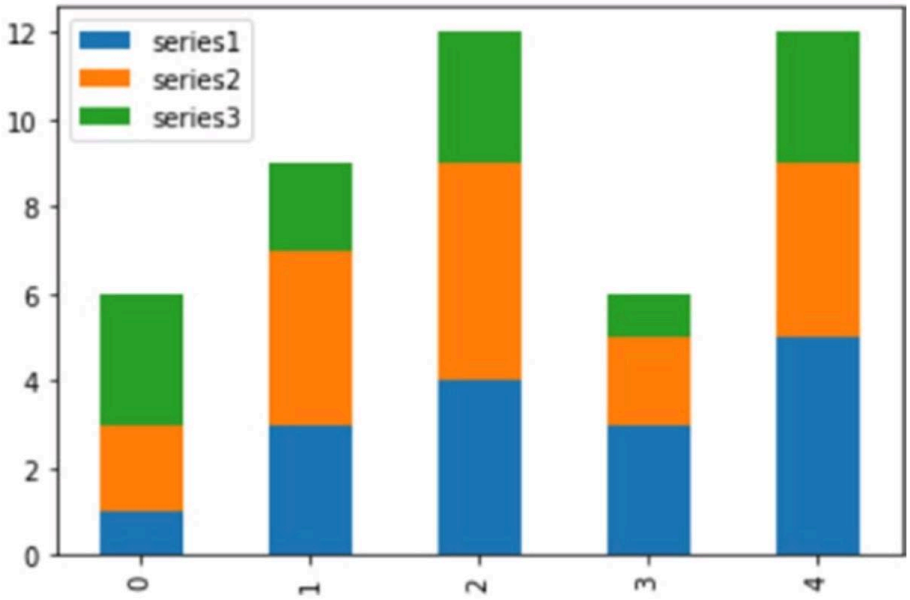


Figure 7-45. *The values of a dataframe can be directly displayed as a stacked bar chart*

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')
```

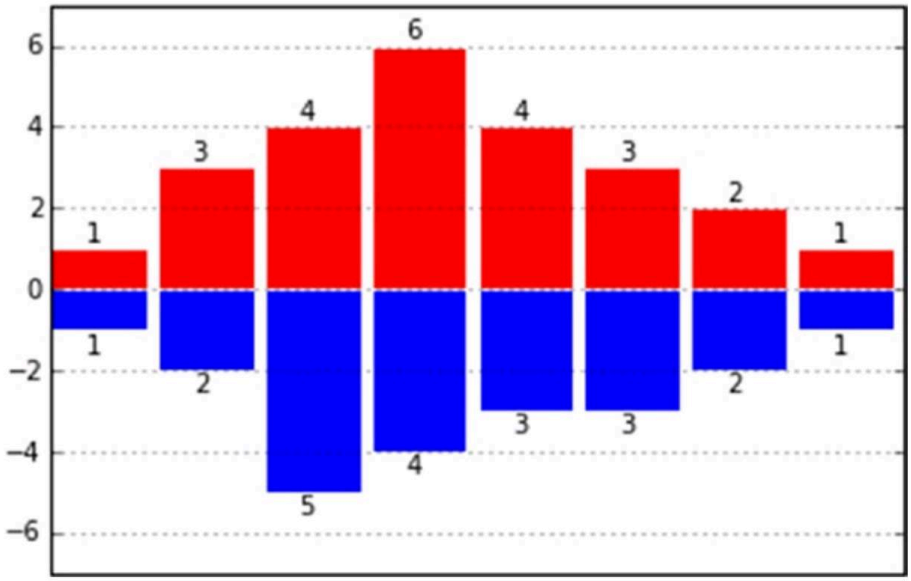


Figure 7-46. *Two series can be compared using this kind of bar chart*

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')
```

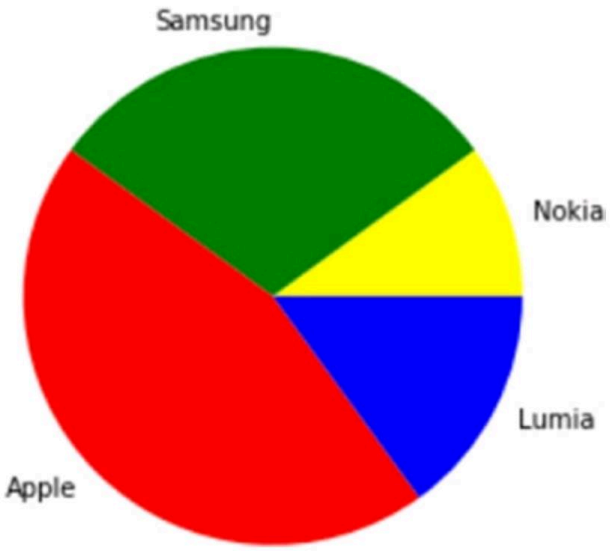


Figure 7-47. *A very simple pie chart*

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')
```

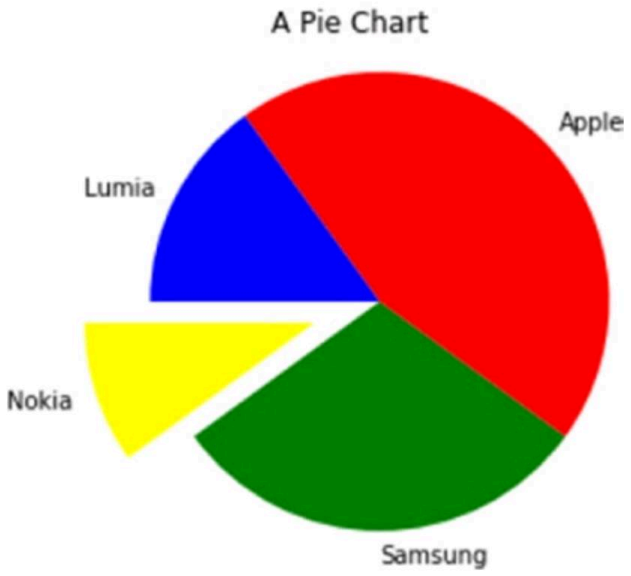


Figure 7-48. *A more advanced pie chart*

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')
```

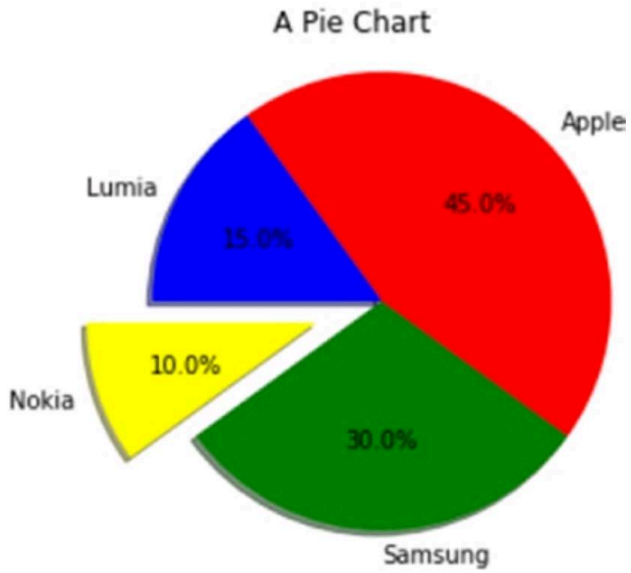


Figure 7-49. *An even more advanced pie chart*

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>
```

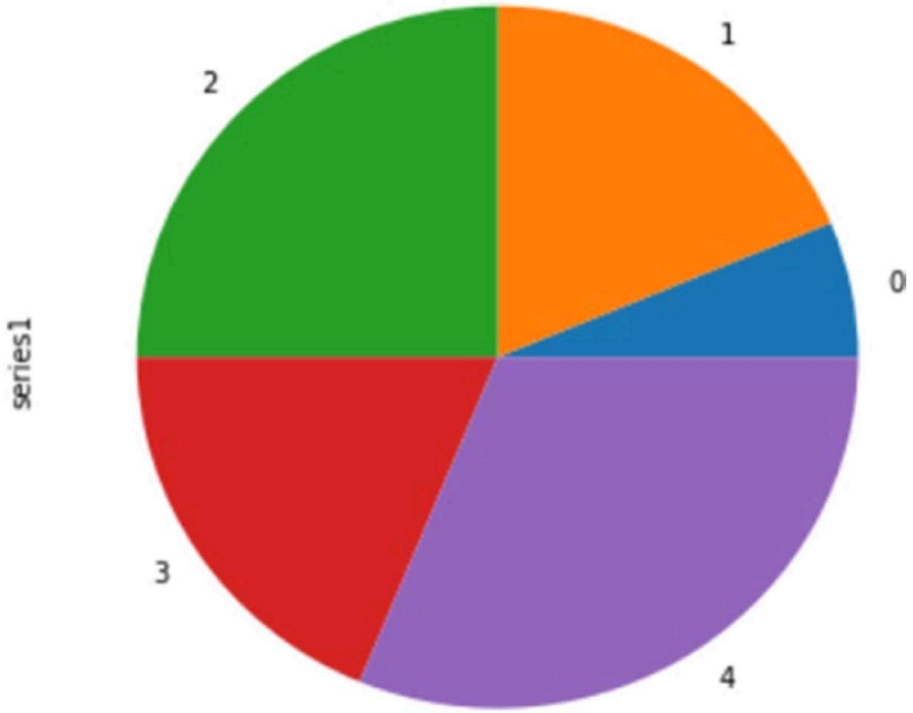


Figure 7-50. *The values in a pandas dataframe can be directly drawn as a pie chart*