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, totally 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 the data analysis.

So the knowledge of the NumPy library is a prerequisite in order to face, in the best way, all scientific Python packages, and particularly, to use and understand more about the pandas library and how to get the most out of it. The pandas library will be the main subject in 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 revise the basic concepts or to regain familiarity with it by running the examples described in this chapter.

NumPy: A Little History

At the dawn of the Python language, the developers began to need to perform numerical calculations, especially when this language began to be considered by the scientific community.

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

From that moment on, NumPy has 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 also comes with a range of functions that allow you to perform operations on arrays in a highly efficient way and to perform high-level mathematical calculations.

Currently, NumPy is open source and licensed under BSD. There are many contributors that with their support 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)

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

On Linux (Fedora)

```
sudo yum install numpy scipy
```

On Windows with Anaconda

```
conda install numpy
```

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

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

Ndarray: The Heart of the Library

The whole 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 within 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 defined their size at the time of creation, it remains unchanged. This behavior is different from Python lists, which can grow or shrink in size.

To define a new ndarray, the easiest way 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, passing the new variable to the **type()** function.

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

In order to know the associated dtype to the just created ndarray, you have to use the **dtype** attribute.

```
>>> a.dtype
dtype('int32')
```

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
(3L,)
```

What you have just seen is the simplest case that is a one-dimensional array. But the use of arrays can be easily extended to the case with 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
(2L, 2L)
```

This array has rank 2, since it has two axis, 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 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 the lists can accept even tuples and sequences of tuples.

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

and also, even sequences of tuples and lists interconnected make no difference.

```
>>> 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 used only numeric values as simple integer and float, 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='|S1')
>>> g.dtype
dtype('S1')
>>> g.dtype.name
'string8'
```

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 is able to associate the most suitable type according to the values contained in the sequence of lists or tuples used. 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 the ndarrays with an initial content, created with some different values depending on the function. Throughout the chapter, but also 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:

```
>>> 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 have created arrays with 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, 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 a 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])
```

But 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 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 the items are defined within it. Now it is the time to see how to apply various operations on them.

Arithmetic Operators

The first operations that you will perform on arrays are the application of arithmetic operators. The most obvious are the sum or the multiplication of an array with 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, namely, that occupy the same position, so that at the end as a result there will be a new array containing the results in the same location of the operands (see Figure 3-1).

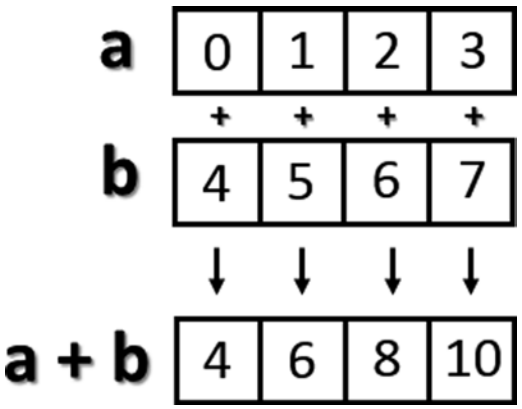

Figure 3-1. *Element-wise addition*

```
>>> 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])
```

Moreover, these operators are also available for functions, provided that the value returned is a NumPy array. For example, you can multiply the array with the sine or the square root of the elements of the 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 **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 between each element of the corresponding row of the first matrix with the corresponding element of the corresponding column of the second matrix. However, Figure 3-2 illustrates the process carried out during the matrix product (only run for two elements).

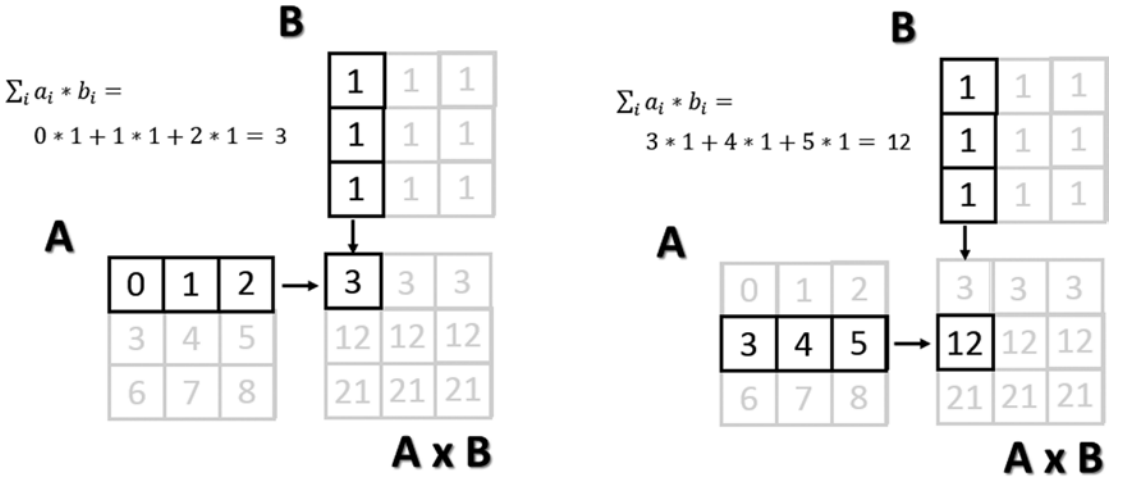

Figure 3-2. *The calculation of matrix elements as result of a matrix product*

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.]])
```

I want to add that since the matrix product is not a commutative operation, then 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 is no such operators in Python, since there are no operators ++ or −−. To increase or decrease the values you have to use operators such as += or −=. These operators are not different from those that we 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, the use of these operators is much more extensive than the simple incremental operators that increase the values by one unit, and then 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 of an array in an element-by-element fashion. This means that it is a function that acts individually on each single element of the input array to generate a corresponding result in a new output array. At the end, you will obtain an array of the same size of the input.

There are many mathematical and trigonometric operations that meet this definition, for example, the calculation of 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 within the library NumPy.

Aggregate Functions

Aggregate functions are those functions that 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 have seen how to create an array and how to perform operations on it. In this section you will see how to manipulate these objects, how to select some elements through indexes and slices, in order to obtain the views of the values contained within them or to make assignments in order to change the value in them. Finally, you will also see how you can make the iterations within them.

Indexing

Array indexing always refers to the use of square brackets ('[ ]') to index the elements of the array so that it 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).

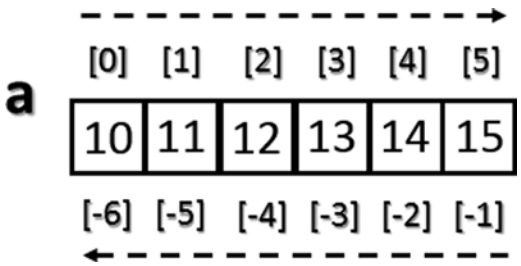

Figure 3-3. *The indexing of an ndarray monodimensional*

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 start to refer the final element to move gradually towards 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 within the 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 to select elements within the matrix you will still use square brackets, but this time the values are two [row index, column index] (Figure 3-4).

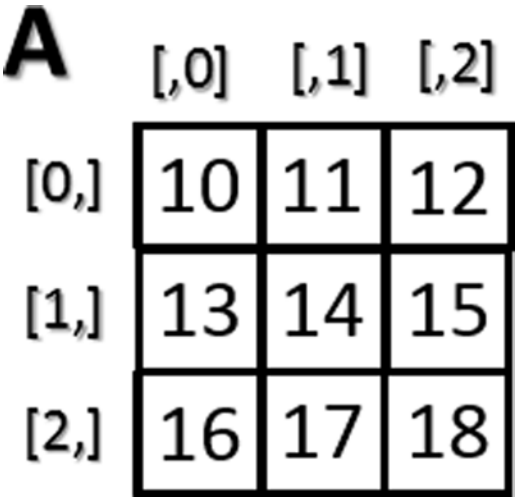

Figure 3-4. *The indexing of a bidimensional array*

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

So 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 is the operation which allows you to extract portions of an array to generate new ones. Whereas using the Python lists the arrays obtained by slicing are copies, in NumPy, arrays are views onto the same underlying buffer.

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

If you want to extract a portion of the array, for example one that goes from the second to the sixth element, then 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 from the previous portion an item, 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 between one element and the next one to take. 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, then implicitly NumPy 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, and then all 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])
```

As regards the case of two-dimensional array, the slicing syntax still applies, but it is separately defined both for the rows and for the 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 any number, then 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 then 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, you will find out that 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 may 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 us 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 to makes that it will be entirely 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 finally the array. If the option **axis** equals 0, then the iteration evaluates the elements column by column, whereas if the **axis** equals 1 then the iteration evaluates the elements row by row. For example, you can calculate the average of the values for the 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 within the NumPy library, but nothing prevents you from defining functions on their own. You also used an aggregate function. However, nothing forbids us from using an ufunc. In this case, using the iteration both by column and by row, the end result is the same. In fact, using a ufunc is how to perform one iteration element-by-element.

```
>>> def foo(x):
...     return x/2
...
>>> np.apply_along_axis(foo, axis=1, arr=A)
array([[5, 5, 6],
       [6, 7, 7],
       [8, 8, 9]])
>>> np.apply_along_axis(foo, axis=0, arr=A)
array([[5, 5, 6],
       [6, 7, 7],
       [8, 8, 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 the indexing and the slicing to select or extract a subset of the array. These methods make use of the indexes in a numerical form. An alternative way to perform the selective extraction of the elements in an array is to use the conditions and Boolean operators.

See this alternative method in detail. For example, suppose you want to select all the values 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, that is, as we said the operator greater, you will receive as a return value Boolean array containing True values in the positions in which the condition is satisfied, 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 have already seen 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 therefore it is useful to 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 possible, that is, to convert a two-dimensional array into a one-dimensional array, through 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 the transposition of a matrix that is an inversion of the columns with 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 together to form a new one that contains all of them. NumPy uses the concept of stacking, providing a number of functions in this regard. For example, you can run the vertical stacking with the **vstack()** function, which combines the second array as new rows of the first array. In this case you have a growth of the array in the 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 that 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

First you saw how to assemble multiple arrays to each other through the operation of stacking. Now you will see the opposite operation, that is, to divide an array into several parts. In NumPy, to do this you will use splitting. 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 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 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. In addition, 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 those of the columns; if instead the option is axis = 0, then they will be the 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, then 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 NumPy library. The difference between copies and views will be illustrated especially when they return values. Also the mechanism of broadcasting that occurs implicitly in many transactions with the functions of NumPy will be covered in this section.

Copies or Views of Objects

As you may have noticed with NumPy, especially when you are performing operations or manipulations on the array, you can have as a return value either a copy or a view of the array. In NumPy, all assignments do not produce copies of arrays, nor any element contained within them.

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

If you assign one array a to another array b, actually you are not doing a copy but b is just another way to call array a. In fact, by changing the value of the third element you change the third value of b too. When you perform the slicing of an array, actually the object returned is only a view of the original array.

```
>>> c = a[0:2]
>>> c
array([1, 2])
>>> a[0] = 0
>>> c
array([0, 2])
```

As you can see, even with the slicing, you are actually pointing to the same object. If you want to generate a complete copy and distinct array you can use the **copy()** function.

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

In this case, even changing the items in array a, array c remains unchanged.

Vectorization

Vectorization is a concept that, along with the broadcasting, is the basis of the internal implementation of NumPy. The vectorization is the absence of explicit loop during the developing of the code. These loops actually cannot be omitted, but are implemented internally and then they are replaced by other constructs in the code. The application of vectorization leads to a more concise and readable code, and you can say that it will appear more "Pythonic" in its appearance. In fact, thanks to the vectorization, many operations take on a more mathematical expression, for example NumPy allows you to express the multiplication of two arrays as shown:

```
a * b
```

or even two matrices:

```
A * B
```

In other languages, such operations would be expressed with many nested loops with the for construct. For example, the first operation would be expressed in the following way:

```
for (i = 0; i < rows; i++){
  c[i] = a[i]*b[i];
}
```

While the product of matrices would be expressed as follows:

```
for( i=0; i < rows; i++){
   for(j=0; j < columns; j++){
      c[i][j] = a[i][j]*b[i][j];
   }
}
```

From all this, it is clear that using NumPy the code is more readable and especially expressed in a more mathematical way.

Broadcasting

Broadcasting is the operation that allows an operator or a function to act on two or more arrays to operate even if these arrays do not have exactly the same shape. Actually, not all the dimensions are compatible with each other in order to be subjected to broadcasting but they must meet certain rules.

You saw that using NumPy, you can classify multidimensional arrays through the shape that is a tuple representing the length of the elements for each dimension.

Thus, two arrays may be subjected to broadcasting when all their dimensions are compatible, i.e., the length of each dimension must be equal between the two arrays or one of them must be equal to 1. If these two conditions are not met, you get an exception given that the two arrays are not compatible.

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

In this case, you obtain two arrays:

```
4 x 4
4
```

The rules of broadcasting are two. The first rule is to add a 1 to each missing dimension. If the compatibility rules are now satisfied you can apply the broadcasting and move to the second rule.

```
4 x 4
4 x 1
```

The rule of compatibility is met. Then you can move on to the second rule of broadcasting. This rule explains how to extend the size of the smallest array so that it takes on the same size of the biggest, so that then the element-wise function or operator is applicable.

The second rule assumes that the missing elements (size, length 1) are filled with replicas of the values contained in extended sizes (see Figure 3-5).

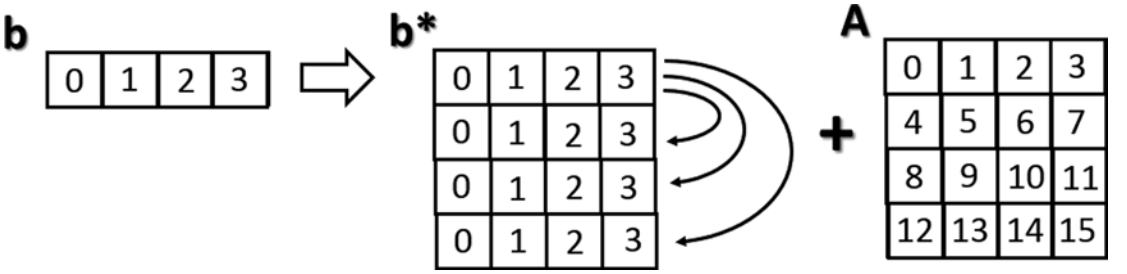

Figure 3-5. *An application of the second broadcasting rule*

Now that the two arrays have the same dimensions, the values inside may be added together.

```
>>> A + b
array([[ 0,  2,  4,  6],
       [ 4,  6,  8, 10],
       [ 8, 10, 12, 14],
       [12, 14, 16, 18]])
```

In this case you are in a simple case in which one of the two arrays is smaller than the other. There may be more complex cases in which the two arrays have different shapes but each of them is smaller than the other only for some dimensions.

```
>>> m = np.arange(6).reshape(3, 1, 2)
>>> n = np.arange(6).reshape(3, 2, 1)
>>> m
array([[[0, 1]],

       [[2, 3]],

       [[4, 5]]])
>>> n
array([[[0],
        [1]],

       [[2],
        [3]],

       [[4],
        [5]]])
```

Even in this case, by analyzing the shapes of the two arrays, you can see that they are compatible and therefore the rules of broadcasting can be applied.

```
3 x 1 x 2
3 x 2 x 1
```

In this case both undergo the extension of dimensions (broadcasting).

```
m* = [[[0,1],            n* = [[[0,0],
       [0,1]],                  [1,1]],
      [[2,3],                 [[2,2],
       [2,3]],                  [3,3]],
      [[4,5],                 [[4,4],
       [4,5]]]                  [5,5]]]
```

And then you can apply, for example, the addition operator between the two arrays, operating element-wise.

```
>>> m + n
array([[[ 0,  1],
        [ 1,  2]],

       [[ 4,  5],
        [ 5,  6]],

       [[ 8,  9],
        [ 9, 10]]])
```

Structured Arrays

So far in the various examples in previous sections, you saw monodimensional and two-dimensional arrays. Actually, NumPy provides the possibility of creating arrays that are much more complex not only in size, but in the structure itself, called precisely **structured arrays**. This type of array contains structs or records instead of the individual items.

For example, you can create a simple array of structs as items. Thanks to the **dtype** option, you can specify a list of comma-separated specifiers to indicate the elements that will constitute the struct, along with its data type and order.

```
bytes                b1
int                  i1, i2, i4, i8
unsigned ints        u1, u2, u4, u8
floats               f2, f4, f8
complex              c8, c16
fixed length strings  a<n>
```

For example, if you want to specify a struct consisting of an integer, a character string of length 6 and a Boolean value, you will specify the three types of data in the dtype option with the right order using the corresponding specifiers.

```
>>> structured = np.array([(1, 'First', 0.5, 1+2j),(2, 'Second', 1.3, 2-2j),
(3, 'Third', 0.8, 1+3j)],dtype=('i2, a6, f4, c8'))
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.8000000011920929, (1+3j))],
      dtype=[('f0', '<i2'), ('f1', 'S6'), ('f2', '<f4'), ('f3', '<c8')])
```

You may also use the data type explicitly specifying int8, uint8, float16, complex64, and so forth.

```
>>> structured = np.array([(1, 'First', 0.5, 1+2j),(2, 'Second', 1.3,2-2j),
(3, 'Third', 0.8, 1+3j)],dtype=('
int16, a6, float32, complex64'))
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.8000000011920929, (1+3j))],
      dtype=[('f0', '<i2'), ('f1', 'S6'), ('f2', '<f4'), ('f3', '<c8')])
```

However, both cases have the same result. Inside the array you see a dtype sequence containing the name of each item of the struct with the corresponding type of data.

Writing the appropriate reference index, you obtain the corresponding row which contains the struct.

```
>>> structured[1]
(2, 'Second', 1.2999999523162842, (2-2j))
```

The names that are assigned automatically to each item of struct can be considered as the names of the columns of the array, and then using them as a structured index, you can refer to all the elements of the same type, or of the same 'column.'

```
>>> structured['f1']
array(['First', 'Second', 'Third'],
      dtype='|S6')
```

As you have just seen, the names are assigned automatically with an **f** (which stands for field) and a progressive integer that indicates the position in the sequence. In fact, it would be more useful to specify the names with something more meaningful. This is possible and you can do it at the time of the declaration of the array:

```
>>> structured = np.array([(1,'First',0.5,1+2j),(2,'Second',1.3,2-2j),(3,'Third',0.8,1+3j)],
dtype=[('id','i2'),('position','a6'),('value','f4'),('complex','c8')])
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.800000011920929, (1+3j))],
      dtype=[('id', '<i2'), ('position', 'S6'), ('value', '<f4'), ('complex', '<c8')])
```

or at a later time, redefining the tuples of names assigned to the dtype attribute of the structured array.

```
>>> structured.dtype.names = ('id','order','value','complex')
```

Now you can use meaningful names for the various types of fields:

```
>>> structured['order']
array(['First', 'Second', 'Third'],
      dtype='|S6')
```

Reading and Writing Array Data on Files

A very important aspect of NumPy that has not been taken into account yet is the reading of the data contained within a file. This procedure is very useful, especially when you have to deal with large amounts of data collected within arrays. This is a very common operation in data analysis, since the size of the dataset to be analyzed is almost always huge, and therefore it is not advisable or even possible to manage the transcription and subsequent reading of data by a computer to another manually, or from one session of the calculation to another.

Indeed NumPy provides in this regard a set of functions that allow the data analyst to save the results of his calculations in a text or binary file. Similarly, NumPy allows reading and conversion of written data within a file into an array.

Loading and Saving Data in Binary Files

Regarding for saving and then later retrieving data stored in binary format, NumPy provides a pair of functions called **save()** and **load()**.

Once you have an array to save, for example, containing the results of your processing during data analysis, you simply call the save() function, specifying as arguments the name of the file, to which **.npy** extension will be automatically added, and then the array itself.

```
>>> data
array([[ 0.86466285,  0.76943895,  0.22678279],
       [ 0.12452825,  0.54751384,  0.06499123],
       [ 0.06216566,  0.85045125,  0.92093862],
       [ 0.58401239,  0.93455057,  0.28972379]])
>>> np.save('saved_data',data)
```

But when you need to recover the data stored within a .npy file, you can use the load() function by specifying the file name as argument, this time adding the extension .npy.

```
>>> loaded_data = np.load('saved_data.npy')
>>> loaded_data
array([[ 0.86466285,  0.76943895,  0.22678279],
       [ 0.12452825,  0.54751384,  0.06499123],
       [ 0.06216566,  0.85045125,  0.92093862],
       [ 0.58401239,  0.93455057,  0.28972379]])
```

Reading File with Tabular Data

Many times, the data that you want to read or save are in textural format (TXT or CSV, for example). Generally, you decide to save the data in this format, instead of binary, because the files can be accessed outside independently if you are working with NumPy or with any other application. Take for example the case of a set of data in CSV (Comma-Separated Values) format, in which data are collected in a tabular form and where all values are separated from each other by commas (see Listing 3-1).

Listing 3-1. data.csv

```
id,value1,value2,value3
1,123,1.4,23
2,110,0.5,18
3,164,2.1,19
```

To be able to read your data in a text file and insert them into an array, NumPy provides a function called **genfromtxt()**. Normally, this function takes three arguments, including the name of the file containing the data, the character that separates a value from another which in our case is a paragraph, and whether it contains the column headers.

```
>>> data = np.genfromtxt('data.csv', delimiter=',', names=True)
>>> data
array([(1.0, 123.0, 1.4, 23.0), (2.0, 110.0, 0.5, 18.0),
       (3.0, 164.0, 2.1, 19.0)],
      dtype=[('id', '<f8'), ('value1', '<f8'), ('value2', '<f8'), ('value3', '<f8')])
```

As we can see from the result, you get a structured array in which the column headings have become the names of the field.

This function performs implicitly two loops: the first reads a line at a time, and the second separates and converts the values contained in it, inserting the consecutive elements created specifically. One positive aspect of this feature is that if within the file there are some missing data, the function is able to handle them.

Take for example the previous file (see Listing 3-2) and remove some items. Save as **data2.csv**.

Listing 3-2. data2.csv

```
id,value1,value2,value3
1,123,1.4,23
2,110,,18
3,,2.1,19
```

Launching these commands, you can see how the genfromtxt() function replaces the blanks in the file with **nan** values.

```
>>> data2 = np.genfromtxt('data2.csv', delimiter=',', names=True)
>>> data2
array([(1.0, 123.0, 1.4, 23.0), (2.0, 110.0, nan, 18.0),
       (3.0, nan, 2.1, 19.0)],
      dtype=[('id', '<f8'), ('value1', '<f8'), ('value2', '<f8'), ('value3', '<f8')])
```

At the bottom of the array, you can find the column headings contained in the file. These headers can be considered as labels which act as indexes to extract the values by column:

```
>>> data2['id']
array([ 1.,  2.,  3.])
```

Instead, using the numerical indexes in the classic way you will extract data corresponding to the rows.

```
>>> data2[0]
(1.0, 123.0, 1.4, 23.0)
```

Conclusions

In this chapter, you saw all the main aspects of the NumPy library and through a series of examples you got familiar with a range of features that form the basis of many other aspects you'll face in the course of the book. In fact, many of these concepts will be taken from other scientific and computing libraries that are more specialized, but that have been structured and developed on the basis of this library.

You saw how thanks to the ndarray you can extend the functionalities of Python, making it a suitable language for scientific computing and in a particular way for data analysis.

Knowledge of NumPy proves therefore to be crucial for anyone who wants to take on the world of the data analysis.

In the next chapter, we will begin to introduce a new library, pandas, that being structured on NumPy will encompass all the basic concepts illustrated in this chapter, but extending them to make them more suitable for data analysis.