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])
```

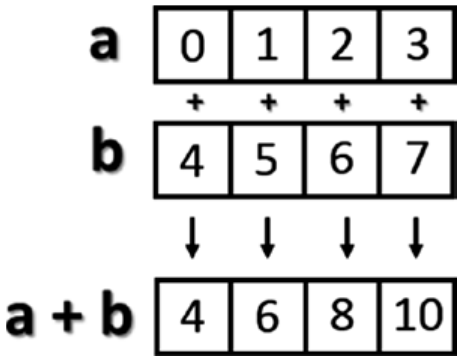

Figure 3-1. *Element-wise addition*

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).

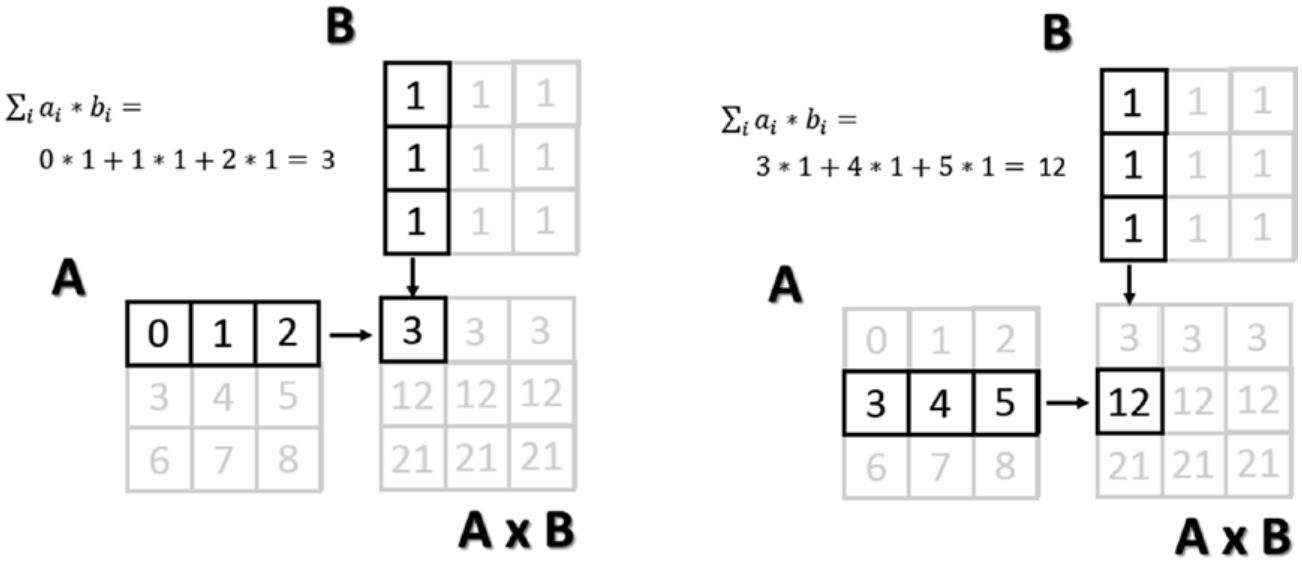

Figure 3-2. *Calculating matrix elements as a result of a matrix product*

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).

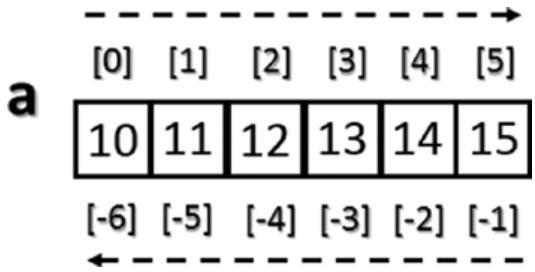

Figure 3-3. *Indexing a monodimensional ndarray*

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).

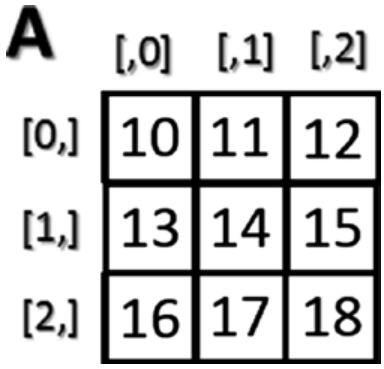

Figure 3-4. *Indexing a bidimensional array*

```
>>> 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])
```