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, is 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 can view 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.

Currently, NumPy is open source and licensed under BSD. There are many contributors who have expanded the potential of this library. At present, NumPy has arrived at release 1.24. As you can see in Figure 3-1, this library is in continuous development, with approximately one release every six months.

© Fabio Nelli 2023
F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-9532-8_3

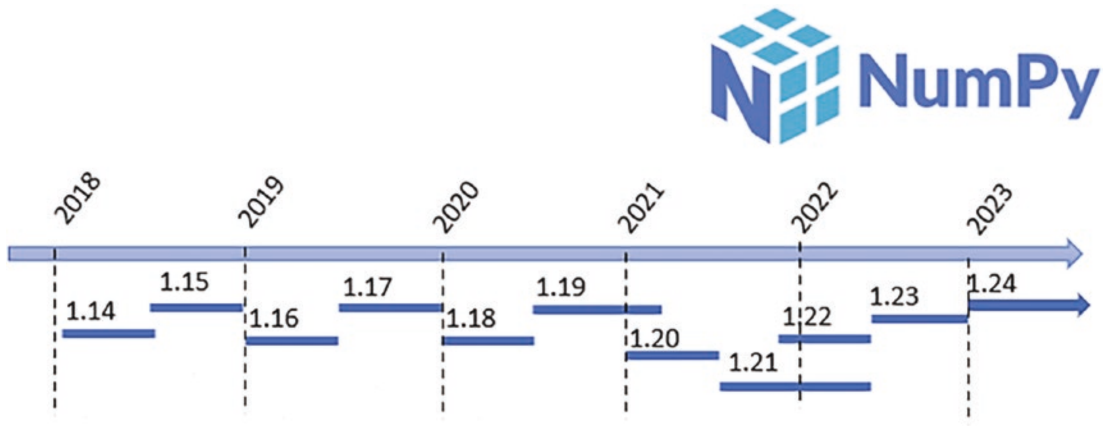

Figure 3-1. *NumPy releases in the last five years, with the new NumPy logo [CC BY-SA 4.0 Isabela Presedo-Floyd]*

The NumPy Installation

This library doesn't have any requirements except that you have a Python platform to run on, so installing it shouldn't cause any problems. Generally, this module is present as a basic package in most Python distributions; however, if not, you can install it later.

However, regardless of the platform you are using, as with all the other libraries that you will use throughout the book, it is also recommended for NumPy to use the platform of the Anaconda distribution. This allows you to cleanly manage the NumPy installation aspect, as well as easily create and manage the virtual environments on which to install it. In this way, it is possible to test and develop your code with different Python versions and NumPy releases without having to uninstall and reinstall everything each time.

If you have Anaconda, just write the following to install NumPy:

```
conda install numpy
```

If instead you want to work without the support of this distribution, use the command-line `pip` command to install the NumPy library (see `https://pypi.org/project/numpy/`):

```
pip install numpy
```

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

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

If, on the other hand, you are writing code, in order to access NumPy and its functions, you have to insert this instruction at the beginning of the Python code.

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)
<class '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('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 determine 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
array([[1.2, 2.4],
       [0.3, 3. ]])
>>> 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, because to access the data in the array you use the indexing mechanism, which you will see in the next sections.

```
>>> b.itemsize
8
>>> b.data
<memory at 0x000001A8AD526A80>
```

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_`	Signed integer type (same as C `long` and Python `int`; normally either `int64` or `int32` depending on the platform)
`intc`	Signed integer type, identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `size_t`; normally either `int32` or `int64`)
`int8`	Alias for the signed integer type with 8 bits (–128 to 127)
`int16`	Alias for the signed integer type with 16 bits (–32768 to 32767)
`int32`	Alias for the signed integer type with 32 bits (–2147483648 to 2147483647)
`int64`	Alias for the signed integer type with 64 bits (–9223372036854775808 to 9223372036854775807)
`uint8`	Alias for the unsigned integer type with 8 bits (0 to 255)
`uint16`	Alias for the unsigned integer type with 16 bits (0 to 65535)
`uint32`	Alias for the unsigned integer type with 32 bits (0 to 4294967295)
`uint64`	Alias for the unsigned integer type with 64 bits (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 an 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 of the 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 is 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—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 0 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 you 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 as 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 these arrays.

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 objects occupy the same position, so that the end result is a new array containing the results in the same location of the operands (see Figure 3-2).

```
>>> 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])
```

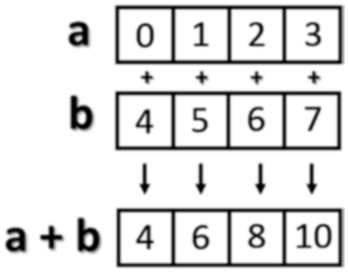

Figure 3-2. *Element-wise addition*

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 3-3 illustrates the process carried out during the matrix product (run for two elements).

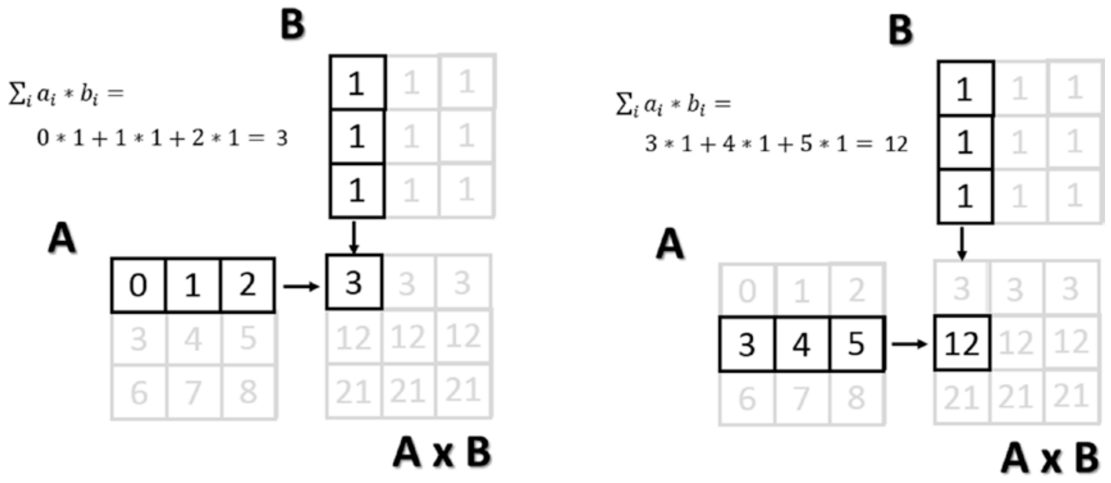

Figure 3-3. *Calculating matrix elements as a result of a matrix product*

An alternative way to write the matrix product is to use 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 because 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, because 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 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 array.

```
>>> 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 common math functions are already implemented in the NumPy library.

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 in the ndarray class and so can be invoked directly from the array on which you want to perform the calculation.

```
>>> a = np.array([3.3, 4.5, 1.2, 5.7, 0.3])
>>> a.sum()
15.0
>>> a.min()
0.3
>>> a.max()
5.7
>>> 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 see how to manipulate these objects. You 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 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 3-4).

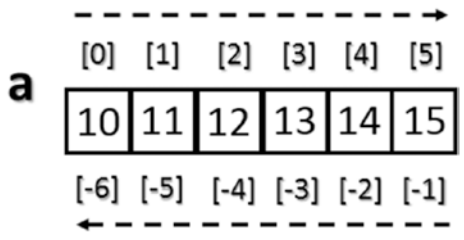

Figure 3-4. *Indexing a monodimensional ndarray*

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 is the one with the more negative index value.

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

To select multiple items at once, you can pass an 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 still use square brackets, but this time there are two values [row index, column index] (see Figure 3-5).

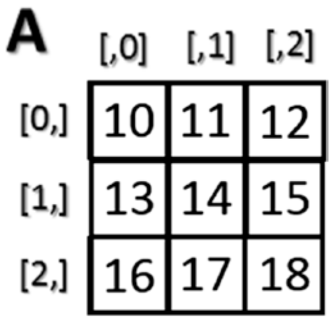

Figure 3-5. *Indexing a two-dimensional array*

```
>>> 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 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 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 colon (`:`).

```
>>> 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, iterating 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, 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 for 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 axis option 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.])
```

The previous case uses 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 can 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, when creating a two-dimensional array, that 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 that changes shape and no object is returned. The inverse operation is also possible; that is, you can convert a two-dimensional array into a one-dimensional array. You do this by using the `ravel()` function.

```
>>> a = a.ravel()
>>> 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])
```

Or you can even act directly on the shape attribute of the array itself.

```
>>>  a.shape = (A.size)
>>> 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 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 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, 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 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 also have to specify the indexes of the parts to be divided. If you use the `axis = 1` option, 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.

Copies or Views of Objects

As you may have noticed with NumPy, especially when you are manipulating an array, you can return a copy or a view of the array. None of the NumPy assignments produces copies of arrays, nor any element contained in 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, you are not copying it; array b is just another way to call array a. In fact, by changing the value of the third element of a, you change the third value of b too. When you slice an array, the object returned is 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 when slicing, you are actually pointing to the same object. If you want to generate a complete and distinct array, 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 when you change the items in array a, array c remains unchanged.

Vectorization

Vectorization, along with *broadcasting*, is the basis of the internal implementation of NumPy. Vectorization is the absence of an explicit loop during the development of the code. These loops actually cannot be omitted, but are implemented internally and then are replaced by other constructs in the code. The application of vectorization leads to 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 and 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];
   }
}
```

You can see that using NumPy makes the code more readable and more mathematical.

Broadcasting

Broadcasting allows an operator or a function to act on two or more arrays even if these arrays do not have the same shape. That said, not all the dimensions can be subjected to broadcasting; they must meet certain rules.

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

Two arrays can be subjected to broadcasting when all their dimensions are compatible, i.e., the length of each dimension must be equal or one of them must be equal to 1. If neither of these conditions is met, you get an exception that states 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
```

There are two rules of broadcasting. First you must add a 1 to each missing dimension. If the compatibility rules are now satisfied, you can apply broadcasting and move to the second rule. For example:

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

The rule of compatibility is met. Then you can move to the second rule of broadcasting. This rule explains how to extend the size of the smallest array so that it's the size of the biggest array, so that 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 3-6).

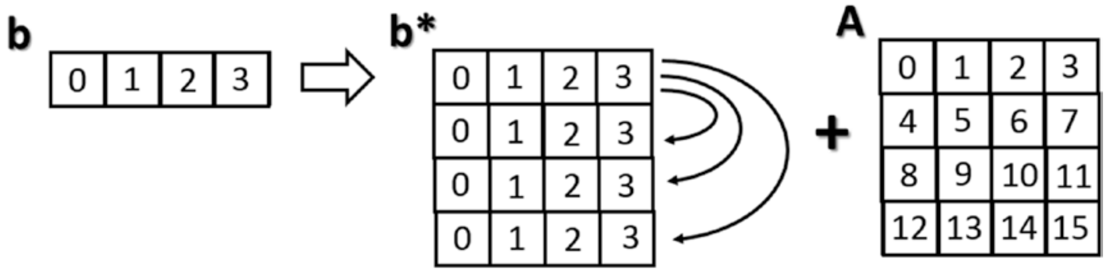

Figure 3-6. *Applying the second broadcasting rule*

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]])
```

This is 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 and each is smaller than the other only in certain 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 arrays 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]]]
```

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 the previous sections, you saw monodimensional and two-dimensional arrays. NumPy allows you to create arrays that are much more complex not only in size, but in the structure, called *structured arrays.* This type of array contains *structs* or records instead of 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 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 specify the three types of data in the dtype option with the right order using the corresponding specifiers.

■ **Note** The result of dtype and other format attributes can vary among different operating systems and Python distributions.

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

You can also use the data type explicitly by 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, b'First', 0.5, 1.+2.j),
       (2, b'Second', 1.3, 2.-2.j),
       (3, b'Third', 0.8, 1.+3.j)],
      dtype=[('f0', '<i2'), ('f1', 'S6'), ('f2', '<f4'), ('f3', '<c8')])
```

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 data type.

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

```
>>> structured[1]
(2, b'Second', 1.3, 2.-2.j)
```

The names that are assigned automatically to each item of the struct can be considered the names of the columns of the array. 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([b'First', b'Second', b'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 array declaration:

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

Or you can do it 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 field types:

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

Reading and Writing Array Data on Files

A very important aspect of NumPy that has not been discussed yet is the process of reading data contained in a file. This procedure is very useful, especially when you have to deal with large amounts of data collected in arrays. This is a very common data analysis operation, since the size of the dataset to be analyzed is almost always huge, and therefore it is not advisable or even possible to manage it manually.

NumPy provides a set of functions that allow data analysts to save the results of their calculations in a text or binary file. Similarly, NumPy allows you to read and convert written data in a file into an array.

Loading and Saving Data in Binary Files

NumPy provides a pair of functions, called `save()` and `load()`, that enable you to save and then later retrieve data stored in binary format.

Once you have an array to save, for example, one that contains the results of your data analysis processing, you simply call the `save()` function and specify as arguments the name of the file and the array. The file will automatically be given the `.npy` extension.

```
>>> data = np.random.random((3,3))
>>> data
array([[0.47941017, 0.43759768, 0.76636206],
       [0.51928993, 0.06358527, 0.72109914],
       [0.64501488, 0.94113659, 0.42052306]])
>>> np.save('saved_data',data)
```

When you need to recover the data stored in a `.npy` file, you use the `load()` function by specifying the file name as the argument, this time adding the `.npy` extension.

```
>>> loaded_data = np.load('saved_data.npy')
>>> loaded_data
array([[0.47941017, 0.43759768, 0.76636206],
       [0.51928993, 0.06358527, 0.72109914],
       [0.64501488, 0.94113659, 0.42052306]])
```

Reading Files with Tabular Data

Many times, the data that you want to read or save are in textural format (TXT or CSV, for example). You might save the data in this format, instead of binary, because the files can then 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 the CSV (Comma-Separated Values) format, in which data are collected in a tabular format and the values are separated by commas (see Listing 3-1).

Listing 3-1. ch3_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 values into an array, NumPy provides a function called genfromtxt(). Normally, this function takes three arguments—the name of the file containing the data, the character that separates the values from each other (in this case, a comma), and whether the data contain column headers.

```
>>> data = np.genfromtxt('ch3_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 you can see from the result, you get a structured array in which the column headings have become the field names.

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

Take for example the previous file (see Listing 3-2) with some items removed. Save it as data2.csv.

Listing 3-2. ch3_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('ch3_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 labels that act as indexes to extract the values by column.

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

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

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

Conclusions

In this chapter, you learned about all the main aspects of the NumPy library and became 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 are 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 ndarray, you can extend the functionalities of Python, making it a suitable language for scientific computing and data analysis.

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

The next chapter introduces a new library, called pandas, which is structured on NumPy and so encompasses all the basic concepts illustrated in this chapter. However, pandas extends these concepts so they are more suitable to data analysis.