CHAPTER

2

Extending Python Using NumPy

What Is NumPy?

In Python, you usually use the `list` data type to store a collection of items. The Python list is similar to the concept of arrays in languages like Java, C#, and JavaScript. The following code snippet shows a Python list:

```
list1 = [1,2,3,4,5]
```

Unlike arrays, a Python list does not need to contain elements of the same type. The following example is a perfectly legal list in Python:

```
list2 = [1,"Hello",3.14,True,5]
```

While this unique feature in Python provides flexibility when handling multiple types in a list, it has its disadvantages when processing large amounts of data (as is typical in machine learning and data science projects). The key problem with Python's `list` data type is its efficiency. To allow a list to have non-uniform type items, each item in the list is stored in a memory location, with the list containing an "array" of pointers to each of these locations. A Python list requires the following:

- At least 4 bytes per pointer.
- At least 16 bytes for the smallest Python object—4 bytes for a pointer, 4 bytes for the reference count, 4 bytes for the value. All of these together round up to 16 bytes.

Python® Machine Learning, First Edition. Wei-Meng Lee.

Due to the way that a Python list is implemented, accessing items in a large list is computationally expensive. To solve this limitation with Python's list feature, Python programmers turn to *NumPy*, an extension to the Python programming language that adds support for large, multidimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

In NumPy, an array is of type `ndarray` (n-dimensional array), and all elements must be of the same type. An `ndarray` object represents a multidimensional, homogeneous array of fixed-size items, and it is much more efficient than Python's list. The `ndarray` object also provides functions that operate on an entire array at once.

Creating NumPy Arrays

Before using NumPy, you first need to import the NumPy package (you may use its conventional alias *np* if you prefer):

```
import numpy as np
```

The first way to make NumPy arrays is to create them intrinsically, using the functions built right into NumPy. First, you can use the `arange()` function to create an evenly spaced array with a given interval:

```
a1 = np.arange(10)        # creates a range from 0 to 9
print(a1)                 # [0 1 2 3 4 5 6 7 8 9]
print(a1.shape)           # (10,)
```

The preceding statement creates a rank 1 array (one-dimensional) of ten elements. To get the shape of the array, use the `shape` property. Think of *a1* as a 10×1 matrix.

You can also specify a step in the `arange()` function. The following code snippet inserts a step value of 2:

```
a2 = np.arange(0,10,2)    # creates a range from 0 to 9, step 2
print(a2)                 # [0 2 4 6 8]
```

To create an array of a specific size filled with 0s, use the `zeros()` function:

```
a3 = np.zeros(5)          # create an array with all 0s
print(a3)                 # [ 0.  0.  0.  0.  0.]
print(a3.shape)           # (5,)
```

You can also create two-dimensional arrays using the `zeros()` function:

```
a4 = np.zeros((2,3))      # array of rank 2 with all 0s; 2 rows and 3
                          # columns
print(a4.shape)           # (2,3)
```

```
print(a4)
'''
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
'''
```

If you want an array filled with a specific number instead of 0, use the `full()` function:

```
a5 = np.full((2,3), 8)    # array of rank 2 with all 8s
print(a5)
'''
[[8 8 8]
 [8 8 8]]
'''
```

Sometimes, you need to create an array that mirrors an identity matrix. In NumPy, you can do so using the `eye()` function:

```
a6 = np.eye(4)            # 4x4 identity matrix
print(a6)
'''
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
'''
```

The `eye()` function returns a 2-D array with ones on the diagonal and zeros elsewhere.

To create an array filled with random numbers, you can use the `random()` function from the `numpy.random` module:

```
a7 = np.random.random((2,4)) # rank 2 array (2 rows 4 columns) with
                             # random values
                             # in the half-open interval [0.0, 1.0)
print(a7)
'''
[[ 0.48255806  0.23928884  0.99861279  0.4624779 ]
 [ 0.18721584  0.71287041  0.84619432  0.65990083]]
'''
```

Another way to create a NumPy array is to create it from a Python list as follows:

```
list1 = [1,2,3,4,5]  # list1 is a list in Python
r1 = np.array(list1) # rank 1 array
print(r1)            # [1 2 3 4 5]
```

The array created in this example is a rank 1 array.

Array Indexing

Accessing elements in the array is similar to accessing elements in a Python list:

```
print(r1[0])          # 1
print(r1[1])          # 2
```

The following code snippet creates another array named *r2*, which is two-dimensional:

```
list2 = [6,7,8,9,0]
r2 = np.array([list1,list2])     # rank 2 array
print(r2)
'''
[[1 2 3 4 5]
 [6 7 8 9 0]]
'''
print(r2.shape)              # (2,5) - 2 rows and 5 columns
print(r2[0,0])               # 1
print(r2[0,1])               # 2
print(r2[1,0])               # 6
```

Here, *r2* is a rank 2 array, with two rows and five columns.

Besides using an index to access elements in an array, you can also use a list as the index as follows:

```
list1 = [1,2,3,4,5]
r1 = np.array(list1)
print(r1[[2,4]])    # [3 5]
```

Boolean Indexing

In addition to using indexing to access elements in an array, there is another very cool way to access elements in a NumPy array. Consider the following:

```
print(r1>2)     # [False False  True  True  True]
```

This statement prints out a list containing Boolean values. What it actually does is to go through each element in *r1* and check if each element is more than two. The result is a Boolean value, and a list of Boolean values is created at the end of the process. You can feed the list results back into the array as the index:

```
print(r1[r1>2])    # [3 4 5]
```

This method of accessing elements in an array is known as *Boolean Indexing*. This method is very useful. Consider the following example:

```
nums = np.arange(20)
print(nums)        # [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
17 18 19]
```

If you want to retrieve all of the odd numbers from the list, you could simply use Boolean Indexing as follows:

```
odd_num = nums[nums % 2 == 1]
print(odd_num)     # [ 1  3  5  7  9 11 13 15 17 19]
```

Slicing Arrays

Slicing in NumPy arrays is similar to how it works with a Python list. Consider the following example:

```
a = np.array([[1,2,3,4,5],
              [4,5,6,7,8],
              [9,8,7,6,5]])    # rank 2 array
print(a)
'''
[[1 2 3 4 5]
 [4 5 6 7 8]
 [9 8 7 6 5]]
'''
```

To extract the last two rows and first two columns, you can use slicing:

```
b1 = a[1:3, :3]     # row 1 to 3 (not inclusive) and first 3 columns
print(b1)
```

The preceding code snippet will print out the following:

```
[[4 5 6]
 [9 8 7]]
```

Let's dissect this code. Slicing has the following syntax: `[start:stop]`. For two-dimensional arrays, the slicing syntax becomes `[start:stop, start:stop]`. The `start:stop` before the comma (,) refers to the rows, and the `start:stop` after the comma (,) refers to the columns. Hence for `[1:3, :3]`, this means that you want to extract the rows with index 1 right up to 3 (but not including 3), and columns starting from the first column right up to index 3 (but not including 3). The general confusion regarding slicing is the end index. You need to remember that the end index is not included in the answer. A better way to visualize slicing is to write the index of each row and column between the numbers, instead of at the center of the number, as shown in Figure 2.1.

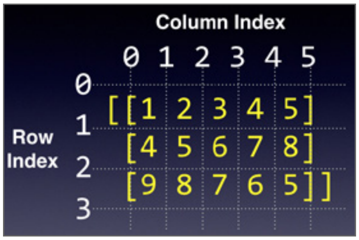

Figure 2.1: Writing the index for row and column in between the numbers

Using this approach, it is now much easier to visualize how slicing works (see Figure 2.2).

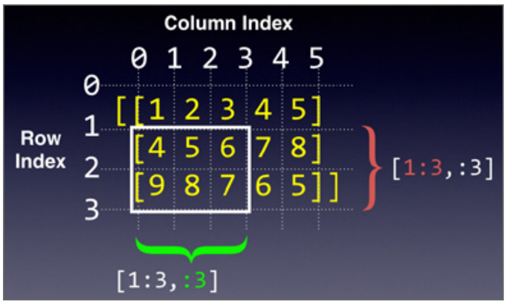

Figure 2.2: Performing slicing using the new approach

What about negative indices? For example, consider the following:

```
b2 = a[-2:,-2:]
print(b2)
```

Using the method just described, you can now write the negative row and column indices, as shown in Figure 2.3.

You should now be able to derive the answer quite easily, which is as follows:

```
[[7 8]
 [6 5]]
```

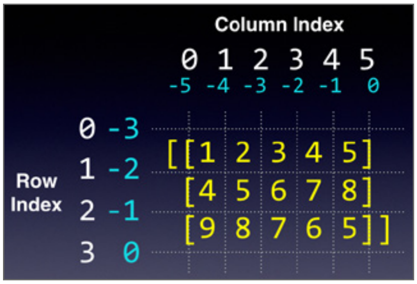

Figure 2.3: Writing the negative indices for rows and columns

NumPy Slice Is a Reference

It is noteworthy that the result of a NumPy slice is a reference and not a copy of the original array. Consider the following:

```
b3 = a[1:, 2:]       # row 1 onwards and column 2 onwards
                     # b3 is now pointing to a subset of a
print(b3)
```

The result is as follows:

```
[[6 7 8]
 [7 6 5]]
```

Here, *b3* is actually a reference to the original array *a* (see Figure 2.4).

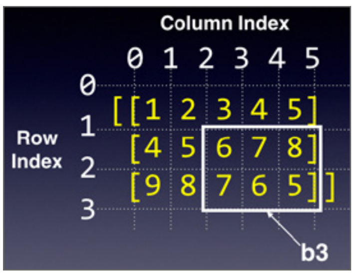

Figure 2.4: Slicing returns a reference to the original array and not a copy

Hence, if you were to change one of the elements in *b3* as follows:

```
b3[0,2] = 88          # b3[0,2] is pointing to a[1,4]; modifying it will
                      # modify the original array
print(a)
```

The result will affect the content of *a* like this:

```
[[ 1  2  3  4  5]
 [ 4  5  6  7 88]
 [ 9  8  7  6  5]]
```

Another salient point to note is that the result of the slicing is dependent on how you slice it. Here is an example:

```
b4 = a[2:, :]       # row 2 onwards and all columns
print(b4)
print(b4.shape)
```

In the preceding statement, you are getting rows with index 2 and above and all of the columns. The result is a rank 2 array, like this:

```
[[9 8 7 6 5]]
(1,5)
```

If you have the following instead . . .

```
b5 = a[2, :]         # row 2 and all columns
print(b5)            # b5 is rank 1
```

. . . then the result would be a rank 1 array:

```
[9 8 7 6 5]
```

Printing the shape of the array confirms this:

```
print(b5.shape)      # (5,)
```

Reshaping Arrays

You can reshape an array to another dimension using the `reshape()` function. Using the *b5* (which is a rank 1 array) example, you can reshape it to a rank 2 array as follows:

```
b5 = b5.reshape(1,-1)
print(b5)
'''
[[9 8 7 6 5]]
'''
```

In this example, you call the `reshape()` function with two arguments. The first `1` indicates that you want to convert it into rank 2 array with 1 row, and the

`-1` indicates that you will leave it to the `reshape()` function to create the correct number of columns. Of course, in this example, it is clear that after reshaping there will be five columns, so you can call the `reshape()` function as `reshape(1,5)`. In more complex cases, however, it is always convenient to be able to use `-1` to let the function decide on the number of rows or columns to create.

Here is another example of how to reshape *b4* (which is a rank 2 array) to rank 1:

```
b4.reshape(-1,)
'''
[9 8 7 6 5]
'''
```

The `-1` indicates that you let the function decide how many rows to create as long as the end result is a rank 1 array.

TIP **To convert a rank 2 array to a rank 1 array, you can also use the `flatten()` or `ravel()` functions. The `flatten()` function always returns a copy of the array, while the `ravel()` and `reshape()` functions return a view (reference) of the original array.**

Array Math

You can perform array math very easily on NumPy arrays. Consider the following two rank 2 arrays:

```
x1 = np.array([[1,2,3],[4,5,6]])
y1 = np.array([[7,8,9],[2,3,4]])
```

To add these two arrays together, you use the `+` operator as follows:

```
print(x1 + y1)
```

The result is the addition of each individual element in the two arrays:

```
[[ 8 10 12]
 [ 6  8 10]]
```

Array math is important, as it can be used to perform vector calculations. A good example is as follows:

```
x = np.array([2,3])
y = np.array([4,2])
z = x + y
'''
[6 5]
'''
```

Figure 2.5 shows the use of arrays to represent vectors and uses array addition to perform vector addition.

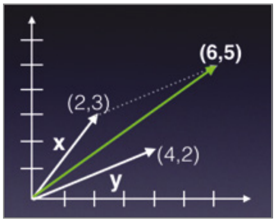

Figure 2.5: Using array addition for vector addition

Besides using the + operator, you can also use the `np.add()` function to add two arrays:

```
np.add(x1,y1)
```

Apart from addition, you can also perform subtraction, multiplication, as well as division with NumPy arrays:

```
print(x1 - y1)     # same as np.subtract(x1,y1)
'''
[[-6 -6 -6]
 [ 2  2  2]]
'''

print(x1 * y1)     # same as np.multiply(x1,y1)
'''
[[ 7 16 27]
 [ 8 15 24]]
'''

print(x1 / y1)     # same as np.divide(x1,y1)
'''
[[ 0.14285714  0.25        0.33333333]
 [ 2.          1.66666667  1.5       ]]
'''
```

What's a practical use of the ability to multiply or divide two arrays? As an example, suppose you have three arrays: one containing the names of a group of people, another the corresponding heights of these individuals, and the last one the corresponding weights of the individuals in the group:

```
names   = np.array(['Ann','Joe','Mark'])
heights = np.array([1.5, 1.78, 1.6])
weights = np.array([65, 46, 59])
```

Now say that you want to calculate the Body Mass Index (BMI) of this group of people. The formula to calculate BMI is as follows:

- Divide the weight in kilograms (kg) by the height in meters (m)
- Divide the answer by the height again

Using the BMI, you can classify a person as healthy, overweight, or underweight using the following categories:

- Underweight if BMI < 18.5
- Overweight if BMI > 25
- Normal weight if 18.5 <= BMI <= 25

Using array division, you could simply calculate BMI using the following statement:

```
bmi = weights/heights **2            # calculate the BMI
print(bmi)                           # [ 28.88888889  14.51836889
23.046875  ]
```

Finding out who is overweight, underweight, or otherwise is now very easy:

```
print("Overweight: " , names[bmi>25])
# Overweight:  ['Ann']
print("Underweight: " , names[bmi<18.5])
# Underweight:  ['Joe']
print("Healthy: " , names[(bmi>=18.5) & (bmi<=25)])
# Healthy:  ['Mark']
```

Dot Product

Note that when you multiply two arrays, you are actually multiplying each of the corresponding elements in the two arrays. Very often, you want to perform a scalar product (also commonly known as *dot product*). The dot product is an algebraic operation that takes two coordinate vectors of equal size and returns a single number. The dot product of two vectors is calculated by multiplying corresponding entries in each vector and adding up all of those products. For example, given two vectors—$a = [a_1, a_2, \ldots, a_n]$ and $b = [b_1, b_2, \ldots, b_n]$—the dot product of these two vectors is $a_1b_1 + a_2b_2 + \ldots + a_nb_n$.

In NumPy, dot product is accomplished using the `dot()` function:

```
x = np.array([2,3])
y = np.array([4,2])
np.dot(x,y)  # 2x4 + 3x2 = 14
```

Dot products also work on rank 2 arrays. If you perform a dot product of two rank 2 arrays, it is equivalent to the following *matrix multiplication*:

```
x2 = np.array([[1,2,3],[4,5,6]])
y2 = np.array([[7,8],[9,10], [11,12]])
print(np.dot(x2,y2))                     # matrix multiplication
'''
[[ 58  64]
 [139 154]]
 '''
```

Figure 2.6 shows how matrix multiplication works. The first result, 58, is derived from the dot product of the first row of the first array and the first column of the second array—1 × 7 + 2 × 9 + 3 × 11 = 58. The second result of 64 is obtained by the dot product of the first row of the first array and the second column of the second array—1 × 8 + 2 × 10 + 3 × 12 = 64. And so on.

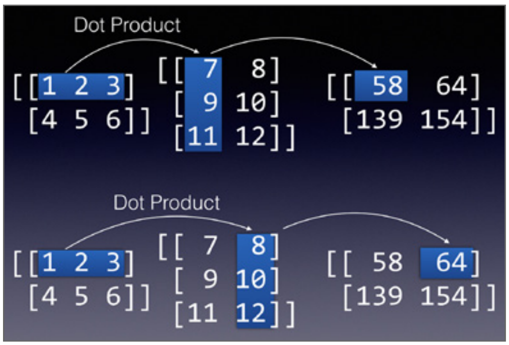

Figure 2.6: Performing matrix multiplication on two arrays

Matrix

NumPy provides another class in addition to arrays (`ndarray`): `matrix`. The *matrix* class is a subclass of the `ndarray`, and it is basically identical to the `ndarray` with one notable exception—a matrix is strictly two-dimensional, while an `ndarray` can be multidimensional. Creating a matrix object is similar to creating a NumPy array:

```
x2 = np.matrix([[1,2],[4,5]])
y2 = np.matrix([[7,8],[2,3]])
```

You can also convert a NumPy array to a matrix using the `asmatrix()` function:

```
x1 = np.array([[1,2],[4,5]])
y1 = np.array([[7,8],[2,3]])
x1 = np.asmatrix(x1)
y1 = np.asmatrix(y1)
```

Another important difference between an `ndarray` and a matrix occurs when you perform multiplications on them. When multiplying two `ndarray` objects, the result is the element-by-element multiplication that we have seen earlier. On the other hand, when multiplying two matrix objects, the result is the dot product (equivalent to the `np.dot()` function):

```
x1 = np.array([[1,2],[4,5]])
y1 = np.array([[7,8],[2,3]])
print(x1 * y1)     # element-by-element multiplication
'''
[[ 7 16]
 [ 8 15]]
'''

x2 = np.matrix([[1,2],[4,5]])
y2 = np.matrix([[7,8],[2,3]])
print(x2 * y2)    # dot product; same as np.dot()
'''
[[11 14]
 [38 47]]
'''
```

Cumulative Sum

Very often, when dealing with numerical data, there is a need to find the cumulative sum of numbers in a NumPy array. Consider the following array:

```
a = np.array([(1,2,3),(4,5,6), (7,8,9)])
print(a)
'''
[[1 2 3]
 [4 5 6]
 [7 8 9]]
'''
```

You can call the `cumsum()` function to get the cumulative sum of the elements:

```
print(a.cumsum())   # prints the cumulative sum of all the
                    # elements in the array
                    # [ 1  3  6 10 15 21 28 36 45]
```

In this case, the `cumsum()` function returns a rank 1 array containing the cumulative sum of all of the elements in the *a* array. The `cumsum()` function also takes in an optional argument—`axis`. Specifying an `axis` of `0` indicates that you want to get the cumulative sum of each column:

```
print(a.cumsum(axis=0))  # sum over rows for each of the 3 columns
'''
```

```
[[ 1  2  3]
 [ 5  7  9]
 [12 15 18]]
'''
```

Specifying an axis of `1` indicates that you want to get the cumulative sum of each row:

```
print(a.cumsum(axis=1))  # sum over columns for each of the 3 rows
'''
[[ 1  3  6]
 [ 4  9 15]
 [ 7 15 24]]
'''
```

Figure 2.7 makes it easy to understand how the `axis` parameter affects the way that cumulative sums are derived.

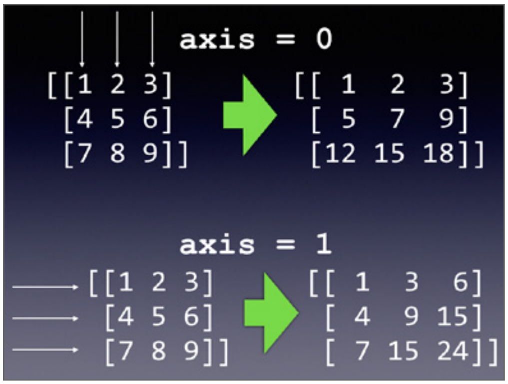

Figure 2.7: Performing cumulative sums on columns and rows

NumPy Sorting

NumPy provides a number of efficient sorting functions that make it very easy to sort an array. The first function for sorting is `sort()`, which takes in an array and returns a sorted array. Consider the following:

```
ages = np.array([34,12,37,5,13])
sorted_ages = np.sort(ages)   # does not modify the original array
print(sorted_ages)            # [ 5 12 13 34 37]
print(ages)                   # [34 12 37  5 13]
```

As you can see from the output, the `sort()` function does not modify the original array. Instead it returns a sorted array. If you want to sort the original array, call the `sort()` function on the array itself as follows:

```
ages.sort()                    # modifies the array
print(ages)                    # [ 5 12 13 34 37]
```

There is another function used for sorting—`argsort()`. To understand how it works, it is useful to examine the following code example:

```
ages = np.array([34,12,37,5,13])
print(ages.argsort())          # [3 1 4 0 2]
```

The `argsort()` function returns the indices that will sort an array. In the preceding example, the first element (3) in the result of the `argsort()` function means that the smallest element after the sort is in index 3 of the original array, which is the number 5. The next number is in index `1`, which is the number 12, and so on. Figure 2.8 shows the meaning of the sort indices.

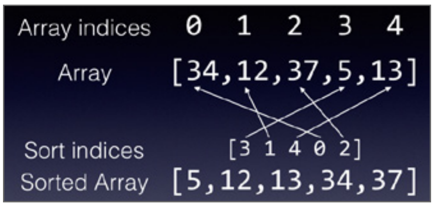

Figure 2.8: Understanding the meaning of the result of the `argsort()` function

To print the sorted *ages* array, use the result of `argsort()` as the index to the *ages* array:

```
print(ages[ages.argsort()])    # [ 5 12 13 34 37]
```

What is the real use of `argsort()`? Imagine that you have three arrays representing a list of people, along with their ages and heights:

```
persons = np.array(['Johnny','Mary','Peter','Will','Joe'])
ages    = np.array([34,12,37,5,13])
heights = np.array([1.76,1.2,1.68,0.5,1.25])
```

Suppose that you want to sort this group of people by age. If you simply sort the *ages* array by itself, the other two arrays would not be sorted correctly based on age. This is where `argsort()` comes in really handy:

```
sort_indices = np.argsort(ages)  # performs a sort based on ages
                                 # and returns an array of indices
                                 # indicating the sort order
```

Once the sort indices are obtained, simply feed them into the three arrays:

```
print(persons[sort_indices])      # ['Will' 'Mary' 'Joe' 'Johnny'
'Peter']
print(ages[sort_indices])         # [ 5 12 13 34 37]
print(heights[sort_indices])      # [ 0.5   1.2   1.25  1.76  1.68]
```

They would now be sorted based on age. As you can see, Will is the youngest, followed by Mary, and so on. The corresponding height for each person would also be in the correct order.

If you wish to sort based on name, then simply use `argsort()` on the *persons* array and feed the resulting indices into the three arrays:

```
sort_indices = np.argsort(persons)   # sort based on names
print(persons[sort_indices])         # ['Joe' 'Johnny' 'Mary' 'Peter'
'Will']
print(ages[sort_indices])            # [13 34 12 37  5]
print(heights[sort_indices])         # [ 1.25  1.76  1.2   1.68  0.5 ]
```

To reverse the order of the names and display them in descending order, use the Python `[::-1]` notation:

```
reverse_sort_indices = np.argsort(persons)[::-1] # reverse the order of a list
print(persons[reverse_sort_indices])     # ['Will' 'Peter' 'Mary'
                                         #  'Johnny' 'Joe']
print(ages[reverse_sort_indices])        # [ 5 37 12 34 13]
print(heights[reverse_sort_indices])     # [ 0.5   1.68  1.2   1.76
                                         #   1.25]
```

Array Assignment

When assigning NumPy arrays, you have to take note of how arrays are assigned. Following are a number of examples to illustrate this.

Copying by Reference

Consider an array named *a1*:

```
list1 = [[1,2,3,4], [5,6,7,8]]
a1 = np.array(list1)
print(a1)
'''
[[1 2 3 4]
 [5 6 7 8]]
'''
```

When you try to assign *a1* to another variable, *a2*, a copy of the array is created:

```
a2 = a1    # creates a copy by reference
print(a1)
'''
[[1 2 3 4]
 [5 6 7 8]]
'''

print(a2)
'''
[[1 2 3 4]
 [5 6 7 8]]
'''
```

However, *a2* is actually pointing to the original *a1*. So, any changes made to either array will affect the other as follows:

```
a2[0][0] = 11      # make some changes to a2
print(a1)          # affects a1
'''
[[11  2  3  4]
 [ 5  6  7  8]]
'''

print(a2)
'''
[[11  2  3  4]
 [ 5  6  7  8]]
'''
```

TIP **In the "Reshaping Arrays" section earlier in this chapter, you saw how to change the shape of an** `ndarray` **using the** `reshape()` **function. In addition to using the** `reshape()` **function, you can also use the shape property of the** `ndarray` **to change its dimension.**

If *a1* now changes shape, *a2* will also be affected as follows:

```
a1.shape = 1,-1   # reshape a1
print(a1)
'''
[[11  2  3  4  5  6  7  8]]
'''

print(a2)         # a2 also changes shape
'''
[[11  2  3  4  5  6  7  8]]
'''
```

Copying by View (Shallow Copy)

NumPy has a `view()` function that allows you to create a copy of an array by reference, while at the same time ensuring that changing the shape of the original array does not affect the shape of the copy. This is known as a *shallow copy*. Let's take a look at an example to understand how this works:

```
a2 = a1.view()     # creates a copy of a1 by reference; but changes
                   # in dimension in a1 will not affect a2
print(a1)
'''
[[1 2 3 4]
 [5 6 7 8]]
'''

print(a2)
'''
[[1 2 3 4]
 [5 6 7 8]]
'''
```

As usual, modify a value in *a1* and you will see the changes in *a2*:

```
a1[0][0] = 11      # make some changes in a1
print(a1)
'''
[[11  2  3  4]
 [ 5  6  7  8]]
'''

print(a2)          # changes is also seen in a2
'''
[[11  2  3  4]
 [ 5  6  7  8]]
'''
```

Up until now, the shallow copy is identical to the copying performed in the previous section. But with shallow copying, when you change the shape of *a1*, *a2* is unaffected:

```
a1.shape = 1,-1    # change the shape of a1
print(a1)
'''
[[11  2  3  4  5  6  7  8]]
'''
```

```
print(a2)          # a2 does not change shape
'''
[[11  2  3  4]
 [ 5  6  7  8]]
'''
```

Copying by Value (Deep Copy)

If you want to copy an array by value, use the `copy()` function, as in the following example:

```
list1 = [[1,2,3,4], [5,6,7,8]]
a1 = np.array(list1)
a2 = a1.copy()     # create a copy of a1 by value (deep copy)
```

The `copy()` function creates a deep copy of the array—it creates a complete copy of the array and its data. When you assign the copy of the array to another variable, any changes made to the shape of the original array will not affect its copy. Here's the proof:

```
a1[0][0] = 11      # make some changes in a1
print(a1)
'''
[[11  2  3  4]
 [ 5  6  7  8]]
'''

print(a2)          # changes is not seen in a2
'''
[[1 2 3 4]
 [5 6 7 8]]
'''

a1.shape = 1,-1    # change the shape of a1
print(a1)
'''
[[11  2  3  4  5  6  7  8]]
'''

print(a2)          # a2 does not change shape
'''
[[1 2 3 4]
 [5 6 7 8]]
'''
```

Summary

In this chapter, you learned about the use of NumPy as a way to represent data of the same type. You also learned how to create arrays of different dimensions, as well as how to access data stored within the arrays. An important feature of NumPy arrays is their ability to perform array math very easily and efficiently, without requiring you to write lots of code.

In the next chapter, you will learn about another important library that makes dealing with tabular data easy—Pandas.