**Part II Data Science Libraries**

**Chapter 10 Introduction to NumPy and Arrays**

The **NumPy** module is used extensively by Python programmers for mathematical and scientific computing. Most of the **NumPy** library is based on ***arrays***, which are data structures that store values (***elements***) that are all the same type. Typically the elements in arrays are numbers, either integers or floats. The **numpy** module has many built-in functions and the array objects have many associated methods for working with numbers to facilitate applications such as linear algebra and statistics. Many data science applications work extensively with the **numpy** module.

Since it is a module, **numpy** must first be imported. It is very common to rename it as *np*:

*>>> import numpy as np*

The rest of this chapter will assume that the **numpy** module has been imported in this manner.

**10.1 Array Basics**

Arrays have ***dimensions***. An array with one dimension is either a row or a column. Sometimes these are called vectors, either a ***row vector*** or a ***column vector***. Like other sequences in Python such as lists, arrays can be indexed, and the indices begin at 0.

A ***one dimensional array*** which is a row vector containing 4 elements might be depicted as:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| 33 | 11 | -5 | 14 |

In **NumPy**, the dimensions are called ***axes***, so this is an array with one axis. We say that this is a 1 x 4 (read “1 by 4”) array; it has 1 row and 4 columns. The ***length*** of the array is the number of elements, which is 4.

A one dimensional column vector might be depicted as:

|  |  |
| --- | --- |
| 0 | 47 |
| 1 | 6 |
| 2 | 15 |

Again, this is an array with one axis. We say that it is a 3 x 1 array; it has 3 rows and 1 column. The length of this array is 3.

Elements in a one dimensional array are found using one index.

A ***two dimensional array*** has two axes. A two dimensional array looks like a table, with both rows and columns. For example, a 2 x 4 array (2 rows, 4 columns) might be depicted as:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | -2 | 18 | 9 | 32 |
| 1 | 44 | -7 | 4 | 99 |

The first axis is the rows, so the length of the first axis is 2. The second axis is the columns, so the length of the second axis is 4. Two dimensional arrays have indices for both the rows and the columns, and both start at 0. Elements are indexed using two indices, the first for the row, and the second for the column.

In general, an ***n-dimensional array*** has n dimensions, or axes. Elements are indexed using n indices. Arrays with two or more dimensions are frequently called ***matrices***.

A ***scalar***, which is a single value, is said to have a dimension of 0 in NumPy. Scalars are not indexed.

**10.2 Creating Arrays and Subarrays**

**The array function and attributes of arrays:**

The basic function that is used to create arrays is the function **ndarray**, but this function has a simpler alias called **array**, which we will use. One method for creating a one dimensional array is to convert a list (or a tuple) to an array using the **array** function. For the rest of this chapter, we will assume that the **array** function has been imported.

*>>> from numpy import array*

*>>> arr1d = array([31, 42, 11])*

Note how the array is displayed by just typing the variable name and by using the **print** function, and note the type of the array.

*>>> arr1d*

array([ 31, 42, 11])

*>>> print(arr1d, type(arr1d))*

[31 42 11] <class 'numpy.ndarray'>

The type of the data elements can be determined using the **dtype** method.

*>>> print(arr1d.dtype)*

int64

The type **int64** is an integer type that uses 64 bits to store integers.

There are several methods that return the number of dimensions, and the lengths of the axes. The **ndim** method returns the number of dimensions, or in other words the number of axes.

*>>> arr1d.ndim*

1

The lengths of the axes are found using the **shape** method.

*>>> arr1d.shape*

(3,)

In this case, there is only one axis since it is a one-dimensional array, and the length is 3. Notice that the result is returned as a tuple.

The **size** of an array is the number of elements. This is always going to be a single integer.

*>>> arr1d.size*

3

Two dimensional arrays can be created by passing a nested list (a list of lists) to the **array** function. In the following example, two lists, each with 3 numbers, are used to create a 2 x 3 array.

*>>> arr2d = array([[1, 3, 8], [-2, 5, 33]])*

*>>> print(arr2d)*

[[ 1 3 8]

[-2 5 33]]

The two nested lists must have the same number of elements, so there are the same number of elements in each row of the array. The number of dimensions is 2. The **shape** function returns the tuple (2, 3) since it is a 2 x 3 array, and the **size** (the total number of elements) is 6.

*>>> arr2d.ndim*

2

*>>> arr2d.shape*

(2, 3)

*>>> arr2d.size*

6

**Functions and methods that create arrays:**

There are quite a few ways in which arrays can be created. Arrays can be created using a range:

*>>> rgarr = array(range(5))*

*>>> print(rgarr)*

[0 1 2 3 4]

The **numpy** function **arange** specifically creates an array using a range.

*>>> rgarr = np.arange(5)*

*>>> print(rgarr)*

[0 1 2 3 4]

Like the **range** function, **arange** can specify starting, ending, and step values.

*>>> myarr = np.arange(3,9,2)*

*>>> print(myarr)*

[3 5 7]

There are several other functions that can be used to create arrays.

The **linspace** function can be used to create array elements that are linearly spaced within a specified range. The following specifies that the range begins with 1, ends with 9, with 3 linearly spaced points. The **linspace** function determines the step value (in this case, 4). Notice that the default type returned by **linspace** is **float64**, even if the numbers are integers.

*>>> myarr = np.linspace(1,9,3)*

*>>> print(myarr, myarr.dtype)*

[1. 5. 9.] float64

If the number of elements is not specified, the default is 50. Also, the type can be specified using **dtype**.

*>>> myarr = np.linspace(1,9,3,dtype=np.int16)*

*>>> print(myarr, myarr.dtype)*

[1 5 9] int16

The type **int16** is an integer type that uses 16 bits, so it stores smaller integers than the type **int64**.

An array of all zeros can be created using the **zeros** function, by specifying the shape. The default type of each element is **float64**, but like the **linspace** function, this can be modified using **dtype**.

*>>> print(np.zeros((2,4), dtype=int))*

[[0 0 0 0]

[0 0 0 0]]

If only one integer *n* is specified for the shape, a 1 x n row vector is created.

*>>> zarr = np.zeros(3)*

>>> print(zarr)

[0. 0. 0.]

Three dimensional arrays can be created by specifying a tuple of 3 integers for the shape. For example, a shape of (2, 3, 5) creates two 3 x 5 arrays. Notice that when displayed, the two arrays are printed separately with a blank line in between them.

*>>> print(np.zeros((2,3,5), dtype=np.int16))*

[[[0 0 0 0 0]

[0 0 0 0 0]

[0 0 0 0 0]]

[[0 0 0 0 0]

[0 0 0 0 0]

[0 0 0 0 0]]]

Images are often stored as three dimensional arrays. Arrays with more than 3 dimensions can also be created, but it becomes more difficult to visualize them and to find useful applications for them.

One reason for creating an array of all zeros might be if they are going to be running sums or counters.

Arrays of all ones can be created using the **ones** function, which uses the same format as the **zeros** function. An array of all ones might be used as a starting point for running products.

*>>> print(np.ones(6, dtype=int))*

[1 1 1 1 1 1]

To fill in an array with any number other than zeros or ones, the **full** function can be used.

*>>> print(np.full((2,3), 11))*

[[11 11 11]

[11 11 11]]

Instead of just a single number, a set of numbers to be used for each row can be passed to the **full** function, for example using **range**.

*>>> newmat = np.full((3,4), range(4))*

*>>> print(newmat)*

[[0 1 2 3]

[0 1 2 3]

[0 1 2 3]]

Another function that creates arrays is the **empty** function, which contrary to its name does not create an empty array, but fills in the elements with whatever is currently in the memory locations used for the array. This function is useful when the shape of an array is known, but the actual values in the elements will be filled in later. Of course, your results may vary!

*>>> weird = np.empty(4)*

*>>> print(weird)*

[-2.68156159e+154 -3.11108761e+231 2.68678769e+154 2.82470694e-309]

It is frequently useful to create arrays in which random numbers are stored in the elements. **NumPy** has a **random** module that has a method, **default\_rng**, which is used to create a random number generator. Once that has been created, the method **random** can be used to create random floats, and the method **integers** can be used to create random integers in a specified range.

An integer can be passed to **default\_rng** which is the seed for the random number generator; if no argument is passed the seed will be obtained from the operating system.

*>>> rng = np.random.default\_rng()*

*>>> farr = rng.random(3)*

*>>> print(farr)*

[0.74899722 0.9414599 0.70685837]

In the following example, a 1 x 4 array of random integers is created in the range from 3 to 9 inclusive.

*>>> rng.integers(3,10,4)*

array([9, 6, 3, 9])

**Indexing and Slicing:**

Arrays can be indexed and sliced similarly to Python sequences such as lists. The indices begin at 0, and negative indices allow indexing from the end of the array.

*>>> arr1d = array([31, 42, 11])*

*>>> arr1d[1]*

42

*>>> arr1d[-1]*

11

With two dimensional arrays, the row index is given first, and then the column index.

*>>> arr2d = array([[1, 3, 8], [-2, 5, 33]])*

*>>> print(arr2d)*

[[ 1 3 8]

[-2 5 33]]

*>>> arr2d[1,2]*

33

Slicing creates another array.

*>>> arr1d[:]*

array([31, 42, 11])

*>>> arr2d[0,1:]*

array([3, 8])

In addition to using integer indices, **NumPy** allows ***logical indexing***. With logical indexing, Boolean expressions resulting in **True** or **False** can be used to index into an array. This returns the elements from the array for which the corresponding element in the logical array is **True**.

*>>> logarr = arr1d > 20*

*>>> logarr*

array([ True, True, False])

*>>> arr1d[logarr]*

array([31, 42])

Notice that for a two dimensional array, with logical indexing the resulting array is flattened into a one dimensional array. The elements are taken from the original array one row at a time.

*>>> print(arr2d)*

[[ 1 3 8]

[-2 5 33]]

*>>> arr2d[arr2d > 0]*

array([ 1, 3, 8, 5, 33])

**Changing Dimensions:**

There are NumPy methods that change the shape of an array.

The **reshape** method can take any array and reshape it into a new array, as long as the new array has the same number of elements as the original. For example, a 1 x 6 array could be reshaped into a 6 x 1 array, a 2 x 3 array, or a 3 x 2 array.

*>>> myarr = np.arange(6)*

*>>> print(myarr)*

[0 1 2 3 4 5]

*>>> myarr.reshape(6,1)*

array([[0],

[1],

[2],

[3],

[4],

[5]])

*>>> arr2by3 = myarr.reshape(2,3)*

*>>> print(arr2by3)*

[[0 1 2]

[3 4 5]]

The **reshape** method returns a reshaped array, but does not modify the original array. The **resize** method works just like **reshape** except that it does modify the original array.

*>>> myarr = np.arange(6)*

*>>> myarr.resize(3,2)*

*>>> print(myarr)*

[[0 1]

[2 3]

[4 5]]

The **ravel** method flattens an n-dimensional array into a one dimensional array. The default is that it flattens it one row at a time. The **ravel** method returns a flattened array, but does not modify the original.

*>>> print(arr2by3)*

[[0 1 2]

[3 4 5]]

*>>> arr2by3.ravel()*

array([0, 1, 2, 3, 4, 5])

*>>> print(arr2by3)*

[[0 1 2]

[3 4 5]]

The method named simply **T** ***transposes*** an array, which for a two dimensional array means that it interchanges the rows and columns. The method returns a new array, but does not modify the original. In the following example, the first row [0 1 2] becomes the first column, and the second row becomes the second column. The other way to view it is that the first column becomes the first row, the second column becomes the second row, and the third column becomes the third row.

*>>> arr3by2 = arr2by3.T*

*>>> print(arr3by2)*

[[0 3]

[1 4]

[2 5]]

**Combining Arrays:**

There are several functions that create new arrays by combining existing arrays.

The function **vstack** vertically ***stacks***, or ***concatenates***, arrays. A tuple consisting of the arrays to be stacked is passed to the **vstack** function. In this case the one-dimensional arrays have to have the same number of elements so that the columns will match up in the stacked array.

*>>> arrone = np.array([4,11,9])*

*>>> arrtwo = np.array([3,15,22])*

*>>> np.vstack((arrone, arrtwo))*

array([[ 4, 11, 9],

[ 3, 15, 22]])

The function **hstack** horizontally stacks arrays.

*>>> np.hstack((arrtwo,arrone))*

array([ 3, 15, 22, 4, 11, 9])