**Python Numpy Tutorial**

One of the robust and most commonly used Python library is NumPy. The Python Library is a collection of script modules which are accessible to a Python program. It helps to simplify the programming process and it also removes the need to rewrite commonly used commands again and again. NumPy stands for Numerical Python which provides a multidimensional array object, an assortment of routines for fast operations on arrays, and various derived objects (such as masked arrays and matrices), including mathematical, logical, basic linear algebra, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic statistical operations, random simulation and much more.

Some of the key features of NumPy are:

· It is a powerful N-dimensional array object

· It is a sophisticated broadcasting functions

· It is a tool for integrating C/C++ and Fortran code

· It is useful linear algebra, Fourier transform, and random number capabilities

In this tutorial, we will go through how to use NumPy to analyze data on Starbucks Menu. This dataset consist of information related to various Beverage available at Starbucks which include attributes like Calories, Total Fat (g), Sodium (mg), Total Carbohydrates (g), Cholesterol (mg), Sugars (g), Protein (g), Caffeine (mg). Here we will learn how we can work with NumPy, and we will try to figure out the Nutrition facts for Starbucks Menu.

Here we have the first few rows of the starbucks.csv file, which we’ll be using throughout this tutorial:

Calories, Total Fat (g), Sodium (mg), Total Carbohydrates (g), Cholesterol (mg), Sugars (g), Protein (g), Caffeine (mg), Nutrition\_Value

3,0.1,0,5,0,0,0.3,175,5

70,0.1,5,75,10,9,6,75,5

110,1.5,5,60,21,17,7,85,6

100,0.1,5,70,19,18,6,75,4

5,0,0,5,1,0,0.4,75,5

The data is in the csv (colon separated values) format — each record is separated by a colon (,), and rows are separated by a new line. There are approx. 1800 rows, including a header row, and 9 columns in the file.

Here we have the list of topics if you want to jump right into a specific one:

· Lists of Lists for CSV Data

· NumPy 2-Dimensional Arrays

o Creating a NumPy Array

o Alternative NumPy Array Creation Methods

· Reading Text Files

· NumPy Arrays — Indexing & Slicing

· Multi-Dimensional NumPy Arrays

· NumPy Data Types

o Converting NumPy Data Types

· NumPy Array Operations

o Simple Array Math

o Multiple Array Math

· Broadcasting

· NumPy Array Methods

· NumPy Array Comparison and Filtering

· Reshaping NumPy Arrays

· Conclusion

**Lists of Lists for CSV Data**

Before proceeding to NumPy, on important thing if you have ssv format file data set, then convert it into csv format file by using the csv.reader object and pass the keyword argument delimiter as “,” , this will help us to read into the content and split up all the content that are available in the ssv file.

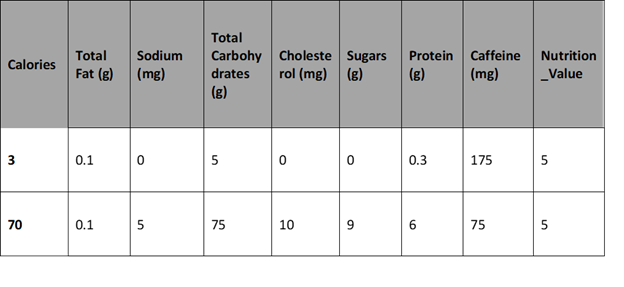
import csv

with open(‘Starbucks.csv’, ‘r’) as f:

starbucks = list(csv.reader(f, delimiter=’,’))

print(starbucks)

It’s always good to have data in the right format in a table to make it easier to view:



As we can observe from the table above, here we have first 3 three rows from the entire table, where the first row contains column headers. The first row is the header row and next rows after header represent the various Beverage at Starbucks. The first element of each row is the Calories, the second is the total fats, and so on. We can find the average Nutrition value. The below code will:

· First of all, we have to extract the last element from each of row after the header row.

· Then we have to convert the extracted element to a float data type.

· Then we have to assign all the extracted elements to the list qualities.

And then we will divide the sum of all the elements in qualities by the total number of elements in qualities to the get the mean.

**Nutrition\_Value = [float (item[-1]) for item in Starbucks[1:]]**

**Sum (Nutrition\_Value) / len (Nutrition\_Value)**

**Out put: 5.6360225140712945**

Here, we are able to do the calculation in the way we wanted, but the code is fairly little complex and it won’t be fun if we have to repeat something similar every time to compute average Nutrition\_Value. So, we are Lucky use NumPy library to make it easier to work with our data.

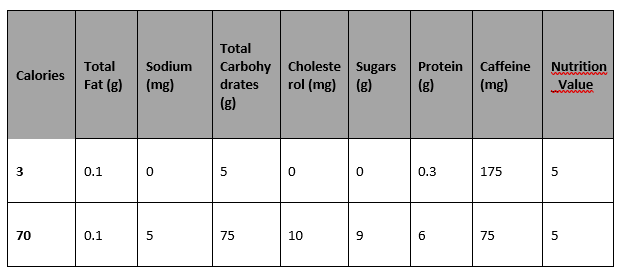
Let’s explore it!

**Numpy 2-Dimensional Arrays**

In NumPy, it is very easy to work with multidimensional arrays. Here in this tutorial, we will dive into various types of multidimensional arrays but currently, we are focusing on 2-dimensional arrays.

A 2-dimensional array is also called as a matrix and is just an acronym about a list of lists. A 2-dimensional array is a collection of rows and columns. By specifying a row number and a column number, we can easily extract an element from a matrix.

In the below 2-dimensional array, the first row is the header row, and the first column is the Beverage column:



If we pick the element which is present at the first row and the second column, that is total fat. If we pick the element in the third row and the second column, here we will get 0.1.

In a NumPy array, the rank is specified to the number of dimensions, and each dimension is called an axis. So, the first axis is the row, and the second axis is the column.

So, this are the basics of matrices, now, we will see how we can get from our list of lists to a NumPy array.

**Creating A NumPy Array**

The numpy.array function is used to create a NumPy array. Here we just have to pass in a list of lists, it will automatically generate a NumPy array with the same number of rows and columns. Because for easy computation, we want all the elements in the array to be float elements, so, we’ll leave off the header row and column, which contains strings.

This is the limitations of NumPy. As in NumPy, all the elements in an array have to be of the same type. Here, if we include the header row and first column, then all the elements in the array will be read in as a string. So, to do computations in the way we want like find the average Nutrition\_Value, we need the elements to be presented in floats.

In the below code, we:

· First, we will Import the numpy package.

· Then we will pass the list of lists Starbucks into the array function, which converts it into a NumPy array.

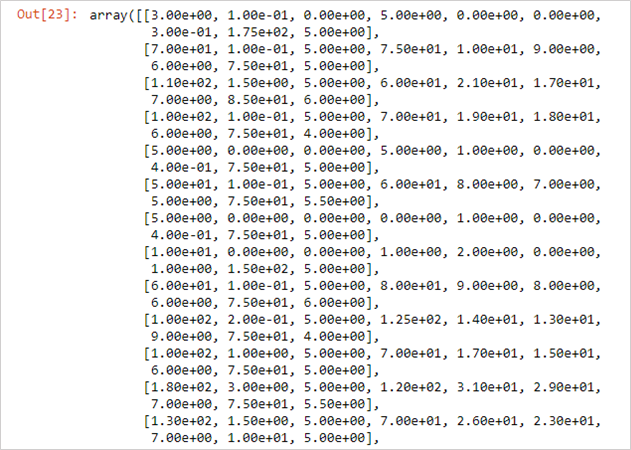
· Here we will exclude the header row and first column with list slicing.

· Then we will specify the keyword argument dtype to make sure each element is converted to a float. We will explore more about what is dtype later on.

**import numpy as np**

**starbucks = np.array(starbucks[1:], dtype=np.float)**

**starbucks**



Now, to check the number of rows and columns in our data, we will use the shape property of NumPy arrays:

**starbucks.shape**



**Alternative NumPy Array Creation Methods**

Do you know, we can use a variety of methods to create NumPy arrays? First, we will look at the creation of an array where every element is zero. The below code will create an array with 4 rows and 3 columns, where every element is 0, here we will be using numpy.zeros:

**import numpy as np**

**empty\_array = np.zeros((4,3))**

**empty\_array**



An array with all zero elements will be useful at the time when you want an array of fixed size, otherwise, it will not have any value.

Similarly, we can create an array of all ones

**import numpy as np**

**All\_One\_array = np.ones((4,3))**

**All\_One\_array**

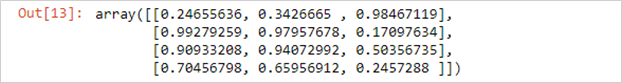


We can also create an array is random number using numpy.random.rand. Here’s an example:

**import numpy as np**

**Random\_array = np.random.rand(4,3)**

**Random\_array**



Creating an array which is completely filled random numbers can be useful at a time when we want to quickly test our code with sample arrays.

**Reading Text Files**

Here in NumPy we can directly read csv or other files into an array. This can be done by using the numpy.genfromtxt function. We will use this function on our initial data on Starbucks.

So, here in the code:

· To read in the Starbucks.csv file, here we will use the genfromtxt function.

· Next, we have specified the keyword argument delimiter as”,” so that the fields are parsed properly.

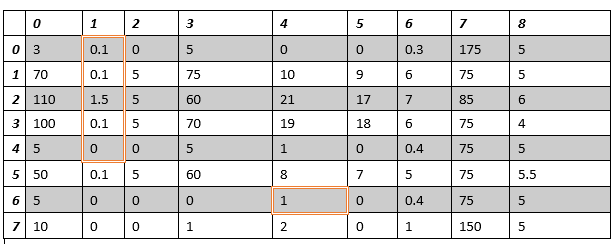
And then we have specified the keyword argument skip\_header=1, this will help to eliminate the header row.

**starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

Here, if we read it into a list and then converted it to an array of floats, the Starbucks will be looking the same. Here, NumPy will automatically pick a data type for the elements in an array based on their format.

**NumPy Arrays — Indexing & Slicing**

So, how we can do Indexing and slicing in the created NumPy Arrays to retrieve results from them. In NumPy, the index for first row and column starts with 0. Suppose if we want to select the fifth column then its index will be 4 or if we want to select 3-row data then, its index is 2 and so on.



Let’s say, we want to select the element at row 7 and column 3 so, we will pass index 6 as row index, and the index 4 as the column index:

**starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

**starbucks[6,4]**



So, here with the help of the index, we have shown indexing.

**starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

**starbucks[0:5,1]**

Suppose we want to select the first 5 elements from the 2nd column. This we can implement by using a colon (:). A colon in slicing indicates that we want to select all the elements from the starting index up to but here we are not including the ending index.

**starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

**starbucks[0:5,1]**



And suppose we want to select the entire column then just by using the colon (:), with no starting or ending indices we will get the desired result.

**starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

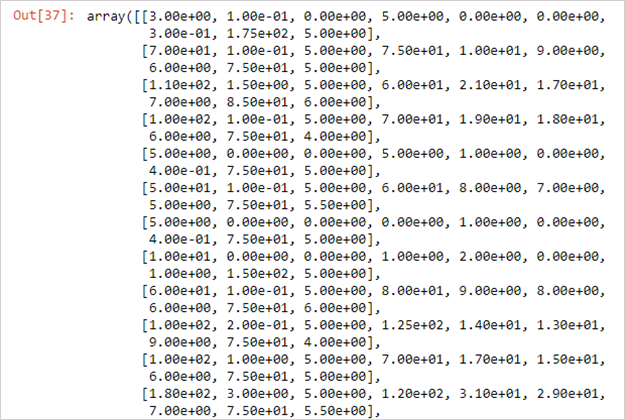
**starbucks[:,1]**



And suppose we want to select the entire array then use two colons to select all the rows and column. But this is not required while creating a good application.

**starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

**starbucks[:,:]**



So, how we, can assign values to certain elements in arrays?

We can do that by directly assigning that value to the particular element. Here in the example code:

**starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

**starbucks[1,5] =10**

**starbucks[1,5]**



Even we can overwrite the entire column by using this code:

**starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

**starbucks[:5]=10**

This will overwrite the entire 6th column with 10

**Multi-Dimensional NumPy Arrays**

Currently, we have worked with Starbucks array which was a 2-dimensional array. However, NumPy package provides us a privilege to work with a multidimensional array. The most common multi-dimensional array is a One-dimensional array. Have you noticed when we sliced the Starbucks data, there we have created the one-dimensional array. A 1-dimensional array will have a single index to retrieve an element from it. Do you know, each row and column in a 2-dimensional array is treated as a 1-dimensional array. As for a list of lists is analogous is a 2-dimensional array, similarly for a single list, the analogous is a 1-dimensional array. Here suppose we slice Starbucks data and retrieve only the fifth row, then as output, we will receive a 1-dimensional array.

**Fifth\_starbucks= np.genfromtxt(“Starbucks.csv”, delimiter=”,”, skip\_header=1)**

**Fifth\_starbucks = starbucks[5,:]**

**Fifth\_starbucks**



And suppose if we want to retrieve an individual element from Fifth\_starbucks we can do that by using a single index.

**Fifth\_starbucks[0]**



Even most of the NumPy functions, such as numpy.random.rand, which we have used with multidimensional arrays to generate a random vector can be used with a single dimensional array as well. Here we just have to pass the single parameter.

**np.random.rand(5)**



Mostly for our applications, we deal with the 1, 2, 3-dimensional array, it is very real that we come across the array which is more than a three-dimensional array, think this as a list of lists of lists.

For a better understanding of this let’s take an example of the monthly earning of a supermarket. The month vise data will be in a form of a list and if we want a quick look on it then we can see that data in a quarter vise and year vise.

A monthly earning of a supermarket will look something like this:

**400, 250, 300, 470, 560, 630, 820, 740, 605, 340,420,340**

Here, the supermarket has earned $400 in January, $250 in February, and so on. If we split this earning in a quarter vise then it will be a list of the list:

**One\_Year = [**

**[400, 250, 300],**

**[470, 560, 630],**

**[820, 740, 605]**

**[340,420,340]]**

**]**

Here, we can retrieve the earning of January month by calling One\_Year[0],[0]. And if we want a result

for a complete quarter, then we can all One\_Year[0] or One\_Year[1]. So, this is a two-dimensional array. But here if we add the earning of another year then it will become the third dimension.

**Yearly\_Earning = [**

**[**

**[400, 250, 300],**

**[470, 560, 630],**

**[820, 740, 605],**

**[340,420,340]]**

**]**

**[**

**[500, 350, 430],**

**[430, 760, 640],**

**[720, 530, 800],**

**[345,900, 700]**

**]**

**]**

Here, we can retrieve the first year earning of the month of January by calling Yearly\_Earning[0],[0],[0]

So, here we need three indexes to retrieve a single element we have the same case in a three-dimensional array in NumPy, in fact, we can convert this Yearly\_Earning to an array and then we can get the earnings for January month of the first year. It will be:

**Yearly\_Earning = np.array(Yearly\_Earning)**

**Yearly\_Earning [0,0,0]**

We will get the result as

**400**

Here, we know the shape of the array, we will use:

**Yearly\_Earning.shape**

The result will be

**2,4,3**

Here, in three-dimensional array also Indexing and slicing work exact in the same way as two-dimensional array but here we have to pass in one extra axis.

Suppose if we need the earning for January of all years, then it could be:

**Yearly\_Earning[:,0,0]**

The result will be:

**array ([400,500])**

Suppose, if we require to get first-quarter earnings from both years, then:

**Yearly\_Earning[:,0,:]**

The result will be:

**array ( [ [400, 250, 300],**

**[500, 350, 430]])**

By adding more dimensions, we can make it much easier for us to query our data as it will be organized in a certain way.

Suppose if we go from 3-dimensional arrays to 4-dimensional or a larger array than that, there also we will apply the same properties, and they can be indexed and sliced in the same ways.

**NumPy Data Types**

As we have discussed earlier, each element of a NumPy array can be stored in a single data type. In our example of Starbucks, all element contains only float values. In NumPy values are stored using its own data types, which are different from Python data types like float and str. The reason behind this is that the core of NumPy is written in the C programming language, which stores data differently in comparison to the Python data types. NumPy itself map data types between Python and C and allow us to use NumPy arrays without any conversion hitches.

You can find the data type of a NumPy array by accessing its dtype property:

**Starbucks.dtype**



NumPy provides various data types, which are inline with Python data types, like float, and str. Some of the important NumPy data types are:

· float — numeric floating-point data.

· int — integer data.

· string — character data.

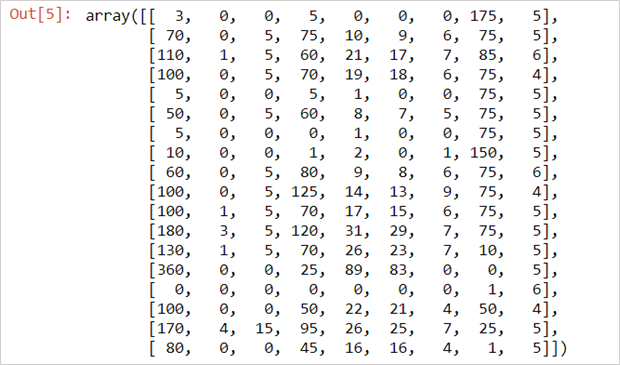
· object — Python objects.

Even we have additional datatypes with a suffix which indicates the bits of memory that that data type can take up. Like int32 is a 32-bit integer data type, and float64 is a 64-bit float data type.

**Converting NumPy Data Types**

To convert an array to a different type we can use the numpy.ndarray.astype method. This method will make a copy of the actual array and will return a new array with the specified data type. For example, if we want to convert Starbucks data to the int data type, we need to perform this:

**starbucks.astype(int)**



With the output, we can observe that all of the elements in the resulting array are integers. And to check the name property of the dtype of the resulting array, we will use the following code:

**integer\_starbucks = starbucks.astype(int)**

**integer\_starbucks.dtype.name**



Here the array has been converted to a 32-bit integer data type which means it will be storing the values as 32-bit integers.

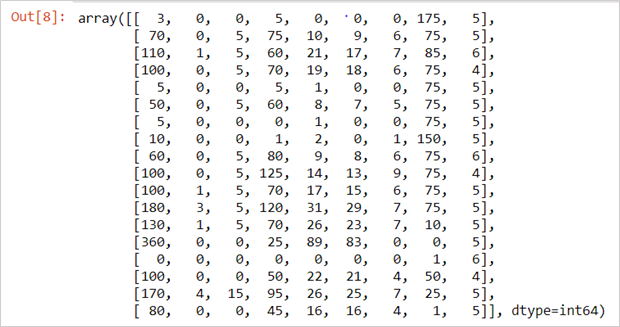
If you want more control over the way, the array is stored in memory and allows for very long integer values then we can directly create a NumPy dtype objects like numpy.int64:

**np.int64**



Now, we can directly use these to convert between data types:

**integer\_starbucks.astype(np.int64)**



**NumPy Array Operations**

It is easy to perform mathematical operations on arrays using NumPy. One of the important advantages of using NumPy is that it male easy to perform the computation.

**Single Array Math**

It is easy to perform basic arithmetic operations on numpy arrays. We can use both +, -, \*, / symbols or add(), subtract(), multiply(), divide() methods to perform basic operations like addition, subtraction, multiplication and division respectively. By using sqrt() function, we can find the square root of each element in numpy array.

Let’s say after the quality check we want to add 10 to the Nutrition\_Value of Starbucks Beverages then

**starbucks[:,8] + 10**



Have you noticed after performing the above operation we have not to change the Starbucks array –but a new 1-dimensional array is returned where 10 has been added to each element in the Nutrition\_Value column of Starbucks data.

Similarly, instead of + if we modify the array with +=, then the result would be:

**starbucks[:,8] += 10**

**starbucks[:,8]**



In the same way, we can perform the other operations. Suppose we want to multiply each of the Nutrition\_Value by 2, which we could do in this way:

**starbucks[:,8] \* 2**



**Multiple Array Math**

We can perform mathematical operations between multiple arrays. This operation will be applied to the pairs of elements. Suppose, if we add the Nutrition\_Value column to itself, here’s what we get:

**starbucks[:,8] + starbucks[:,8]**



Did you noticed that the output is equivalent to starbucks[:,8] \* 2 — this is because here each pair of elements is added by NumPy. The first array first element is added to the second array first element, the first array second element to the second array second element, and so on.

Also, we can use this to multiple arrays. Let’s say we want to pick a beverage that is fat filled with nutrition value then we have to multiply Total fat, Protein, and nutrition\_Value, and later we can select the beverage with the highest score

**starbucks[:,1] \* starbucks[:,6] \* starbucks[:,8]**



We can perform all of the common operations like /, \*, -, +, ^, to work between arrays.

**Broadcasting**

Till now, we have performed the operation on are the exact same size array, and it is done the corresponded elements. But what if the dimension of the two arrays is not similar. However, it is possible to perform NumPy on two arrays which are dissimilar by using broadcast. In broadcast, we will try to match up the elements by using certain rules. Few essential steps involve in broadcasting are:

· Compared the last dimension of each array.

o If the lengths of the dimension are equal, or one of the dimensions lengths is 1, then we keep working.

o If the lengths of dimension aren’t equal, and none of the dimensions lengths have 1, then there is an error.

· Repeat checking dimensions until the shortest array is out of dimensions.

For example, we can compare the following two array shapes:

**X: (60,5)**

**Y: (5)**

The comparison is possible here because the array X has the trailing dimension length as 5, and the array Y has trailing dimension length as 5. They’re equal, as trailing dimension are equal. But Array Y is then out of elements, so for broadcasting, array X is stretched to become an array of with the same shape as array Y., and then arrays are compatible for mathematical operations.

**X: (3,4)**

**Y: (10,4)**

Let’s take another example which is also compatible:

Here, the last dimension of both arrays are matching, and Array X first dimension is of length 1.

**X: (52,52)**

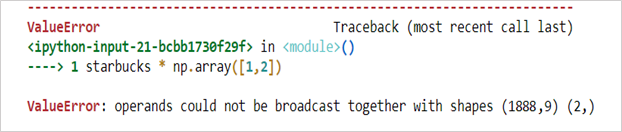
**Y: (55,55)**

Now, for better understanding we will look at another two arrays that don’t match:

Here, in this example neither the lengths of the dimensions are equal, nor either of the arrays has dimension length equal to 1.

Let’s illustrate the principle of Broadcast with the help of our Starbucks dataset:

**starbucks \* np.array([1,2])**



The error statement “ValueError: operands could not be broadcast together with shapes (1888,9) (2,)”

As the two arrays don’t have a matching trailing dimension so the above example didn’t work.

Here’s an example where the last dimension is matching:

**X\_array = np.array(**

**[**

**[3,4],**

**[5,7]**

**]**

**)**

**Y\_array = np.array([5,2])**

**X\_array + Y\_array**



Elements of random\_Example\_array are broadcast over each row of Starbucks, so the first column of Starbucks has the first value in random\_Example\_array added to it, and so on.

**NumPy Array Methods**

NumPy provides so many methods other than arithmetic operations to solve more complex calculations in the array. One of the most commonly used NumPy array methods is numpy.ndarray.sum method. This method helps to find the sum of all the elements in an array when used by default:

**starbucks[:,8].sum()**



The total of all of our Nutrition\_Value column is 271.5.

Here, as a keyword argument for sum method, we can also pass the axis to find sums over an axis.

Suppose if we call sum across our Starbucks matrix, and pass in axis as 0, then we will be able to find the sums over the first axis of the array. As a result, this will provide the sum of all the values in every column.

You would have understood this as the sums over the first axis would give us the sum of each column, or another way to think about this is that the specified axis is the one “going away”.

So if we assign axis=0, then this means that we would like the rows to go away, and we are willing to find the sums for each of the remaining axes across each row:

**starbucks.sum(axis=0)**



To verify whether our sum is correct, we can check the shape. In our dataset the shape should be 9, corresponding to the number of columns:

**starbucks.sum(axis=0).shape**



Here, if we provide in axis=1, then it will find the sums over the second axis of the array.

**starbucks.sum(axis=1)**



Other than the sum, in NumPy we have several other methods which work like the sum method, including:

· numpy.ndarray.min — finds the minimum value in an array.

· numpy.ndarray.max — finds the maximum value in an array.

· numpy.ndarray.std — finds the standard deviation of an array.

· numpy.ndarray.mean — finds the mean of an array.

· And many more

**NumPy Array Comparisons**

In NumPy, it possible to test and check whether the rows match the certain values by using mathematical comparison operations like <, >, >=, <=, and ==.

Suppose in our Starbucks data, if we want to check which beverages have a Nutrition\_Value greater than 5, we can do this:

**starbucks[:,8] > 5**



As a result, we have received a Boolean array which tells us which of the beverage have a Nutrition\_Value greater than 5. We can perform similar things with the other operators. For instance, we can see if there is any beverage which has a Nutrition\_Value equal to 10:

**starbucks[:,8] == 10**

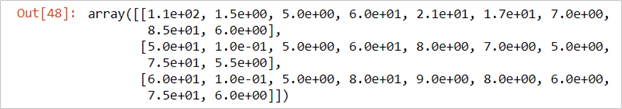


**Subsetting**

With a Boolean and a NumPy array, one of the powerful things we can do is select only certain rows or columns as per our requirement. For example, if we are will select only those rows from the Starbucks data where Nutrition\_Value of beverage is greater than 5:

**Highly\_Nutrition = starbucks[:,8] > 5**

**starbucks[Highly\_Nutrition,:][:3,:]**



Here, we have selected only three rows where Highly\_Nutrition contains value is True, with its all columns.

So, Subsetting makes it easier to filter arrays with certain criteria.

Another example, If we are looking for a beverage with a lot of protein and Highly\_Nutrition. In order to specify the multiple conditions, we will place each condition in parentheses, and we will separate conditions with an ampersand (&):

**Highly\_Nutrition\_and\_Max\_Protein = (starbucks[:,8] > 5) & (starbucks[:,6] > 6)**

**starbucks[Highly\_Nutrition\_and\_Max\_Protein,2:]**



Here, even we can combine subsetting and assignment to overwrite certain values in an array:

**Highly\_Nutrition\_and\_Max\_Protein = (starbucks[:,8] > 5) & (starbucks[:,6] > 6)**

**starbucks[Highly\_Nutrition\_and\_Max\_Protein,2:] = 5**

**starbucks[Highly\_Nutrition\_and\_Max\_Protein,2:]**



**Reshaping NumPy Arrays**

In NumPy, it is very easy to change the shape of arrays and still protective all of their elements. There are often many functions which make it easier to access array elements.

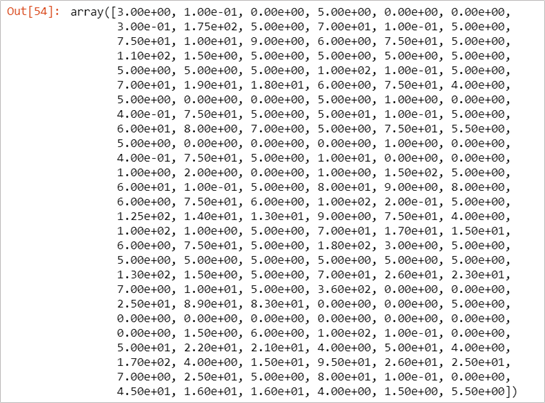
One of the simplest ways of reshaping an array is to flip the axes, like columns, become rows and vice versa. We can perform this operation with the numpy.transpose function.

**np.transpose(starbucks).shape**



Another important function is numpy.ravel function. This function will turn an array into a one-dimensional representation. This function will turn an array into a long sequence of values:

**starbucks.ravel()**



Here we have an example which will help you to better understand and see the ordering of numpy.ravel:

**Example\_Array\_One = np.array(**

**[**

**[1, 2, 3, 4],**

**[5, 6, 7, 8]**

**]**

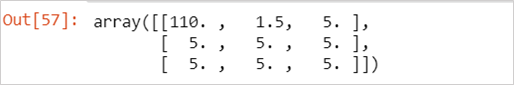
**)**

**Example\_Array\_One.ravel()**



And finally, we are going to use the numpy.reshape function. This function will help to reshape an array to a certain shape as per our requirement. In the below example we will turn the third row of Starbucks data into a 2-dimensional array with 3 rows and 3 columns:

**starbucks[2,:].reshape((3,3))**



Combining NumPy Arrays

With NumPy, we can easily combine multiple arrays into a single unified array. To perform this task, we can use numpy.vstack to vertically stack multiple arrays. Consider this in this way the second arrays’ items are being added as new rows to the first array.

Let’s take an example where we want to combine the old Nutritional dataset of Starbucks beverages with our existing dataset, wines, which contains information on the current Nutritional value of Starbucks beverages.

In the below code, we:

· Read in starbucks\_old\_data.

· Display the shape of starbucks\_old.

**import csv**

**with open(‘starbucks\_old\_data.csv’, ‘r’) as f:**

**starbucks\_old = list(csv.reader(f, delimiter=’,’))**

**import numpy as np**

**starbucks\_old = np.array(starbucks\_old[1:], dtype=np.float)**

**starbucks\_old.shape**



Here we can see, we have attributes for 196 beverages that we have the starbucks\_old data, we can combine all the wine data.

Now, we will use the vstack function to combine starbucks and starbucks\_old data and then we will display the shape of the result

**All\_beverages = np.vstack((starbucks, starbucks\_old))**

**All\_beverages.shape**

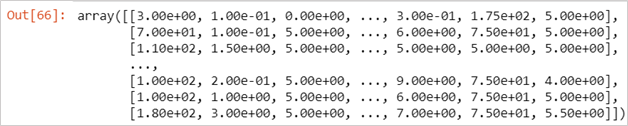


So, here we can observe, the result has 2084 rows, which is the sum of the number of rows in starbucks and starbucks\_old data.

Similarly, If we want to combine arrays horizontally, which means that our number of rows will stay constant, but the columns will be joined. For this purpose we can use the numpy.hstack function.

And another useful function is numpy.concatenate. it is a general purpose version of hstack and vstack. With the help of this fuction, if we want to concatenate two arrays, then we can pass them into concatenate and we have to specify the axis keyword argument that we want to concatenate along. When we concatenating along the first axis, it is similar to vstack, and when we concatenating along the second axis, it is similar to hstack:

**np.concatenate((starbucks, starbucks\_old), axis=0)**



**Conclusion**

This brings us to the end of the Pythons NumPy library tutorial. In this tutorial, we learned in detail the NumPy library with the help of a real-time data set. Here we have also explored how to perform various operations via the NumPy library, which is most commonly used in many data science applications.

Source Link: <https://medium.com/intellipaat/python-numpy-tutorial-b0118b79cf3c>

Further study:

<https://medium.com/@vsvaibhav2016/basics-of-numpy-python-for-data-analysis-45b0c43f591b>

<https://medium.com/@lucyji/numpy-tutorial-the-basics-3ecd2717b855>