WILEY

Data Engineering

Module: Python programming

Topic: Working with NumPy

Scenario

- Understanding NumPy(numerical python) library in Python and implementing creation and manipulation of arrays in NumPy.
- Reshaping of NumPy is very essential operation in data science and can be done using reshape() function
- Index and slicing of NumPy array is yet another essential topic used extensively in preprocessing of data
- A brief view of the practical applications of NumPy array

Background

Understanding Starbucks drinks dataset which contains data about the different nutrients in Starbucks drinks along with their contents and performing finding out the drink with the most nutrition.

Objectives

After the completing this exercise, the learner will be able to –

- How to create NumPy arrays using various methods
- Manipulating NumPy arrays to to perform useful calculations
- Operations on NumPy arrays
- Reshaping NumPy arrays
- Indexing in NumPy
- Sorting and Searching in NumPy

Problem statement

- Load the dataset
- Perform various operations
- Identify the calorie, fats, fibre, protein and sodium array
- Find the beverage with highest fat content

Dataset information

The data set provided consists of 84 beverages served by Starbucks to their customers. It also contains the calories, fats, fibre, protein and sodium contents in each of these.

Python lists are a substitute for arrays, but they fail to deliver the performance required while computing large sets of numerical data. To address this issue we use a python library called NumPy. The word NumPy stands for Numerical Python. NumPy offers an array object called ndarray.

Creating NumPy array

We will use pandas here to read the csv file for the sake of simplicity. Import pandas library and use the read_csv() to get the csv file in the form of a DataFrame.

```
1. import pandas as pd
2. data = pd.read_csv('starbucks-menu-nutrition-drinks.csv')
3. print(data)
```

Let us create a numpy array using np.array() function. Here we will create a simple numerical numpy array of calories function from our previously extracted data.

```
1. cal = np.array(data['Calories'])
2. cal
```

```
array([ 45,  80,  60, 110,   0, 130, 140, 130,  80,  60, 150,
140, 140,       150,  70, 120, 140,  60, 120, 140, 150, 140,
30,  70,  30,  70,        30,  90,  60, 130, 140, 130,  90,  90,
210, 200,  60,  50,  10,        60, 150, 140, 140, 150,  70,
120, 140,  60, 120, 140, 150, 140,         5,  10,   5,   5,
5,   5,   0,   5,  70,   5, 110, 320, 430,       190, 290, 120,
250, 260, 250, 180, 130, 230, 250, 200, 260, 190,       300,
190, 190, 250, 360, 350])
```

Thus, we can see all the calorie contents in each of our drinks present in data.

You can also create an array from scratch from list as follows:

```
1. li = [11,2,53,74,5]
2. nd_li = np.array(li)
3. nd_li
```

```
array([11,  2, 53, 74,  5])
```

A numpy array can also contain strings, lists, numbers, etc, but all the data in the array should be of same type.

```
1. str_li = ['Wiley', 'Python']
2. nd_str_li = np.array(str_li)
3. nd_str_li
```

```
array(['Wiley', 'Python'], dtype='<U6')
```

A array or a matrix can be created as follows

```
1. mat1 = np.array([[1,2,3],
                    [4,5,6],
                    [7,8,9]])        # This is a 3x3 matrix
```

We can also create an ndarray containing all zeros or all ones of any size instead of writing a whole big array manually.

```
1. z = np.zeros(100, dtype=int)
2. z
```

```
array([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, 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, 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, 0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Thus, 100 zeroes of integer datatype are now stored in this array.

We can also work on 2D array as follows. Here we are storing zeros in float datatype.

```
1. z = np.zeros([5,5], dtype=float)
2. z
```

```
array([[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.]])
```

Similarly, a whole bunch of ones can also be stored in numpy arrays.

```
1. o = np.ones(50, dtype=int)
2. o
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
1. o = np.ones([5,5], dtype=str)
2. o
```

```
 array([['1', '1', '1', '1','1'],['1', '1', '1', '1',
'1'],['1', '1', '1', '1', '1'],['1', '1', '1', '1', '1'],['1',
'1', '1', '1', '1']], dtype='<U1')
```

The arange function takes the arguments start, stop and step and returns an array with evenly spaced elements as per the interval. The interval mentioned is half-opened i.e.[start,stop)

```
1. arr1 = np.arange(3)
2. arr1
```

```
array([0, 1, 2])
```

```
1. arr2 = np.arange(1,10,1)
2. arr2
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Now we take step=2, so every 2nd element in the range will be stored in the array.

```
1. arr3 = np.arange(2,30,2)
2. arr3
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

NumPy Array Manipulation

copy() is a function which is used to copy contents of one array to another. Suppose we extract a column say 'Protein' from our data and convert it to numpy and make a copy of it.

```
protein = np.array(data['Protein'])
print(protein)

protein_copy = np.copy(protein)
print(protein_copy)
```

```
[ 0  0  0  0  0  5  5  5  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  5  5  5  0  0 20 20  1  1  1  0  0  0  0
0  0  0  0  0  0  0  0  0  1  1  1  1  0  0  0  0  1  1  1 14 12
13 13  8 10 11 12 12  8  9 10  7 11  7 10 12 12 12 14 15]
[ 0  0  0  0  0  5  5  5  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  5  5  5  0  0 20 20  1  1  1  0  0  0  0
0  0  0  0  0  0  0  0  0  1  1  1  1  0  0  0  0  1  1  1 14 12
13 13  8 10 11 12 12  8  9 10  7 11  7 10 12 12 12 14 15]
```

As we can see that protein_copy copied all the contents of the protein array and thus both the arrays are same.

We can also append certain values to the numpy array using the append() function. This function will insert an element at the end of the list.

```
arr1 = [3,4,5,2]
arr1.append(10)
arr1
```

```
[3, 4, 5, 2, 10]
```

append() function can also be used to append an array to another array.

```
arr2 = [4.3, 23.33, 90.8, 32.75]
arr1.append(arr2)
arr1
```

```
[3, 4, 5, 2, 10, [4.3, 23.33, 90.8, 32.75]]
```

The concatenate function in NumPy joins two or more arrays along a specified axis.

```
fruits = np.array(['apple', 'banana', 'orange', 'kiwi'])
veggies = np.array(['kale', 'cabbage', 'avocado'])
print(np.concatenate((fruits, veggies)))

arr1 = np.array([1,23])
```

```
arr2 = np.array([3,4])
arr3 = np.concatenate((arr1,arr2))
print(arr3)
```

```
['apple' 'banana' 'orange' 'kiwi' 'kale' 'cabbage' ,avocado']
[ 1 23  3  4]
```

Numpy provides some helper functions for stacking elements in two arrays in different ways. hstack() is a function that stacks elements in a row.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.hstack((arr1, arr2))
print(arr)
```

```
[1 2 3 4 5 6]
```

vstack() is another function that stacks array column wise.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.vstack((arr1, arr2))
print(arr)
```

```
[[1 2 3
][4 5 6]]
```

dstack() stacks along height, which is the same as depth.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.dstack((arr1, arr2))
print(arr)
```

```
[[[1 4]
[2 5]                                                    [3
6]]]
```

```
Let's observe matrix operations using numpy
matrix1 = np.array([[1,2], [8,7]])
matrix2 = np.array([[5,6], [0,9]])
# adding two matrix
print("Addition of two matrix")
print(np.add(matrix1, matrix2))
```

```
Addition of two matrix
[[ 6  8 ]
[8 16]]
```

```
# subtraction two matrix
print("Subtraction of two matrix")
print(np.subtract(matrix1, matrix2))
```

```
Subtraction of two matrix
[[-4 -4 ]
[8 -2]]
```

```
# Multiplication of two matrix
print("Multiplication of two matrix")
print(np.dot(matrix1, matrix2))
```

```
Multiplication of two matrix
[[  5 24]
[ 40 111]]
```

It is noticeable here that the multiply() function will simply multiply the numbers at same indexes in two matrices instead of doing the dot product.

```
#Multiplication of two simple arrays
print("Multiplication of array")
print(np.multiply(matrix1, matrix2))
```

```
Multiplication of array
[[ 5 12 ]                                                    [ 0
63]]
```

Reshaping NumPy array

The numpy.reshape() function shapes an array without changing the data of the array. Here we are changing a 1D array to 2x4 dimension.

```
1. arr1 = np.arange(8)
2. arr1_re = arr1.reshape(2, 4)
3. arr1_re
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

We can also convert to a 3D matrix as follows:

```
1. arr2 = np.arange(8).reshape(2, 2, 2)
2. arr2
```

```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```

Operations on NumPy

char.lower() function is used to convert all the characters in the array in lowercase.

```
1. import numpy as np
2. fruits = np.array(['Kiwi', 'ORANGES', 'StrawBerries'])
3. fruits = np.char.lower(fruits)
4. print(fruits)
```

```
['kiwi' 'oranges' 'strawberries']
```

char.upper() function is used to convert all the characters in the array in uppercase.

```
1. fruits = np.array(['Kiwi', 'ORANGES', 'StrawBerries'])
2.  fruits =
   np.char.upper(fruits)
3. print(fruits)
```

```
['KIWI' 'ORANGES' 'STRAWBERRIES']
```

numpy.split() function returns a list of strings after breaking the given string by the specified separator.

```
1. str1 = "Hello World!"
2. np1 = np.char.split(str1)
3. print(np1)
```

```
['Hello', 'World!']
```

numpy.join() function is a string method and returns a string in which the elements of sequence have been joined by str separator.

```
1. print(np.char.join('-', 'wiley'))
2. print(np.char.join(['-', ':', '.'], ['python', 'for',
   'beginners']))
```

```
w-i-l-e-y['p-y-t-h-o-n' 'f:o:r' 'b.e.g.i.n.n.e.r.s']
```

numpy.bitwise_and() function is used to compute the bit-wise AND of two array element-wise. This function computes the bit-wise AND of the underlying binary representation of the integers in the input arrays.

```
1. arr1 = [2, 8, 125]
2. arr2 = [3, 3, 115]

3. print("Input array1 : ", arr1)
4. print("Input array2 : ", arr2)
5. out_arr = np.bitwise_and(arr1, arr2)
6. print("Output array after bitwise_and: ", out_arr)
```

```
Input array1 :  [2, 8, 125]
Input array2 :  [3, 3, 115]
Output array after bitwise_and:  [  2   0 113]
```

numpy.bitwise_or() function is used to compute the bit-wise OR of two array element-wise. This function computes the bit-wise OR of the underlying binary representation of the integers in the input arrays.

```
1. arr1 = [2, 8, 125]
2. arr2 = [3, 3, 115]
3. print ("Input array1 : ", arr1)
4. print ("Input array2 : ", arr2)
5. out_arr = np.bitwise_or(arr1, arr2)
6. print("Output array after bitwise_or: ", out_arr)
```

```
Input array1 :  [2, 8, 125 ] ]                                  Input
array2 :  [3, 3, 115 ]
Output array after bitwise_or:  [  3  11 127]
```

numpy.bitwise_xor() function is used to compute the bit-wise XOR of two array element-wise. This function computes the bit-wise XOR of the underlying binary representation of the integers in the input arrays.

```
1. arr1 = [2, 8, 125]
2. arr2 = [3, 3, 115]

3. print ("Input array1 : ", arr1)
4. print ("Input array2 : ", arr2)

5. out_arr = np.bitwise_xor(arr1, arr2)
6. print ("Output array after bitwise_xor: ", out_arr)
```

```
Input array1 :  [2, 8, 125]                                     Input
array2 :  [3, 3, 115]                                  Output array
after bitwise_xor:  [ 1 11 14]
```

numpy.invert() function is used to compute the bit-wise Inversion of an array element-wise. It computes the bit-wise NOT of the underlying binary representation of the integers in the input arrays.

```
1. arr1 = [2, 8, 125]
2. print ("Input array1 : ", arr1)
3. out_arr = np.invert(arr1)
4. print ("Output array after invert: ", out_arr)
```

```
Input array1 :  [2, 8, 125]                                    Output
array after invert:  [  -3   -9 -126]
```

Indexing NumPy array

Slicing is of the form a[x:y] means that output all the elements from xth to y-1th index.

```
1. # Arange elements from 0 to 19
2. a = np.arange(20)print("Array is:",a)
3. # a[start:stop]
4. print("\na[3:7] = ", a[3:7])
   # The : operator means all elements till the end
5. print ("\na[10:] = ",a[10:])
    # Negative integer means start from the end. so -1th index will
   actually be the last element in array
6. print ("\na[-3:-7] = ", a[-7:-3])
```

```
Array is: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
18 19]                                                 a[3:7] =
[3 4 5 6]                                        a[10:] =  [10 11 12
13 14 15 16 17 18 19]                 a[-3:-7] =  [13 14 15 16]
```

Sorting and Searching

Using dstack we combine two columns 'Proteins' and 'Name' so that we know the contents of each beverage.

```
1. a = np.array(data['Protein'], dtype=int)
2. b = np.array(data['Name'
3. np1 = np.dstack((a,b))
4. np.shape(np1)
```

```
(1, 84, 2)
```

As we see that we our array has become 3 dimensional so we reshape it using reshape() for sorting.

```
1. np1 = np1.reshape(84,2)
2. np1
```

```
array([[0, 'Cool Lime Starbucks Refreshers™ Beverage'],
       [0, 'Strawberry Acai Starbucks Refreshers™ Beverage'],
       [0, 'Very Berry Hibiscus Starbucks Refreshers™ Beverage'],
       [0, 'Evolution Fresh™ Organic Ginger Limeade'],
       [0, 'Iced Coffee'],
       [5, 'Iced Espresso Classics - Vanilla Latte'],
       [5, 'Iced Espresso Classics - Caffe Mocha'],
       [5, 'Iced Espresso Classics - Caramel Macchiato'],
       [0, 'Shaken Sweet Tea'],
       [0, 'Tazo® Bottled Berry Blossom White'],
       [0, 'Tazo® Bottled Black Mango'],
       [0, 'Tazo® Bottled Black with Lemon'],
       [0, 'Tazo® Bottled Brambleberry'],
       [0, 'Tazo® Bottled Giant Peach'],
       [0, 'Tazo® Bottled Iced Passion'],
       [0, 'Tazo® Bottled Lemon Ginger'],
       [0, 'Tazo® Bottled Organic Black Lemonade'],
       [0, 'Tazo® Bottled Organic Iced Black Tea'],
       [0, 'Tazo® Bottled Organic Iced Green Tea'],
       [0, 'Tazo® Bottled Plum Pomegranate'],
       [0, 'Tazo® Bottled Tazoberry'],
       [0, 'Tazo® Bottled White Cranberry'],
       [0, 'Teavana® Shaken Iced Black Tea'],
       [0, 'Teavana® Shaken Iced Black Tea Lemonade'],
       [0, 'Teavana® Shaken Iced Green Tea'],
       [0, 'Teavana® Shaken Iced Green Tea Lemonade'],
       [0, 'Teavana® Shaken Iced Passion Tango™ Tea'],
       [0, 'Teavana® Shaken Iced Passion Tango™ Tea Lemonade'],
       [0, 'Teavana® Shaken Iced Peach Green Tea'],
       [5, 'Iced Espresso Classics - Vanilla Latte'],
       [5, 'Iced Espresso Classics - Caffe Mocha'],
       [5, 'Iced Espresso Classics - Caramel Macchiato'],
       [0, 'Starbucks Refreshers™ Raspberry Pomegranate'],
       [0, 'Starbucks Refreshers™ Strawberry Lemonade'],
       [20, 'Starbucks® Doubleshot Protein Dark Chocolate'],
       [20, 'Starbucks® Doubleshot Protein Vanilla'],
       [1, 'Starbucks® Iced Coffee Caramel'],
       [1, 'Starbucks® Iced Coffee Light Sweetened'],
       [1, 'Starbucks® Iced Coffee Unsweetened'],
       [0, 'Tazo® Bottled Berry Blossom White'],
       [0, 'Tazo® Bottled Black Mango'],
```

To sort the fats column, we use sort() function.

```
1. np1 = np.array(data['Fat (g)'], dtype=int)
2. np.sort(np1)
```

```
array([ 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,  0,  0,        0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,        0,  0,  2,  2,  2,  2,  2,  2,
        2,  2,  4,  4,  4,  4,  4,  5,  6,        6,  6,  6,  7,  7,  7,
        7,  7,  7,  8,  8,  9,  9,  9, 11, 26])
```

As we can see that the maximum amount of fat content is 26 units. So we try to find the beverage with maiximum amount of fat content using searching. Searching is done with the help of where() function.
First we will dstack the array so that we associate the fat content with their respective drink.

```
1. np1 = np.array(data['Fat (g)'], dtype=int)
2. np2 = np.array(data['Name'])
3. np3 = np.dstack((np1, np2))
```

Now we search the row which has fat=26 grams

```
1. i = np.where(np3 == 26)
2. i
```

```
(array([0]), array([64]), array([0]))
```

Thus, we get the location at (0,64)

```
1. np3[0,64]
```

```
array([26, 'Starbucks® Signature Hot Chocolate'], dtype=object)
```

Conclusion

Thus, we conclude that the Starbucks' Signature Hot Chocolate contains the highest fat content. So now you can perform similar operations to check various contents in other drinks.