17. Numpy

October 27, 2022

1 NumPy

- At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.
- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

1.1 Why is NumPy Fast?

- Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just "behind the scenes" in optimized, pre-compiled C code. Vectorized code has many advantages, among which are:
  - vectorized code is more concise and easier to read
  - fewer lines of code generally means fewer bugs
  - the code more closely resembles standard mathematical notation (making it easier, typically, to correctly code mathematical constructs)
  - vectorization results in more "Pythonic" code. Without vectorization, our code would be littered with inefficient and difficult to read for loops.

2 Install Numpy (in your Virtual Environment)

```
pip install numpy
```

[1]:
```
import numpy as np
```

[2]:
```
arr = np.array([1, 2, 3, 4, 5]) #pass list or tuple
```

[3]:
```
print(arr)
```

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

```
[4]: type(arr)
```

```
[4]: numpy.ndarray
```

```
[5]: len(arr)
```

```
[5]: 5
```

```
[6]: arr.shape
```

```
[6]: (5,)
```

```
[7]: arr.ndim
```

```
[7]: 1
```

```
[8]: arr2d = np.array([[1, 2, 3], [4, 5, 6]]) # 2d arrays
```

```
[9]: arr2d
```

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

```
[10]: arr2d.shape #  array has 2 dimensions, and each dimension has 3 elements.
```

```
[10]: (2, 3)
```

```
[11]: arr2d.ndim
```

```
[11]: 2
```

```
[12]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]]) # 3d arrays
```

```
[13]: arr3d
```

```
[13]: array([[[1, 2, 3],
             [4, 5, 6]],

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

```
[14]: arr3d.ndim
```

```
[14]: 3
```

```
[15]: arr3d.shape # The shape of an array is the number of elements in each dimension.
```

[15]:
```
(2, 2, 3)
```

- Access array elements using indexing.
- Also slicing can be performed on arrays.

3 Array Indexing

3.1 Access 1-D Arrays

[16]:
```
arr = np.array([1, 2, 3, 4])
```

[17]:
```
print(arr[0])
```

```
1
```

[18]:
```
arr[1]
```

[18]:
```
2
```

3.2 Access 2-D Arrays

[19]:
```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
```

[20]:
```
print('2nd element on 1st row: ', arr[0, 1])
```

```
2nd element on 1st row:  2
```

[21]:
```
arr[0][1]
```

[21]:
```
2
```

3.3 Access 3-D Arrays

[22]:
```
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

[24]:
```
arr
```

[24]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

[23]:
```
arr[0, 1, 2]
```

[23]:
```
6
```

```
[25]: arr[0][1][2]
```

```
[25]: 6
```

3.4 Negative Indexing

```
[26]: arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
```

```
[27]: print('Last element from 2nd dim: ', arr[1, -1])
```

```
Last element from 2nd dim:  10
```

4 Array Slicing

- Slicing in python means taking elements from one given index to another given index.
- We pass slice instead of index like this: `[start:end+1]`.
- We can also define the step, like this: `[start:end:step]`.

```
[28]: arr = np.array([1, 2, 3, 4, 5, 6, 7])
```

```
[29]: arr[1:5]
```

```
[29]: array([2, 3, 4, 5])
```

4.1 Negative Slicing

- Use the minus operator to refer to an index from the end

```
[30]: arr = np.array([1, 2, 3, 4, 5, 6, 7])
```

```
[31]: print(arr[-3:-1])
```

```
[5 6]
```

5 Step

- Use the step value to determine the step of the slicing:

```
[32]: arr = np.array([1, 2, 3, 4, 5, 6, 7])
```

```
[34]: arr
```

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

```
[33]: print(arr[1:5:2])
```

```
[2 4]
```

5.1 Slicing 2-D Arrays

```
[35]: arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
```

```
[37]: arr
```

```
[37]: array([[ 1,  2,  3,  4,  5],
             [ 6,  7,  8,  9, 10]])
```

```
[36]: print(arr[1, 1:4])
```

```
[7 8 9]
```

```
[38]: arr[0:2, 2]
```

```
[38]: array([3, 8])
```

```
[39]: arr
```

```
[39]: array([[ 1,  2,  3,  4,  5],
             [ 6,  7,  8,  9, 10]])
```

```
[40]: arr[0:2, 1:4]
```

```
[40]: array([[2, 3, 4],
             [7, 8, 9]])
```

```
[41]: arr.dtype
```

```
[41]: dtype('int64')
```

6 Reshaping arrays

- Reshaping means changing the shape of an array.
- The shape of an array is the number of elements in each dimension.
- By reshaping we can add or remove dimensions or change number of elements in each dimension.

6.1 Reshape From 1-D to 2-D

```
[42]: arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

```
[43]: newarr = arr.reshape(4, 3)
```

```
[44]: print(newarr)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

6.2 Reshape From 1-D to 3-D

[45]:
```
newarr = arr.reshape(2, 3, 2)
```

[46]:
```
newarr
```

[46]:
```
array([[[ 1,  2],
        [ 3,  4],
        [ 5,  6]],

       [[ 7,  8],
        [ 9, 10],
        [11, 12]]])
```

6.3 Can We Reshape Into any Shape?

- Yes, as long as the elements required for reshaping are equal in both shapes.
- We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

6.4 Unknown Dimension

- You are allowed to have one "unknown" dimension.
- Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.
- Pass -1 as the value, and NumPy will calculate this number for you.

[47]:
```
arr
```

[47]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

[48]:
```
newarr = arr.reshape(2, 2, -1)
```

[49]:
```
print(newarr)
```

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

 [[ 7  8  9]
  [10 11 12]]]
```

6.5 Flattening the arrays

- Flattening array means converting a multidimensional array into a 1D array.

- We can use `reshape(-1)` to do this.

```
[52]: newarr
```

```
[52]: array([[[ 1,  2,  3],
              [ 4,  5,  6]],

             [[ 7,  8,  9],
              [10, 11, 12]]])
```

```
[50]: newarr2 = newarr.reshape(-1)
```

```
[51]: newarr2
```

```
[51]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

7 Joining Arrays

- Joining means putting contents of two or more arrays in a single array.
- In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.
- We pass a sequence of arrays that we want to join to the `concatenate()` function, along with the `axis`.
- If axis is not explicitly passed, it is taken as `0`.

```
[53]: arr1 = np.array([1, 2, 3])
      arr2 = np.array([4, 5, 6])
```

```
[54]: arr = np.concatenate((arr1, arr2))
```

```
[55]: arr
```

```
[55]: array([1, 2, 3, 4, 5, 6])
```

```
[56]: arr1 = np.array([[1, 2], [3, 4]])
      arr2 = np.array([[5, 6], [7, 8]])
```

```
[57]: arr1
```

```
[57]: array([[1, 2],
             [3, 4]])
```

```
[58]: arr2
```

```
[58]: array([[5, 6],
             [7, 8]])
```

```
[59]: arr_a1 = np.concatenate((arr1, arr2), axis=1)
```

```
[61]: arr1
```

```
[61]: array([[1, 2],
             [3, 4]])
```

```
[62]: arr2
```

```
[62]: array([[5, 6],
             [7, 8]])
```

```
[60]: arr_a1
```

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

```
[63]: arr_a0 = np.concatenate((arr1, arr2), axis=0)
```

```
[64]: arr_a0
```

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

7.1 Joining Arrays Using Stack Function

- Stacking is same as concatenation, the only difference is that stacking is done along a new axis.
- We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.
- We pass a sequence of arrays that we want to join to the `stack()` method along with the axis. If axis is not explicitly passed it is taken as 0.

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

```
[ ]: arr = np.stack((arr1, arr2), axis=1)
```

```
[ ]: print(arr)
```

7.1.1 Stacking Along Rows

```
[ ]: arr = np.hstack((arr1, arr2))
```

```
[ ]: print(arr)
```

7.1.2 Stacking Along Columns

```
[ ]: arr = np.vstack((arr1, arr2))
```

```
[ ]: arr
```

7.1.3 Stacking Along Height (depth)

```
[ ]: arr = np.dstack((arr1, arr2))
```

```
[ ]: arr
```

8 Splitting NumPy Arrays

- We use `array_split()` for splitting arrays, we pass it the array we want to split and the number of splits.

```
[65]: arr = np.array([1, 2, 3, 4, 5, 6])
```

```
[66]: newarr = np.array_split(arr, 3)
```

```
[67]: newarr
```

```
[67]: [array([1, 2]), array([3, 4]), array([5, 6])]
```

```
[68]: # If the array has less elements than required, it will adjust from the end␣
        ↪accordingly.
      np.array_split(arr, 4)
```

```
[68]: [array([1, 2]), array([3, 4]), array([5]), array([6])]
```

8.1 Splitting 2-D Arrays

```
[69]: arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
```

```
[70]: np.array_split(arr, 3)
```

```
[70]: [array([[1, 2],
              [3, 4]]),
       array([[5, 6],
              [7, 8]]),
       array([[ 9, 10],
              [11, 12]])]
```

```
[71]: arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15],␣
        ↪[16, 17, 18]])
```

[72]:
```
np.array_split(arr, 3)
```

[72]:
```
[array([[1, 2, 3],
        [4, 5, 6]]),
 array([[ 7,  8,  9],
        [10, 11, 12]]),
 array([[13, 14, 15],
        [16, 17, 18]])]
```

[73]:
```
# Split the 2-D array into three 2-D arrays along rows.
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15],␣
 ↪[16, 17, 18]])
```

[74]:
```
np.array_split(arr, 3, axis=1)
```

[74]:
```
[array([[ 1],
        [ 4],
        [ 7],
        [10],
        [13],
        [16]]),
 array([[ 2],
        [ 5],
        [ 8],
        [11],
        [14],
        [17]]),
 array([[ 3],
        [ 6],
        [ 9],
        [12],
        [15],
        [18]])]
```

[75]:
```
np.hsplit(arr, 3)
```

[75]:
```
[array([[ 1],
        [ 4],
        [ 7],
        [10],
        [13],
        [16]]),
 array([[ 2],
        [ 5],
        [ 8],
        [11],
        [14],
        [17]]),
 array([[ 3],
        [ 6],
        [ 9],
        [12],
        [15],
        [18]])]
```

9 Some functions in numpy

9.1 arange()

- Return evenly spaced values within a given interval.

[76]:
```
arr = np.arange(10)
```

[77]:
```
arr
```

[77]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

9.2 linspace()

- Function returns evenly spaced numbers over a specified interval defined by the first two arguments of the function (start and stop)

[78]:
```
np.linspace(10,20, num=100)
```

[78]:
```
array([10.        , 10.1010101 , 10.2020202 , 10.3030303 , 10.4040404 ,
       10.50505051, 10.60606061, 10.70707071, 10.80808081, 10.90909091,
       11.01010101, 11.11111111, 11.21212121, 11.31313131, 11.41414141,
       11.51515152, 11.61616162, 11.71717172, 11.81818182, 11.91919192,
       12.02020202, 12.12121212, 12.22222222, 12.32323232, 12.42424242,
       12.52525253, 12.62626263, 12.72727273, 12.82828283, 12.92929293,
       13.03030303, 13.13131313, 13.23232323, 13.33333333, 13.43434343,
       13.53535354, 13.63636364, 13.73737374, 13.83838384, 13.93939394,
       14.04040404, 14.14141414, 14.24242424, 14.34343434, 14.44444444,
       14.54545455, 14.64646465, 14.74747475, 14.84848485, 14.94949495,
       15.05050505, 15.15151515, 15.25252525, 15.35353535, 15.45454545,
       15.55555556, 15.65656566, 15.75757576, 15.85858586, 15.95959596,
       16.06060606, 16.16161616, 16.26262626, 16.36363636, 16.46464646,
       16.56565657, 16.66666667, 16.76767677, 16.86868687, 16.96969697,
       17.07070707, 17.17171717, 17.27272727, 17.37373737, 17.47474747,
       17.57575758, 17.67676768, 17.77777778, 17.87878788, 17.97979798,
       18.08080808, 18.18181818, 18.28282828, 18.38383838, 18.48484848,
       18.58585859, 18.68686869, 18.78787879, 18.88888889, 18.98989899,
       19.09090909, 19.19191919, 19.29292929, 19.39393939, 19.49494949,
       19.5959596 , 19.6969697 , 19.7979798 , 19.8989899 , 20.        ])
```

9.3 `repeat()`

- function repeats the elements of an array. The number of repetitions is specified by the second argument repeats.

```
[79]: np.repeat(3,5)
```

```
[79]: array([3, 3, 3, 3, 3])
```

9.4 `random.randint()`

- Returns random integers from the interval

```
[96]: np.random.randint(10,100, size=2)
```

```
[96]: array([50, 60])
```

9.5 `argmax()`

- Returns the indices of the maximum value along an axis.

```
[81]: arr = np.array([8,2,4,0,6,9,7])
```

```
[82]: np.argmax(arr)
```

```
[82]: 5
```

10 Operations on arrays

```
[98]: arr1 = np.array([1,2,3,4])
      arr2 = np.array([1,2,3,4])
```

```
[99]: arr1
```

```
[99]: array([1, 2, 3, 4])
```

```
[100]: arr2
```

```
[100]: array([1, 2, 3, 4])
```

```
[103]: arr1 + arr2
```

```
[103]: array([2, 4, 6, 8])
```

```
[104]: arr1 - arr2
```

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

```
[105]: arr1 * arr2
```

```
[105]: array([ 1,  4,  9, 16])
```

There are many more functions available in numpy package. Explore them!!!

https://www.machinelearningplus.com/101-numpy-exercises-python/