Advanced_Numpy_CheatSheet_From_Beginner_to_Pro_by_Aliza_Mustaf

October 28, 2022

What is Numpy?

Numpy is a Python library for creating n-dimensional arrays.

Why to use Numpy?

Numpy structure look similar to standard Python lists but Numpy arrays are more efficient e.g. 1. Its broadcasting capabilities are extremely useful for quickly applying functions to any dataset. 2. It has alot of built-in features like linear algebra, statistical distribution, trignometry and random numbers handling capabilities.

How many methods are there to create Numpy arrays?

There are three methods for it:

1. Getting Numpy arrays by transforming python lists to arrays
2. Getting Numpy arrays by using builtin function
3. Getting Numpy arrays by generating random data in whatever shape you want

Method-1:

[ ]:
```
# Must import numpy library before using it
import numpy as np

# 1D Numpy Array ----->>>>
mylist = [1,2,3]   # creating the list
# transforming the mylist to numpy array as below
np.array(mylist)
```

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

[ ]:
```
#directly writing mylist into np.array() function gives same result
np.array([1,2,3])
```

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

[ ]:
```
# 2D Numpy Array ----->>>>
mymatrix = [[1,2,3], [5,6,7], [8,9,10]]
np.array(mymatrix)
```

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

```
[ ]: #directly writing 2D list(matrix) into np.array() function gives same result
     np.array([[1,2,3], [5,6,7], [8,9,10]])
```

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

Method-2:

1. Using arange function:

Syntax: np.arange(start, stop, step)

Nota bene:

1. It always produces integers in return 2.'Stop' number is not included in result
2. It always gives 1D in return. To make it 2D, reshape it.
3. np.arange(0,9) = np.arange(9)

```
[ ]: np.arange(0, 10, 1)
```

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

```
[ ]: np.arange(0, 10, 2)
```

```
[ ]: array([0, 2, 4, 6, 8])
```

2. Producing array of zeros and ones with np.zeros and np.ones function:

Syntax: 1. np.zeros(shape) 2. np.ones(shape)

Nota bene: Shape = ((Number of rows, Number of columns)) : must have double brackets if you want to produce 2D array.

```
[ ]: np.zeros(5) # To get 1D array, Shape = Number of zeros needed in 1D array
```

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

```
[ ]: np.zeros((4,5))   # Shape = (4,5) = (4 rows, 5 columns)
```

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

```
[ ]: np.ones(5)
[ ]: array([1., 1., 1., 1., 1.])
```

```
[ ]: np.ones((4,5))
```

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

3. Using linspace function:

It gives n number of evenly spaced numbers in range min_limit to max_limit

Syntax: np.linspace(min_limit, max_limit, n)

Nota bene:

1. 'max_limit' number is included in result
2. It produces floats in return e.g. 0., 1. etc.

```
[ ]: np.linspace(0,10,3)
```

```
[ ]: array([ 0.,  5., 10.])
```

```
[4]: np.linspace(0,10,11)
```

```
[4]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

4. Using eye function to get identity matrix:

Syntax: np.eye(shape)

Nota bene: Shape = (n) : Number of rows = n and number of columns = n

```
[ ]: np.eye(5)
```

```
[ ]: array([[1., 0., 0., 0., 0.],
            [0., 1., 0., 0., 0.],
            [0., 0., 1., 0., 0.],
            [0., 0., 0., 1., 0.],
            [0., 0., 0., 0., 1.]])
```

Method-3:

1. Using np.random.rand() function:

Syntax: np.random.rand(shape)

Nota bene: 1. Shape = ((Number of rows, Number of columns)) : must have double brackets 2. All generated numbers lie between 0 & 1

```
[ ]: np.random.rand(5) # To get 1D array of random numbers, Shape = Number of random␣
     ↪integers needed in 1D array
```

```
[ ]: array([0.19094959, 0.17330918, 0.77339487, 0.50050293, 0.47724927])
```

```
[ ]: np.random.rand(4,5)
```

```
[ ]: array([[0.11160308, 0.01451946, 0.69295744, 0.94286861, 0.79286264],
            [0.83949239, 0.04591028, 0.37970204, 0.95892099, 0.37398896],
            [0.36004257, 0.80366683, 0.44411184, 0.70100194, 0.20929319],
            [0.11235763, 0.02314392, 0.99399227, 0.78890615, 0.22399345]])
```

2. Using np.random.randn() function:

It gives standard normally distributed random integers. It means that the variance of all produced random inetgers = 1 and mean = 1.

Syntax: np.random.randn(shape)

Nota bene: 1. Shape = (Number of rows, Number of columns) 2. All generated random numbers lie between -1 & 1 (because we can have mean = 0 only if we have negative numbers with positive numbers to give us 0 sum so thats why limit is -1 to +1)

```
[ ]: np.random.randn(10)
```

```
[ ]: array([ 0.27497129, -0.58232082,  0.02950219, -2.94568226,  0.58800858,
           -0.74145297, -0.29757586, -0.18415128,  1.95819524,  0.47188413])
```

```
[ ]: np.random.randn(4,5)
```

```
[ ]: array([[ 1.40578371,  0.05956482, -0.98894187,  1.3074693 , -0.82914767],
            [-0.26557385,  1.08353995, -2.60316621, -0.65912286, -1.51705903],
            [-1.69939497,  0.80169912,  1.09999835, -0.96117025,  0.46186788],
            [ 0.47898847,  1.30347   , -0.11065359, -0.4481102 ,  0.15262056]])
```

2. Using np.random.randint() function:

Syntax: np.random.randint(min_limit, max_limit, shape)

Nota bene: 1. Shape = (Number of rows, Number of columns) for 2D 2. 'max_limit' number is not included in result 3. It works like np.linspace(min_limit, max_limit, shape) but the only difference is: we can give shape like (3,4) etc. in np.random.randint to generate 2D arrays but not in np.linspace. The np.linspace just needs one number as a shape and always produces 1D array.

```
[ ]: np.random.randint(0, 10, 4) # To get 1D array of random integers, Shape =␣
      ↪Number of random numbers needed in 1D array
```

```
[ ]: array([4, 1, 7, 9])
```

```
[ ]: np.random.randint(0, 10, (2,3))
```

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

Why we need to use seed in numpy?

The seed is used to set a random state so that the random results can actually be reproduced. Every time we give it the same seed, it generates the same numbers.

```
[ ]: np.random.seed(42)
     np.random.rand(5)
```

```
[ ]: array([0.37454012, 0.95071431, 0.73199394, 0.59865848, 0.15601864])
```

```
[ ]: # Reproducing the same result
     np.random.seed(42)
     np.random.rand(5)
```

```
[ ]: array([0.37454012, 0.95071431, 0.73199394, 0.59865848, 0.15601864])
```

Nota bene: 1. Never forget to write *np.random.seed(42)* on top each time you want to reproduce the result 2. 42 is default number in python to use as seed because it was mentioned in book 'Hitchhiker's Guide to the Galaxy' of python. you can use number '101' etc. also.

0.1 Useful Attributes and Methods calls in Numpy:

```
[ ]: firstarray = np.arange(0,25) # here step = 1. By default, step = 1 so not␣
      ↪mentioning doesn't make any difference.
     print(firstarray)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24]
```

```
[ ]: firstarray.reshape(5,5)
```

```
[ ]: array([[ 0,  1,  2,  3,  4],
            [ 5,  6,  7,  8,  9],
            [10, 11, 12, 13, 14],
            [15, 16, 17, 18, 19],
            [20, 21, 22, 23, 24]])
```

```
[ ]: second_array = np.arange(1,40,2)
     second_array.reshape(5,4)
```

```
[ ]: array([[ 1,  3,  5,  7],
            [ 9, 11, 13, 15],
            [17, 19, 21, 23],
            [25, 27, 29, 31],
            [33, 35, 37, 39]])
```

Reshaping into wrong dimensions ---> It will raise error

[ ]:
```
# reshaping into wrong dimensionality will raise error. For example, for above␣
↪example, myarray.reshape(4,5) will raise error as 4 x 5 != 25

firstarray.reshape(4,5)
```

```
      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-15-901062155721> in <module>
         1 # reshaping into wrong dimensionality will raise error. For example,␣
↪for above example, myarray.reshape(4,5) will raise error as 4 x 5 != 25
         2
   ----> 3 firstarray.reshape(4,5)


       ValueError: cannot reshape array of size 25 into shape (4,5)
```

[ ]:
```
# to get maximum number from array
firstarray.max()
```

[ ]:
```
24
```

[ ]:
```
# to get minimum number from array
firstarray.min()
```

[ ]:
```
0
```

[ ]:
```
# to get index of maximum number from array
firstarray.argmax()
```

[ ]:
```
24
```

[ ]:
```
# to get index of minimum number from array
firstarray.argmin()
```

[ ]:
```
0
```

[ ]:
```
# to check data type of elements in array
firstarray.dtype
```

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

```
[ ]: # to check data type of array: to check if the array is a list or its a numpy␣
      ↪array?
     type(firstarray)
```

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

```
[ ]: # to check shape of the array
     firstarray.shape
```

```
[ ]: (25,)
```

Nota bene: (25,) and (25,1) are different.

- (25,) means 25 columns and 1 row
- (25,1) means 25 rows and 1 column

```
[ ]: np.random.rand(5,)
```

```
[ ]: array([0.04645041, 0.60754485, 0.17052412, 0.06505159, 0.94888554])
```

```
[ ]: np.random.rand(5,1)
```

```
[ ]: array([[0.96563203],
            [0.80839735],
            [0.30461377],
            [0.09767211],
            [0.68423303]])
```

```
[ ]: # Indexing
     firstarray[2]
```

```
[ ]: 2
```

```
[ ]: # Slicing
     firstarray[1:3]
```

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

```
[ ]: firstarray[:4]
```

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

```
[ ]: firstarray[3:]
```

```
[ ]: array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
            20, 21, 22, 23, 24])
```

0.2 Broadcasting:

Numpy is different from the normal Python list due to its ability of broadcasting. We can broadcast a single value across a larger set of values in numpy.

```
[ ]: firstarray[0:5]
```

```
[ ]: array([0, 1, 2, 3, 4])
```

```
[ ]: firstarray[0:5] = 100
```

```
[ ]: firstarray
```

```
[ ]: array([100, 100, 100, 100, 100,   5,   6,   7,   8,   9,  10,  11,  12,
             13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24])
```

Hurrah ! Broadcasting worked !!! We can't do this with normal python lists. This is called **broadcasting reassignment**.

0.3 ** ---- Hold on (0_0) ----** What if we just want to take slice of array and do operations on that but not on original array?

Solution ---> Use .copy() method. Don't copy directly like did above, in case you don't want to modify your original array. If we do it directly like above, it doesn't copy the array. It just creates a pointer to the original array.

```
[ ]: array_copy = firstarray.copy()
     array_copy[:] = 90
     print(array_copy)
```

```
[90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
 90]
```

```
[ ]: firstarray
```

```
[ ]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
            17, 18, 19, 20, 21, 22, 23, 24])
```

0.4 Conditional Selection:

```
[ ]: myarray = np.arange(1,11)
     print(myarray)
```

```
[ ]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
[ ]: # to check the values greater than 4 in the array
     myarray > 4
```

```
[ ]: array([False, False, False, False,  True,  True,  True,  True,  True,
             True])
```

```
[ ]: # to get the actual values of array instead of booleans
     myarray[myarray > 4]
```

```
[ ]: array([ 5,  6,  7,  8,  9, 10])
```

0.5 Numpy Operations:

```
[ ]: lastarray = np.arange(0,10)
     print(lastarray)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
[ ]: # adding same number in all elements of array
     lastarray + 5
```

```
[ ]: array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
[ ]: lastarray - 2
```

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

```
[ ]: lastarray * lastarray
```

```
[ ]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

```
[ ]: lastarray / lastarray
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: RuntimeWarning:
divide by zero encountered in true_divide
  """Entry point for launching an IPython kernel.
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: RuntimeWarning:
invalid value encountered in true_divide
  """Entry point for launching an IPython kernel.
```

```
[ ]: array([nan, inf, inf, inf, inf, inf, inf, inf, inf, inf])
```

We just caught **nan** value in case of diving array. So, just few things to remember, in Numpy:

1. 0/0 = nan
2. scalar / 0 = inf (infinity)

Example is below to test:

```
[ ]: lastarray / 0
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: RuntimeWarning:
divide by zero encountered in true_divide
  """Entry point for launching an IPython kernel.
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: RuntimeWarning:
invalid value encountered in true_divide
  """Entry point for launching an IPython kernel.
```

```
[ ]: array([nan, inf, inf, inf, inf, inf, inf, inf, inf, inf])
```

```
[ ]: # to take sqaure root of all values
     np.sqrt(lastarray)
```

```
[ ]: array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
            2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

```
[ ]: np.log(lastarray) # log(0) = -inf (in Numpy)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: RuntimeWarning:
divide by zero encountered in log
  """Entry point for launching an IPython kernel.
```

```
[ ]: array([      -inf, 0.        , 0.69314718, 1.09861229, 1.38629436,
            1.60943791, 1.79175947, 1.94591015, 2.07944154, 2.19722458])
```

```
[ ]: lastarray.sum()
```

```
[ ]: 45
```

```
[ ]: lastarray.mean()
```

```
[ ]: 4.5
```

```
[ ]: # To get sum of all rows in 2D Array in Numpy:

     last2darray = np.random.randint(0,20,(4,5))
     print(last2darray)
```

```
[[ 4 10 15  0  4]
 [16 11 15  8  4]
 [17 10  8  6 10]
 [ 2  2 14 11  9]]
```

```
[ ]: # to get sum of all column: use axis = 0
     last2darray.sum(axis=0)     # 4 + 16 + 17 + 2 = 39 and so on.
```

```
[ ]: array([39, 33, 52, 25, 27])
```

```
[ ]: # to get sum of all rows: : use axis = 1
     last2darray.sum(axis=1)      # 4 + 10 + 15 + 0 + 4 = 33 and so on.
```

```
[ ]: array([33, 54, 51, 38])
```

0.6 Hope you enjoyed both flavours of Numpy: vectors (1D) and Arrays (nD)

Keep it to your computer to use it for any Numpy related functions. Thanks.