Numpy

February 1, 2018

1 Chapter 2. Numpy

In [ ]:
```
import numpy as np

        # load data from txt file
        world_alcohol = np.genfromtxt("world_alcohol.txt", delimiter=",", dtype=str)
        print(type(world_alcohol))
        print(world_alcohol)
```

```
<class 'numpy.ndarray'>
[['Year' 'WHO region' 'Country' 'Beverage Types' 'Display Value']
 ['1986' 'Western Pacific' 'Viet Nam' 'Wine' '0']
 ['1986' 'Americas' 'Uruguay' 'Other' '0.5']
 ...
 ['1987' 'Africa' 'Malawi' 'Other' '0.75']
 ['1989' 'Americas' 'Bahamas' 'Wine' '1.5']
 ['1985' 'Africa' 'Malawi' 'Spirits' '0.31']]
```

In [2]:
```
        # create a one-Demension matrix from a list
        vector = np.array([5, 10, 15, 20])

        #create a two-Demension matrix from a 2-D list
        matrix = np.array([[5, 10, 15], [20, 25, 30], [35, 40, 45]])

        print(vector)
        print(matrix)
```

```
[ 5 10 15 20]
[[ 5 10 15]
 [20 25 30]
 [35 40 45]]
```

In [3]:
```
        #numpy ndarray shapes
        vector = np.array([5, 10, 15, 20])
        print(vector.shape)
        matrix = np.array([[5, 10, 15], [20, 25, 30]])
        print(matrix.shape)
```

```
(4,)
(2, 3)
```

```
In [4]: # in the numpy array, only one datatype is allowed
        # different data type will be normalized to the same one

        nums = np.array([1, 2, 3, 4])
        print(nums)
        print(nums.dtype)

        nums = np.array([1, 2, 3, 4.0])
        print(nums)
        print(nums.dtype)

        nums = np.array([1, 2, 3, '4'])
        print(nums)
        print(nums.dtype)
```

```
[1 2 3 4]
int64
[1. 2. 3. 4.]
float64
['1' '2' '3' '4']
<U21
```

```
In [5]: # Load the data according to the index of the array
        world_alcohol = np.genfromtxt("world_alcohol.txt", delimiter=",",
                                      dtype=str, skip_header=1)
        print(world_alcohol)
        print('----------------------------')
        uruguay_other_1986 = world_alcohol[1, 4]
        third_country = world_alcohol[2, 2]

        print(uruguay_other_1986)
        print(third_country)
```

```
[['1986' 'Western Pacific' 'Viet Nam' 'Wine' '0']
 ['1986' 'Americas' 'Uruguay' 'Other' '0.5']
 ['1985' 'Africa' "Cte d'Ivoire" 'Wine' '1.62']
 ...
 ['1987' 'Africa' 'Malawi' 'Other' '0.75']
 ['1989' 'Americas' 'Bahamas' 'Wine' '1.5']
 ['1985' 'Africa' 'Malawi' 'Spirits' '0.31']]
----------------------------
0.5
Cte d'Ivoire
```

```
In [6]: # cut the array
        vector = np.array([5, 10, 15, 20])
        print(vector[:2])
```

```
[ 5 10]
```

```
In [7]: # cut the matrix
        matrix = np.array([[5, 10, 15],
                           [20, 25, 30],
                           [35, 40, 45]])

        # matrix(row,column,index)

        # single column:
        print(matrix[:, 1])

        # double columns:
        print(matrix[:, :2])

        # some row and some column
        print(matrix[1:, :2])
```

```
[10 25 40]
[[ 5 10]
 [20 25]
 [35 40]]
[[20 25]
 [35 40]]
```

```
In [8]: # judgement in array
        vector = np.array([5, 10, 15, 20])
        print(vector == 10)
        print('----------------------------')
        # judgement in matrix
        matrix = np.array([[5, 10, 15], [20, 25, 30], [25, 40, 45]])
        print(matrix == 25)
        print(matrix[matrix == 25])
```

```
[False  True False False]
----------------------------
[[False False False]
 [False  True False]
 [ True False False]]
[25 25]
```

```
In [9]: # Judgement, and return a row that contain certain values
        matrix = np.array([[5, 10, 15], [20, 25, 30], [35, 40, 45]])
```

```
        second_column = (matrix[:, 1] == 25)
        print(second_column)
        print(matrix[second_column, :])
```

```
[False  True False]
[[20 25 30]]
```

```
In [10]: # Logic operation
         vector = np.array([5, 10, 15, 20])
         print((vector == 5) & (vector == 10))
         print((vector == 5) | (vector == 10))
```

```
[False False False False]
[ True  True False False]
```

```
In [11]: # conditional change value, using true/false as a index
         vector = np.array([5, 10, 15, 20])
         equal_to_ten_or_five = (vector == 10) | (vector == 5)
         vector[equal_to_ten_or_five] = 50
         print(vector)
         print('----------------------------')

         matrix = np.array([[5, 10, 15],
                            [20, 25, 30],
                            [35, 40, 45]])

         second_column = (matrix[:, 1] == 25)
         print(second_column)
         matrix[second_column, :] = 100
         print(matrix)
```

```
[50 50 15 20]
----------------------------
[False  True False]
[[  5  10  15]
 [100 100 100]
 [ 35  40  45]]
```

```
In [12]: vector = np.array(['1', '2', '3'])
         print(vector.dtype)
         print(vector)

         # change the data type
         vector = vector.astype(float)
         print(vector.dtype)
         print(vector)
```

```
<U1
['1' '2' '3']
float64
[1. 2. 3.]
```

```
In [13]: # minimum
         vector = np.array([5, 10, 15, 20])
         print(vector.min())

         # maximum
         print(vector.max())
```

```
5
20
```

```
In [14]: # sum
         matrix = np.array([[5, 10, 15],
                            [20, 25, 30],
                            [35, 40, 45]])

         # sum of every row, dimension=1
         print(matrix.sum(axis=1))

         # sum of each column,dimension=0
         print(matrix.sum(axis=0))
```

```
[ 30  75 120]
[60 75 90]
```

```
In [15]: # reshape
         print(np.arange(15))

         matrix = np.arange(15).reshape(3, 5)
         matrix2 = np.arange(15).reshape(5, 3)

         print(matrix)
         print()
         print(matrix2)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

[[ 0  1  2]
 [ 3  4  5]
```

```
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

```
In [16]: # get basic information of a matrix
         # shape
         print(matrix.shape)

         # dimension
         print(matrix.ndim)

         # data type
         print(matrix.dtype)

         # matrix size
         print(matrix.size)
```

```
(3, 5)
2
int64
15
```

```
In [17]: # ZERO
         zero = np.zeros((3, 4))  #float, the size arg should be a tuple
         print(zero)

         # ONE
         one = np.ones((2, 3))  #float
         print(one)

         one = np.ones((2, 3), dtype=np.int64)  #int64
         print(one)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]]
[[1 1 1]
 [1 1 1]]
```

```
In [18]: # arange
         print(np.arange(10, 30, 5))
         print(np.arange(0, 2, 0.3))
         print(np.arange(12).reshape(4, 3))
```

```
[10 15 20 25]
[0.  0.3 0.6 0.9 1.2 1.5 1.8]
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

```
In [19]: # random
         print(np.random.random((2, 3)))
```

```
[[0.81128485 0.75854628 0.00968269]
 [0.44593597 0.50593454 0.50431435]]
```

```
In [20]: # Linespace
         from numpy import pi

         # linspace(start,end, totalNum)
         np.linspace(0, 2 * pi, 100)
```

```
Out[20]: array([0.        , 0.06346652, 0.12693304, 0.19039955, 0.25386607,
                0.31733259, 0.38079911, 0.44426563, 0.50773215, 0.57119866,
                0.63466518, 0.6981317 , 0.76159822, 0.82506474, 0.88853126,
                0.95199777, 1.01546429, 1.07893081, 1.14239733, 1.20586385,
                1.26933037, 1.33279688, 1.3962634 , 1.45972992, 1.52319644,
                1.58666296, 1.65012947, 1.71359599, 1.77706251, 1.84052903,
                1.90399555, 1.96746207, 2.03092858, 2.0943951 , 2.15786162,
                2.22132814, 2.28479466, 2.34826118, 2.41172769, 2.47519421,
                2.53866073, 2.60212725, 2.66559377, 2.72906028, 2.7925268 ,
                2.85599332, 2.91945984, 2.98292636, 3.04639288, 3.10985939,
                3.17332591, 3.23679243, 3.30025895, 3.36372547, 3.42719199,
                3.4906585 , 3.55412502, 3.61759154, 3.68105806, 3.74452458,
                3.8079911 , 3.87145761, 3.93492413, 3.99839065, 4.06185717,
                4.12532369, 4.1887902 , 4.25225672, 4.31572324, 4.37918976,
                4.44265628, 4.5061228 , 4.56958931, 4.63305583, 4.69652235,
                4.75998887, 4.82345539, 4.88692191, 4.95038842, 5.01385494,
                5.07732146, 5.14078798, 5.2042545 , 5.26772102, 5.33118753,
                5.39465405, 5.45812057, 5.52158709, 5.58505361, 5.64852012,
                5.71198664, 5.77545316, 5.83891968, 5.9023862 , 5.96585272,
                6.02931923, 6.09278575, 6.15625227, 6.21971879, 6.28318531])
```

```
In [21]: # sine value of angle
         np.sin(np.linspace(0, 2 * pi, 100))
```

```
Out[21]: array([ 0.00000000e+00,  6.34239197e-02,  1.26592454e-01,  1.89251244e-01,
                 2.51147987e-01,  3.12033446e-01,  3.71662456e-01,  4.29794912e-01,
                 4.86196736e-01,  5.40640817e-01,  5.92907929e-01,  6.42787610e-01,
                 6.90079011e-01,  7.34591709e-01,  7.76146464e-01,  8.14575952e-01,
```

```
                8.49725430e-01,  8.81453363e-01,  9.09631995e-01,  9.34147860e-01,
                9.54902241e-01,  9.71811568e-01,  9.84807753e-01,  9.93838464e-01,
                9.98867339e-01,  9.99874128e-01,  9.96854776e-01,  9.89821442e-01,
                9.78802446e-01,  9.63842159e-01,  9.45000819e-01,  9.22354294e-01,
                8.95993774e-01,  8.66025404e-01,  8.32569855e-01,  7.95761841e-01,
                7.55749574e-01,  7.12694171e-01,  6.66769001e-01,  6.18158986e-01,
                5.67059864e-01,  5.13677392e-01,  4.58226522e-01,  4.00930535e-01,
                3.42020143e-01,  2.81732557e-01,  2.20310533e-01,  1.58001396e-01,
                9.50560433e-02,  3.17279335e-02, -3.17279335e-02, -9.50560433e-02,
               -1.58001396e-01, -2.20310533e-01, -2.81732557e-01, -3.42020143e-01,
               -4.00930535e-01, -4.58226522e-01, -5.13677392e-01, -5.67059864e-01,
               -6.18158986e-01, -6.66769001e-01, -7.12694171e-01, -7.55749574e-01,
               -7.95761841e-01, -8.32569855e-01, -8.66025404e-01, -8.95993774e-01,
               -9.22354294e-01, -9.45000819e-01, -9.63842159e-01, -9.78802446e-01,
               -9.89821442e-01, -9.96854776e-01, -9.99874128e-01, -9.98867339e-01,
               -9.93838464e-01, -9.84807753e-01, -9.71811568e-01, -9.54902241e-01,
               -9.34147860e-01, -9.09631995e-01, -8.81453363e-01, -8.49725430e-01,
               -8.14575952e-01, -7.76146464e-01, -7.34591709e-01, -6.90079011e-01,
               -6.42787610e-01, -5.92907929e-01, -5.40640817e-01, -4.86196736e-01,
               -4.29794912e-01, -3.71662456e-01, -3.12033446e-01, -2.51147987e-01,
               -1.89251244e-01, -1.26592454e-01, -6.34239197e-02, -2.44929360e-16])
```

In [22]:
```
#Operations
a = np.array([20, 30, 40, 50])
b = np.arange(4)
print(a)
print(b)

print(a + b)
print(a - b)

#square
print(b ** 2)
#exponential
print(np.exp(b))
# square root
print(np.sqrt(b))
```

```
[20 30 40 50]
[0 1 2 3]
[20 31 42 53]
[20 29 38 47]
[0 1 4 9]
[ 1.          2.71828183  7.3890561  20.08553692]
[0.         1.         1.41421356 1.73205081]
```

In [23]:
```
# matrix production
A = np.array([[1, 1],
```

```
                    [0, 1]])
        B = np.array([[2, 0],
                    [3, 4]])
        print('-------A:---------')
        print(A)
        print('-------B:---------')
        print(B)
        print('-------A*B:-------')  # coresponding position mutiple
        print(A * B)
        print('-----A.dot(B)-----')  # matrix mutiple
        print(A.dot(B))
        print('---np.dot(A,B)----')  # matrix mutiple
        print(np.dot(A, B))
```

```
-------A:---------
[[1 1]
 [0 1]]
-------B:---------
[[2 0]
 [3 4]]
-------A*B:-------
[[2 0]
 [0 4]]
-----A.dot(B)-----
[[5 4]
 [3 4]]
---np.dot(A,B)----
[[5 4]
 [3 4]]
```

In [24]:
```
        # floor: keep integer
        a = np.floor(10 * np.random.random((3, 4)))
        print(a)
        print('-------------------------')

        # ravel a matrix to a vector
        print(a.ravel())

        print('--------reshape----------')
        print(a.reshape(2, 6))

        print('--------reshape----------')
        print(a.reshape(2, -1))  #auto cal column

        print('--------shape------------')
        # permanently change the shape of the matrix
        a.shape = (2, 6)
```

```
        print(a)

        print('--------Transfer--------')
        # transfer
        print(a.T)
```

```
[[2. 1. 9. 1.]
 [5. 4. 6. 1.]
 [2. 8. 3. 2.]]
-------------------------
[2. 1. 9. 1. 5. 4. 6. 1. 2. 8. 3. 2.]
--------reshape----------
[[2. 1. 9. 1. 5. 4.]
 [6. 1. 2. 8. 3. 2.]]
--------reshape----------
[[2. 1. 9. 1. 5. 4.]
 [6. 1. 2. 8. 3. 2.]]
--------shape------------
[[2. 1. 9. 1. 5. 4.]
 [6. 1. 2. 8. 3. 2.]]
--------Transfer--------
[[2. 6.]
 [1. 1.]
 [9. 2.]
 [1. 8.]
 [5. 3.]
 [4. 2.]]
```

In [25]:
```
# Combinations & Connection
A = np.floor(10 * np.random.random((2, 2)))
B = np.floor(10 * np.random.random((2, 2)))
print(A)
print(B)

print('-------horizontally stack----------')
# horizontally stack
print(np.hstack((A, B)))

print('---------vertical stack------------')
# vertical stack
print(np.vstack((A, B)))
```

```
[[7. 2.]
 [9. 1.]]
[[4. 9.]
 [1. 8.]]
-------horizontally stack----------
```

```
[[7. 2. 4. 9.]
 [9. 1. 1. 8.]]
---------vertical stack------------
[[7. 2.]
 [9. 1.]
 [4. 9.]
 [1. 8.]]
```

```
In [26]: # Split
         C = np.floor(10 * np.random.random((4, 6)))
         print(C)

         print('-------average horizontally split----------')
         # horizontally split
         hs1 = np.hsplit(C, 3)  #average split into 3 parts
         for s in hs1:
             print(s)
             print()

         print('-------certain horizontally split----------')
         hs2 = np.hsplit(C, (3, 5))  #tuple is the split location
         for s in hs2:
             print(s)
             print()

         print('-------average vertical split----------')
         # vertical split
         vs1 = np.vsplit(C, 2)  #average split into 3 parts
         for v in vs1:
             print(v)
             print()

         print('-------certain vertical split----------')
         vs2 = np.vsplit(C, (1, 3))  #tuple is the split location
         for v in vs2:
             print(v)
             print()
```

```
[[4. 7. 2. 8. 0. 9.]
 [0. 0. 6. 9. 9. 1.]
 [8. 3. 3. 8. 4. 7.]
 [4. 5. 3. 9. 8. 6.]]
-------average horizontally split----------
[[4. 7.]
 [0. 0.]
 [8. 3.]
 [4. 5.]]
```

```
[[2. 8.]
 [6. 9.]
 [3. 8.]
 [3. 9.]]

[[0. 9.]
 [9. 1.]
 [4. 7.]
 [8. 6.]]

-------certain horizontally split----------
[[4. 7. 2.]
 [0. 0. 6.]
 [8. 3. 3.]
 [4. 5. 3.]]

[[8. 0.]
 [9. 9.]
 [8. 4.]
 [9. 8.]]

[[9.]
 [1.]
 [7.]
 [6.]]

-------average vertical split----------
[[4. 7. 2. 8. 0. 9.]
 [0. 0. 6. 9. 9. 1.]]

[[8. 3. 3. 8. 4. 7.]
 [4. 5. 3. 9. 8. 6.]]

-------certain vertical split----------
[[4. 7. 2. 8. 0. 9.]]

[[0. 0. 6. 9. 9. 1.]
 [8. 3. 3. 8. 4. 7.]]

[[4. 5. 3. 9. 8. 6.]]
```

```
In [34]: # Soft COPY
         #Simple assignments make no copy of array objects or of their data.
         a = np.arange(12)
         b = a  #Soft copy: same value same location with differnt name
```

```
print('b is a', b is a)
b.shape = (3, 4)

print('a shape', str(a.shape))
print(id(a))
print(id(b))
```

```
b is a True
a shape (3, 4)
4381908384
4381908384
```

```
In [42]: # Shallow Copy
         #The view method creates a new array object that looks at the same data.
         a = np.arange(12)
         c = a.view()
         print('c is a', c is a)
         print(a)
         print(c)

         print('---reshape----')
         c.shape = (2, 6)
         print(a)
         print(c)

         print('---change value----')
         c[0, 4] = 1234
         print(a)
         print(c)
```

```
c is a False
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[ 0  1  2  3  4  5  6  7  8  9 10 11]
---reshape----
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
---change value----
[   0    1    2    3 1234    5    6    7    8    9   10   11]
[[   0    1    2    3 1234    5]
 [   6    7    8    9   10   11]]
```

```
In [43]: # Hard Copy
         #The copy method makes a complete copy of the array and its data.
         a = np.arange(12).reshape(3, 4)
         d = a.copy()
         print('d is a', d is a)
```

```
        d[0, 0] = 999
        print(a)
        print(d)
```

```
d is a False
[[999   1   2   3]
 [  4   5   6   7]
 [  8   9  10  11]]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [49]:
```
        # print maximum value for each column
        data = np.sin(np.arange(20)).reshape(5, 4)
        print(data)
        index = data.argmax(axis=0)  #index of the maximum value for each column
        print(index)

        data_max = data[index, range(data.shape[1])]  #data.shape[1] is the number of the col
        print(data_max)
```

```
[[ 0.          0.84147098  0.90929743  0.14112001]
 [-0.7568025  -0.95892427 -0.2794155   0.6569866 ]
 [ 0.98935825  0.41211849 -0.54402111 -0.99999021]
 [-0.53657292  0.42016704  0.99060736  0.65028784]
 [-0.28790332 -0.96139749 -0.75098725  0.14987721]]
[2 0 3 1]
[0.98935825 0.84147098 0.99060736 0.6569866 ]
```

In [59]:
```
        # extensiontile the origin matrix as a unit
        a = np.arange(0, 40, 10).reshape(2, -1)
        b = np.tile(a, (2, 2))
        print(a)
        print(b)
        print(a.shape)
        print(b.shape)
```

```
[[ 0 10]
 [20 30]]
[[ 0 10  0 10]
 [20 30 20 30]
 [ 0 10  0 10]
 [20 30 20 30]]
(2, 2)
(4, 4)
```

```
In [64]: # sort
         a = np.array([[4, 3, 5], [1, 2, 1]])
         print(a)
         print('----------------------')
         b = np.sort(a, axis=1)  #ranking for each row
         print(b)
         print('----------------------')
         c = np.sort(a, axis=0)  #ranking for each row
         print(c)
         print('----------------------')
         a = np.array([4, 3, 1, 2])
         d = np.argsort(a)  #the sorted index for orginal data
         print(d)
         print(a[d])

[[4 3 5]
 [1 2 1]]
----------------------
[[3 4 5]
 [1 1 2]]
----------------------
[[1 2 1]
 [4 3 5]]
----------------------
[2 3 1 0]
[1 2 3 4]
```