# NumPy

September 4, 2017

## 1 NUMPY

#### 1.0.1 Array = MATRIX

**Numpy's array class is called** $ndarray$ **.** $ndarray.ndim$ : the number of axes (dimensions) of the array.
$ndarray.shape$ : the dimensions of the array.
$ndarray.size$ : the total number of elements of the array.
$ndarray.dtype$ : an object describing the type of the elements in the array.
$ndarray.itemsize$ : the size in bytes of each element of the array.

```
In [1]: import numpy as np
        from numpy import pi

In [2]: a = np.arange(15).reshape(3,5)
        print(a)

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


In [3]: print(a.ndim, a.shape, a.size, a.dtype, a.itemsize)
        B,C = a.shape
        print(B,C)
        #print the attributes of the array class

2 (3, 5) 15 int64 8
3 5


In [4]: b = np.array([(6.0, 7.3, 8.45)])     #declaring a one dimensional array
        print(b)

[[ 6.    7.3   8.45]]


In [5]: print(b.shape, b.dtype)
```

```
(1, 3) float64
```

```
In [6]: b = np.array([[1.5, 2, 3] , [4, 5, 6]])    #declaring a two-dimensional arr
        print(b, b.shape, b.dtype)
```

```
[[ 1.5  2.   3. ]
 [ 4.   5.   6. ]] (2, 3) float64
```

```
In [7]: c = np.array( [ [1,2] , [3,4] ] , dtype = complex )
        print(c)
```

```
[[ 1.+0.j  2.+0.j]
 [ 3.+0.j  4.+0.j]]
```

```
In [8]: np.zeros([3,4])

Out[8]: array([[ 0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.]])
```

```
In [9]: A = np.ones( [ 2, 3, 4 ] , dtype=np.int16 )    #declaring a 3-D array conta
        print(A, A.ndim , A.shape, A.itemsize, A.size)
```

```
[[[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]

 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]] 3 (2, 3, 4) 2 24
```

**To create sequence of numbers, NumPy provides a function analogus to range that returns arrays instead of lists.**

```
np.arange(...)
```

```
In [10]: np.arange(10)

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

In [11]: np.arange(0, 12, 3)  #numbers from 0 to 12 in steps of 3 excluding 12

Out[11]: array([0, 3, 6, 9])

In [12]: np.arange(0, 2, 0.5)    # it accepts float arguments
```

```
Out[12]: array([ 0. ,  0.5,  1. ,  1.5])

In [13]: np.linspace(0, 2, 9)    # 9 equidistant numbers from 0 to 2
                                 #same as linspace of Octave or Matlab

Out[13]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])

In [14]: x = np.array(np.linspace(0, 2*pi, 100))  #useful to evaluate function at l
         f = np.sin(x)

In [15]: np.arange(6)    #1-D array

Out[15]: array([0, 1, 2, 3, 4, 5])

In [16]: np.arange(12).reshape(4,3)    #2-D array

Out[16]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11]])

In [17]: np.arange(24).reshape(2,3,4)   #3-D array

Out[17]: 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]]])
```

### 1.0.2 BASIC OPERATIONS:

Arithmetic operators on arrays apply elementwise.

```
In [18]: a = np.array([20, 30 , 40, 50 ])
         b = np.arange( 4 )                 # b = [[0 1 2 3]]
         c = a-b
         print(c)

[20 29 38 47]


In [19]: print(b**2 , 10*np.sin(a))

[0 1 4 9] [ 9.12945251 -9.88031624  7.4511316  -2.62374854]


In [20]: print(a<35)
```

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

```
In [21]: A = np.array( [ [1,1], [0,1] ] )
         B = np.array( [ [2,0], [3,4] ] )
```

```
In [22]: A*B     # elementwise product
```

```
Out[22]: array([[2, 0],
                [0, 4]])
```

```
In [23]: A.dot(B)     # matrix product
```

```
Out[23]: array([[5, 4],
                [3, 4]])
```

```
In [24]: np.dot(A,B)    # another matrix product
```

```
Out[24]: array([[5, 4],
                [3, 4]])
```

```
In [25]: G=np.dot(A,B)
         F=np.arange(4).reshape(2,2)   # another matrix product
         np.dot(G, F)
```

```
Out[25]: array([[ 8, 17],
                [ 8, 15]])
```

```
In [26]: a = np.random.random((2,3))
         print(a)
```

```
[[ 0.1978628   0.11198542  0.7058449 ]
 [ 0.93923633  0.94289321  0.31265101]]
```

$a.sum()$ **: calculates the sum of all elements in the matrix.**

**Similar jobs are performed by** $a.min()$ **,** $a.max()$

```
In [27]: print(a.sum() , a.min() , a.max() )
```

```
3.2104736679 0.111985420866 0.942893212256
```

```
In [28]: b = np.arange(12).reshape(3,4)
         print(b)
```

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

```
In [29]: b.sum(axis=0)     #sum of each column (axis 0 is vertical axis)

Out[29]: array([12, 15, 18, 21])

In [30]: b.min(axis=1)     #min of each row (axis 1 is horizontal axis)

Out[30]: array([0, 4, 8])
```

## 1.1 UNIVERSAL FUNCTIONS ( ufunc ) :

A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a "vectorized" wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.

### 1.1.1 Some of the universal funtions:

$exp()$ , $add()$ , $subtract()$ , $multiply()$ , $divide()$ , $power()$ , $log()$ , $log10()$ , $sqrt()$ , $cbrt()$ , $sin()$ , $cos()$ , $tan()$ , $floor()$ , $ceil()$

### 1.1.2 Find more about them at:

https://docs.scipy.org/doc/numpy/reference/ufuncs.html

```
In [31]: B = np.arange(3)
         C = np.array([2, -1, 4])
         print(np.exp(B) , np.sqrt(B), np.add(B,C))

[ 1.          2.71828183  7.3890561 ] [ 0.          1.          1.41421356] [2 0 6]


In [32]: C.sort()
         print(C)

[-1  2  4]


In [33]: def f(x,y):
             return 10*x+y

         b = np.fromfunction(f, (5,4) , dtype=int)
         print(b)

[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
```

### 1.1.3 INDEXING , SLICING, AND ITERATING

```
In [34]: print(b[2,3] , b[1,1] , b[0,0] )    # accessing a certain element
```

```
23 11 0
```

```
In [35]: print(b[1])    # accessing a row
```

```
[10 11 12 13]
```

```
In [36]: print(b[1,0:2])    # accessing first two elements in second row
```

```
[10 11]
```

```
In [37]: print(b[-1])    # accessing last row
```

```
[40 41 42 43]
```

```
In [38]: c = np.array( [ [ [0,1,2], [10,12,13] ], [ [100,101,102], [110,112,113] ]
         c.shape
```

```
Out[38]: (2, 2, 3)
```

```
In [39]: print(c[1,...], "\n\n" , c[...,2])    # same as c[1,:,:] & c[:,:,2]
```

```
[[100 101 102]
 [110 112 113]]

 [[  2  13]
 [102 113]]
```

```
In [40]: for row in b:     # same as printing b[0], b[1], ... , etc
             print(row)
```

```
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

```
In [41]: for element in b.flat:    # printing all elements without any array speci
             print(element)
```

```
0
1
2
3
10
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```

```
In [42]: import numpy as np
         np.ravel(b)
```

```
Out[42]: array([ 0,  1,  2,  3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 40,
                41, 42, 43])
```

```
In [43]: b.reshape(2,10)    # reshaping the array into desired dimension
```

```
Out[43]: array([[ 0,  1,  2,  3, 10, 11, 12, 13, 20, 21],
                [22, 23, 30, 31, 32, 33, 40, 41, 42, 43]])
```

```
In [44]: b.ravel()    # flatten the array
```

```
Out[44]: array([ 0,  1,  2,  3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 40,
                41, 42, 43])
```

```
In [45]: print(b)
```

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
```

```
In [46]: print(b.T)    # transpose of the array
```

```
[[ 0 10 20 30 40]
 [ 1 11 21 31 41]
 [ 2 12 22 32 42]
 [ 3 13 23 33 43]]
```

```
In [47]: b.shape = (2,10)     # another way to reshape the matrix
         print(b)
```

```
[[ 0  1  2  3 10 11 12 13 20 21]
 [22 23 30 31 32 33 40 41 42 43]]
```

### 1.1.4 STACKING TOGETHER DIFFERENT ARRAYS

```
In [48]: a = np.floor(10*np.random.random((3,3)))
         print(a)
```

```
[[ 6.  5.  7.]
 [ 3.  9.  5.]
 [ 0.  1.  8.]]
```

```
In [49]: b = np.floor(10*np.random.random((3,3)))
         print(b)
```

```
[[ 4.  1.  3.]
 [ 6.  7.  5.]
 [ 1.  3.  6.]]
```

```
In [50]: np.vstack((a,b))     #vertical stack of a over b: StackHead=TOP
```

```
Out[50]: array([[ 6.,  5.,  7.],
                [ 3.,  9.,  5.],
                [ 0.,  1.,  8.],
                [ 4.,  1.,  3.],
                [ 6.,  7.,  5.],
                [ 1.,  3.,  6.]])
```

```
In [51]: np.hstack((a,b))     #horizontal stack of a over b : StackHead=LEFT
```

```
Out[51]: array([[ 6.,  5.,  7.,  4.,  1.,  3.],
                [ 3.,  9.,  5.,  6.,  7.,  5.],
                [ 0.,  1.,  8.,  1.,  3.,  6.]])
```

```
In [52]: from numpy import newaxis
         np.column_stack((a,b))     # with 2D arrays looks same as hstack
```

```
Out[52]: array([[ 6.,  5.,  7.,  4.,  1.,  3.],
                [ 3.,  9.,  5.,  6.,  7.,  5.],
                [ 0.,  1.,  8.,  1.,  3.,  6.]])
```

```
In [53]: a = np.array([4.,2.])
         b = np.array([3.,8.])
         np.column_stack((a,b))    # returns a 2D array
```

```
Out[53]: array([[ 4.,  3.],
                [ 2.,  8.]])
```

```
In [54]: np.hstack((a,b))    #now the results look different
```

```
Out[54]: array([ 4.,  2.,  3.,  8.])
```

```
In [55]: a[:,newaxis]    #this allows to have a 2D columns vector
```

```
Out[55]: array([[ 4.],
                [ 2.]])
```

```
In [56]: np.column_stack((a[:,newaxis],b[:,newaxis]))
```

```
Out[56]: array([[ 4.,  3.],
                [ 2.,  8.]])
```

```
In [57]: np.hstack((a[:,newaxis],b[:,newaxis]))    # the result is the same
```

```
Out[57]: array([[ 4.,  3.],
                [ 2.,  8.]])
```

```
In [58]: np.r_[1:4,0,4]
```

```
Out[58]: array([1, 2, 3, 0, 4])
```