Content Delivered on class-21_21-July-2016

- Chapter 12: Socket Programming
  - SimpleHTTPServer
- Chapter 17: Numpy –numerical Package
  - Working with Numpy module

Assignments Given

Assignment 1: Try the demonstrated SimpleHTTPServer example

Assignment 2: How different is a.T from a.transpose(), for a numpy array 'a'?

Assignment 3 : Similarly, work on *arcsin*, *arccos*, *arctan*,*sinh*, *cosh*, *tanh*

Interview Questions

Interview Question 1: What is the difference between array and list?

Interview Question 2: what is the difference between zeros_like() and fill(), in numpy arrays?

Interview Question 3: what is the alogorithms used for random number generation, by numpy?

SimpleHTTPServer

As demonstrated, open commandd prompt (or terminal), go to your desired directory, and type

```
python -m  SimpleHTTPServer  <portNumber>

ex: python -m SimpleHTTPServer 9999
```

Note: This command is to be executed in the terminal, and not within the interpreter.

Then, note the IP address of the machine. For example, let it be 192.168.90.76

Now, go to another machine located in the same network, and in the browser, type

```
IPaddress:portNumber.

ex: 192.168.90.76:9999
```

Assignment 1: Try the demonstrated SimpleHTTPServer example

Numpy

Numpy (Numeric Python) is a popular python module used for mathematical and any other numerical computations. It comes pre-installed with python(x,y) distribution. For other distribution, it should be installed manually.

To install numpy in pip, type pip install numpy

Interview Question 1: What is the difference between array and list?

list1 = [12, 34.56, True, 'string', [23, 45, 34, 7.8]]

array is a homogeneous list, containing only one type of datatypes.

array1 = [1., 4., 5., 8.] array2 = [1, 4, 5, 8] array3 = [True, True, True, True]

Various ways of importing a python module

```
import numpy            # preferred
from numpy import array # not useful, with working with more methods of that mod
ule
from numpy import *     # not recommended
import numpy as np    # alias import ; preferred
```

In [188]:

```
import numpy as np
```

Arrays

In [190]:

```
a = np.array([1,4,5,8], float)  # observe that the input to array is a list
```

In [3]:

```
a
```

Out[3]:

```
array([ 1.,  4.,  5.,  8.])
```

In [4]:

```
type(a)
```

Out[4]:

```
numpy.ndarray
```

In [5]:

```
a[:2]   # slicing array
```

Out[5]:

```
array([ 1.,  4.])
```

In [6]:

```
b = np.array([1, 4, 5, 8])
```

In [7]:

```
b, type(b)
```

Out[7]:

```
(array([1, 4, 5, 8]), numpy.ndarray)
```

In [8]:

```
a = np.array([[1, 2, 4], [3, 5, 6]], float)
```

In [9]:

```
a
```

Out[9]:

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

In [10]:

```
a[0,0]    # Unlike lists, arrays can be accessed using (row,column) position
```

Out[10]:

```
1.0
```

In [13]:

```
a[1,2]
```

Out[13]:

```
6.0
```

In [14]:

```
a[1,1]
```

Out[14]:

```
5.0
```

In [15]:

```
a[1,:]
```

Out[15]:

```
array([ 3.,  5.,  6.])
```

In [16]:

```
a[:, 2]     # : indicates everything in that dimension
```

Out[16]:

```
array([ 4.,  6.])
```

In [17]:

```
a[-1]
```

Out[17]:

```
array([ 3.,  5.,  6.])
```

In [18]:

```
a[-2]
```

Out[18]:

```
array([ 1.,  2.,  4.])
```

In [19]:

```
a[0]
```

Out[19]:

```
array([ 1.,  2.,  4.])
```

In [20]:

```
a[-1:, -2:]
```

Out[20]:

```
array([[ 5.,  6.]])
```

In [21]:

```
a.shape  # retuns a tuple (NoOfRows, NoOfColumns)
```

Out[21]:

```
(2, 3)
```

In [22]:

```
a.dtype
```

Out[22]:

```
dtype('float64')
```

In [23]:

```
a = np.array([[1,2,3], [4,5,6]], float)
```

In [24]:

```
a
```

Out[24]:

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

In [25]:

```
len(a)        # returns length of the first axis
```

Out[25]:

```
2
```

In [26]:

```
len(a[0])
```

Out[26]:

```
3
```

In [27]:

```
2 in a
```

Out[27]:

```
True
```

In [28]:

```
0 in a
```

Out[28]:

```
False
```

In [29]:

```
a = np.array(range(10), float)
```

In [30]:

```
a
```

Out[30]:

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

In [31]:

```
a = a.reshape((5,2))   # 5*2 = 10 = total elements in array
```

In [32]:

```
a                    # reshape function creates a new array and doesn't itself modify
 the original array
```

Out[32]:

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

In [34]:

```
a.shape
```

Out[34]:

```
(5, 2)
```

In [35]:

```
b = a
```

In [36]:

```
c = a.copy()              # copy() of array does the same job as copy.deepcopy() of list
s
```

In [37]:

```
a[0] = 999
```

In [38]:

```
a
```

Out[38]:

```
array([[ 999.,  999.],
       [   2.,    3.],
       [   4.,    5.],
       [   6.,    7.],
       [   8.,    9.]])
```

In [39]:

```
b
```

Out[39]:

```
array([[ 999.,  999.],
       [   2.,    3.],
       [   4.,    5.],
       [   6.,    7.],
       [   8.,    9.]])
```

In [40]:

```
c
```

Out[40]:

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

In [41]:

```
a
```

Out[41]:

```
array([[ 999.,  999.],
       [   2.,    3.],
       [   4.,    5.],
       [   6.,    7.],
       [   8.,    9.]])
```

In [42]:

```
a.tolist()   # list of lists
```

Out[42]:

```
[[999.0, 999.0], [2.0, 3.0], [4.0, 5.0], [6.0, 7.0], [8.0, 9.0]]
```

In [43]:

```
type(a)
```

Out[43]:

```
numpy.ndarray
```

In [44]:

```
type(a.tolist())
```

Out[44]:

```
list
```

In [45]:

```
list(a)    # list of arrays
```

Out[45]:

```
[array([ 999.,  999.]),
 array([ 2.,  3.]),
 array([ 4.,  5.]),
 array([ 6.,  7.]),
 array([ 8.,  9.])]
```

In [47]:

```
d = np.array([1,2,3], float)
```

In [48]:

```
d1 = d.tostring()     # converts the raw data to a binary string (Non-human readable)
```

In [49]:

```
d1, type(d1)
```

Out[49]:

```
('\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x
00\x00\x00\x08@',
 str)
```

In [50]:

```
np.fromstring(d1)
```

Out[50]:

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

In [51]:

```
d
```

Out[51]:

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

In [52]:

```
d.fill(8)        # fill() - to fill the complete array with a single element
```

In [53]:

```
d
```

Out[53]:

```
array([ 8.,  8.,  8.])
```

In [54]:

```
a = np.array(range(6), float)
```

In [55]:

```
a
```

Out[55]:

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

In [56]:

```
a = np.array(range(6), float).reshape((2,3))
```

In [57]:

```
a
```

Out[57]:

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

In [58]:

```
a.transpose()   # won't change the original object
```

Out[58]:

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

Assignment 2: How different is a.T from a.transpose(), for a numpy array 'a'?

In [59]:

```
a
```

Out[59]:

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

In [60]:

```
a.flatten()
```

Out[60]:

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

In [61]:

```
a1 = a.transpose()
a1.flatten()
```

Out[61]:

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

In [63]:

```
ac = np.concatenate((a.flatten(), b.flatten()))
```

In [64]:

```
ac
```

Out[64]:

```
array([   0.,    1.,    2.,    3.,    4.,    5.,  999.,  999.,    2.,
          3.,    4.,    5.,    6.,    7.,    8.,    9.])
```

In [65]:

```
ac.shape
```

Out[65]:

```
(16,)
```

In [71]:

```
a = np.array([[1,2],[3,4]], float)
```

In [72]:

```
b = np.array([[5,6], [7,8]])
```

In [73]:

```
np.concatenate((a,b))   # Default is axis = 0
```

Out[73]:

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

In [74]:

```
np.concatenate((a,b), axis =0)  # Both should be of same dimension, in the axis od conc
atenation
```

Out[74]:

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

In [75]:

```
np.concatenate((a,b), axis = 1)
```

Out[75]:

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

In [76]:

```
a = np.array([1,2,3], float)
```

In [77]:

```
a
```

Out[77]:

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

In [78]:

```
a[:,np.newaxis]     # newaxis - used to increase the dimensionality of an array
```

Out[78]:

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

In [79]:

```
a[:,np.newaxis].shape
```

Out[79]:

```
(3, 1)
```

In [80]:

```
a[np.newaxis, :]
```

Out[80]:

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

In [81]:

```
b[np.newaxis, :]
```

Out[81]:

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

Other ways to create arrays

In [82]:

```
np.arange(5, dtype= float)   # works same as range()
```

Out[82]:

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

In [83]:

```
np.arange(1, 6, 2, dtype= float)
```

Out[83]:

```
array([ 1.,  3.,  5.])
```

In [84]:

```
np.arange(1, 100, 10, dtype= int)
```

Out[84]:

```
array([ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91])
```

In [85]:

```
np.arange(0, 100, 10, dtype= int)
```

Out[85]:

```
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

In [86]:

```
a = np.array([[1,2,3],[4,5,6]], float)
```

In [191]:

```
np.ones((3,2), dtype = int)
```

Out[191]:

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

In [192]:

```
np.zeros((3,2), dtype = float)
```

Out[192]:

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

In [87]:

```
np.zeros_like(a)
```

Out[87]:

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

In [88]:

```
a2 = np.zeros_like(a)
```

In [89]:

```
type(a2)
```

Out[89]:

```
numpy.ndarray
```

In [90]:

```
np.ones_like(a)
```

Out[90]:

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

Interview Question 2: what is the difference between zeros_like() and fill(), in numpy arrays?

In [91]:

```
np.identity(4, dtype= float)  # Identity matric had 1 in principal diagonal, and zeros
 in other positions
```

Out[91]:

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

In [92]:

```
np.eye(4, k=1, dtype= float)
```

Out[92]:

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

In [93]:

```
np.eye(4, k=0, dtype= float)   # equal to np.identity(4, dtype= float)
```

Out[93]:

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

Array mathematics

In [95]:

```
a = np.array([1,2,3], float)
```

In [96]:

```
b = np.array([5,2,6], float)
```

In [97]:

```
a+b
```

Out[97]:

```
array([ 6.,  4.,  9.])
```

In [98]:

```
a-b
```

Out[98]:

```
array([-4.,  0., -3.])
```

In [99]:

```
a*b
```

Out[99]:

```
array([  5.,   4.,  18.])
```

In [100]:

```
a/b
```

Out[100]:

```
array([ 0.2,  1. ,  0.5])
```

In [101]:

```
b/a
```

Out[101]:

```
array([ 5.,  1.,  2.])
```

In [102]:

```
a%b
```

Out[102]:

```
array([ 1.,  0.,  3.])
```

In [103]:

```
a**b
```

Out[103]:

```
array([   1.,    4.,  729.])
```

In [104]:

```
b**a
```

Out[104]:

```
array([   5.,    4.,  216.])
```

In [105]:

```
b**2
```

Out[105]:

```
array([ 25.,   4.,  36.])
```

In [108]:

```
a
```

Out[108]:

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

In [109]:

```
b
```

Out[109]:

```
array([ 5.,  2.,  6.])
```

In [111]:

```
b.reshape((3,1))
```

Out[111]:

```
array([[ 5.],
       [ 2.],
       [ 6.]])
```

In [114]:

```
a+b.reshape((3,1))
```

Out[114]:

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

In [115]:

```
a
```

Out[115]:

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

In [116]:

```
a = np.zeros((2,2), float)
```

In [117]:

```
a
```

Out[117]:

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

In [118]:

```
b = np.array([-1., 3.], float)
```

In [119]:

```
b
```

Out[119]:

```
array([-1.,  3.])
```

In [120]:

```
np.sqrt(b)      # because b[0] = -1. observe that unlike lists, it didn't return an exce
ption
```

```
c:\python27\lib\site-packages\ipykernel\__main__.py:1: RuntimeWarning: inva
lid value encountered in sqrt
  if __name__ == '__main__':
```

Out[120]:

```
array([        nan,  1.73205081])
```

In [121]:

```
np.sqrt(a)
```

Out[121]:

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

In [122]:

```
a = np.array([1.1, 1.5, 1.6, 1.9], float)
```

In [123]:

```
a
```

Out[123]:

```
array([ 1.1,  1.5,  1.6,  1.9])
```

In [124]:

```
np.floor(a)
```

Out[124]:

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

In [125]:

```
np.ceil(a)
```

Out[125]:

```
array([ 2.,  2.,  2.,  2.])
```

In [126]:

```
np.rint(a)     # Results in rounded integer
```

Out[126]:

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

In [127]:

```
a = np.array([1.1, 1.4, 1.5, 1.6, 1.9], float)
```

In [128]:

```
np.rint(a)     # Try np.round() and np.round_()
```

Out[128]:

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

In [129]:

```
np.pi
```

Out[129]:

```
3.141592653589793
```

In [130]:

```
np.e
```

Out[130]:

```
2.718281828459045
```

Array Iterations

In [132]:

```
a = np.array([1,2,3,4], int)
```

In [133]:

```
for i in a:
    print i
```

```
1
2
3
4
```

In [134]:

```
i, type(i)
```

Out[134]:

```
(4, numpy.int32)
```

In [135]:

```
a = np.array([[1,2], [3,4], [5,6]], float)
```

In [136]:

```
a
```

Out[136]:

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

In [137]:

```
for (i,j) in a:
    print i*j
```

```
2.0
12.0
30.0
```

In [138]:

```
a.sum()
```

Out[138]:

```
21.0
```

In [139]:

```
a.prod()
```

Out[139]:

```
720.0
```

In [140]:

```
a.mean()
```

Out[140]:

```
3.5
```

In [141]:

```
a.var()  #variance
```

Out[141]:

```
2.9166666666666665
```

In [142]:

```
a.std() # standard deviation
```

Out[142]:

```
1.707825127659933
```

In [143]:

```
a.min()
```

Out[143]:

```
1.0
```

In [144]:

```
a.max()
```

Out[144]:

```
6.0
```

In [146]:

```
a
```

Out[146]:

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

In [145]:

```
a.argmin(), a.argmax()  # retuns the min and max positions, in array, correspondingly
```

Out[145]:

```
(0, 5)
```

In [147]:

```
a = np.array([[0,2],[3,-1], [3,5]], float)
```

In [148]:

```
a
```

Out[148]:

```
array([[ 0.,  2.],
       [ 3., -1.],
       [ 3.,  5.]])
```

In [149]:

```
a.mean(axis =0)
```

Out[149]:

```
array([ 2.,  2.])
```

In [150]:

```
a.mean(axis = 1)
```

Out[150]:

```
array([ 1.,  1.,  4.])
```

In [151]:

```
a.min(axis=0)
```

Out[151]:

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

In [152]:

```
a.max(axis =1)
```

Out[152]:

```
array([ 2.,  3.,  5.])
```

In [153]:

```
a = np.array([6, 5, 3, 6, 2, 7, 2, -1, -34], float)
```

In [154]:

```
a
```

Out[154]:

```
array([  6.,   5.,   3.,   6.,   2.,   7.,   2.,  -1., -34.])
```

In [156]:

```
sorted(a)
```

Out[156]:

```
[-34.0, -1.0, 2.0, 2.0, 3.0, 5.0, 6.0, 6.0, 7.0]
```

In [157]:

```
a.sort()
```

In [158]:

```
a
```

Out[158]:

```
array([-34.,  -1.,   2.,   2.,   3.,   5.,   6.,   6.,   7.])
```

In [159]:

```
a = np.array([6, 2, 5, -2, 0], float)
```

In [160]:

```
a.clip(0,5)
```

Out[160]:

```
array([ 5.,  2.,  5.,  0.,  0.])
```

In [161]:

```
a = np.array([11, 4, 55, 55, 4, 0, 11])
```

In [162]:

```
a
```

Out[162]:

```
array([11,  4, 55, 55,  4,  0, 11])
```

In [163]:

```
np.unique(a)       # duplicates are removed
```

Out[163]:

```
array([ 0,  4, 11, 55])
```

In [164]:

```
a = np.array(range(2,3), float)
```

In [165]:

```
a
```

Out[165]:

```
array([ 2.])
```

In [166]:

```
a= np.array([[2, 34], [23, 345]], float)
```

In [168]:

```
a
```

Out[168]:

```
array([[   2.,   34.],
       [  23.,  345.]])
```

In [167]:

```
a.diagonal()
```

Out[167]:

```
array([   2.,  345.])
```

Comparision operators

In [170]:

```
a = np.array([1, 3, 0], float)
```

In [171]:

```
b = np.array([0,3,2], float)
```

In [172]:

```
a>b
```

Out[172]:

```
array([ True, False, False], dtype=bool)
```

In [173]:

```
a == b
```

Out[173]:

```
array([False,  True, False], dtype=bool)
```

In [174]:

```
a<=b
```

Out[174]:

```
array([False,  True,  True], dtype=bool)
```

In [175]:

```
c = a<=b
```

In [176]:

```
c
```

Out[176]:

```
array([False,  True,  True], dtype=bool)
```

In [177]:

```
print c
```

```
[False  True  True]
```

In [178]:

```
a = np.array([1,3,0], float)
```

In [179]:

```
a>2
```

Out[179]:

```
array([False,  True, False], dtype=bool)
```

In [180]:

```
any(a)
```

Out[180]:

```
True
```

In [181]:

```
all(a)
```

Out[181]:

```
False
```

In [182]:

```
a
```

Out[182]:

```
array([ 1.,  3.,  0.])
```

In [183]:

```
a>0
```

Out[183]:

```
array([ True,  True, False], dtype=bool)
```

In [184]:

```
a<3
```

Out[184]:

```
array([ True, False,  True], dtype=bool)
```

In [185]:

```
np.logical_and(a>0, a<3)
```

Out[185]:

```
array([ True, False, False], dtype=bool)
```

In [186]:

```
np.logical_or(a>0, a<3)
```

Out[186]:

```
array([ True,  True,  True], dtype=bool)
```

In [187]:

```
np.logical_not(a<3)
```

Out[187]:

```
array([False,  True, False], dtype=bool)
```

where function forms a new array from two arrays of equaivalent size using a boolean filter to choose between elements of the two .

syntax: np.where(boolarray, truearray, falsearray)

In [1]:

```
import numpy as np
```

In [2]:

```
a = np.array([1, 3, 0], float)
```

In [3]:

```
a
```

Out[3]:

```
array([ 1.,  3.,  0.])
```

In [4]:

```
1/a                                                    # Observe that it resulted in
 warning
```

```
c:\python27\lib\site-packages\ipykernel\__main__.py:1: RuntimeWarning: divi
de by zero encountered in divide
  if __name__ == '__main__':
```

Out[4]:

```
array([ 1.        ,  0.33333333,         inf])
```

In [5]:

```
np.where(a!=0, 1/a, a)
```

```
c:\python27\lib\site-packages\ipykernel\__main__.py:1: RuntimeWarning: divi
de by zero encountered in divide
  if __name__ == '__main__':
```

Out[5]:

```
array([ 1.        ,  0.33333333,  0.        ])
```

In [6]:

```
1/0                                                    # Doing the same on integers,
 with result in ZeroDivisionError.
```

```
---------------------------------------------------------------------------
-
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-6-05c9758a9c21> in <module>()
----> 1 1/0

ZeroDivisionError: integer division or modulo by zero
```

In [7]:

```
np.where(a>0, 3, 2)    # using broadcasting
```

Out[7]:

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

In [40]:

```
a = np.array([[0,1], [-3,0]],float) # returns tuple of indices of nonzero values in an a
rray
a.nonzero()
```

Out[40]:

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

In [41]:

```
a = np.array([1, np.NaN, np.Inf], float) # NaN - Not a Number Inf - Infinite
```

In [42]:

```
a
```

Out[42]:

```
array([  1.,  nan,  inf])
```

In [43]:

```
np.isnan(a)
```

Out[43]:

```
array([False,  True, False], dtype=bool)
```

In [44]:

```
np.isfinite(a)
```

Out[44]:

```
array([ True, False, False], dtype=bool)
```

array Broaddcasting in numpy

During arithmetic operations, the smaller array is *broadcasted* across the larger array so that they have compatible shapes.

In [9]:

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([3.0, 3.0, 3.0])
a*b            # normal case
```

Out[9]:

```
array([ 3.,  6.,  9.])
```

In [11]:

```
a = np.array([1.0,2.0,3.0])
b = 3.0      # Here, array 'b' broadcasts over array 'a'.
a*b          # equivalent to array([1.0,2.0,3.0])*array([3.0,3.0,3.0])
```

Out[11]:

```
array([ 3.,  6.,  9.])
```

In [12]:

```
b           # 'b' remains the same, after this arithmetic operation
```

Out[12]:

```
3.0
```

In [13]:

```
a = np.array([[1,2], [3,4], [5,6]], float)
```

In [14]:

```
b = np.array([-1,3], float)
```

In [15]:

```
a
```

Out[15]:

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

In [16]:

```
b
```

Out[16]:

```
array([-1.,  3.])
```

In [17]:

```
a+b
```

Out[17]:

```
array([[ 0.,  5.],
       [ 2.,  7.],
       [ 4.,  9.]])
```

Here, 'b' is broadcasted over 'a'. The broadcasted array will be like

```
    array([[-1.,  3.],
           [-1.,  3.],
           [-1.,  3.]])
```

In [18]:

```
a = np.zeros((2,2), float)
b = np.array([-1, -3], float)
```

In [19]:

```
a
```

Out[19]:

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

In [20]:

```
b
```

Out[20]:

```
array([-1., -3.])
```

In [21]:

```
a+b
```

Out[21]:

```
array([[-1., -3.],
       [-1., -3.]])
```

In [22]:

```
b[np.newaxis,:]
```

Out[22]:

```
array([[-1., -3.]])
```

In [23]:

```
a+b[np.newaxis,:]   # broadcasting 'b' as a row
```

Out[23]:

```
array([[-1., -3.],
       [-1., -3.]])
```

In [24]:

```
a+b[:,np.newaxis]   # broadcasting 'b' as a column
```

Out[24]:

```
array([[-1., -1.],
       [-3., -3.]])
```

But, in some cases, broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

Trignometic operations

In [25]:

```
np.pi
```

Out[25]:

```
3.141592653589793
```

In [26]:

```
np.e
```

Out[26]:

```
2.718281828459045
```

In [28]:

```
a = np.array([-25, 0, 36])
```

In [29]:

```
np.sign(a)
```

Out[29]:

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

In [30]:

```
np.log(a)   # observe two warnings, here
```

```
c:\python27\lib\site-packages\ipykernel\__main__.py:1: RuntimeWarning: divi
de by zero encountered in log
  if __name__ == '__main__':
c:\python27\lib\site-packages\ipykernel\__main__.py:1: RuntimeWarning: inva
lid value encountered in log
  if __name__ == '__main__':
```

Out[30]:

```
array([        nan,        -inf,  3.58351894])
```

In [31]:

```
np.log10(a)
```

```
c:\python27\lib\site-packages\ipykernel\__main__.py:1: RuntimeWarning: divi
de by zero encountered in log10
  if __name__ == '__main__':
c:\python27\lib\site-packages\ipykernel\__main__.py:1: RuntimeWarning: inva
lid value encountered in log10
  if __name__ == '__main__':
```

Out[31]:

```
array([       nan,       -inf,  1.5563025])
```

In [32]:

```
np.sin(a)
```

Out[32]:

```
array([ 0.13235175,  0.        , -0.99177885])
```

In [33]:

```
np.cos(a)
```

Out[33]:

```
array([ 0.99120281,  1.        , -0.12796369])
```

In [34]:

```
np.tan(a)
```

Out[34]:

```
array([ 0.13352641,  0.        ,  7.75047091])
```

Assignment 3 : Similarly, work on *arcsin*, *arccos*, *arctan*,*sinh*, *cosh*, *tanh*

Array item selection and manipulation

In [45]:

```
a = np.array([[6, 4], [5, 9]], float)
```

In [46]:

```
a>=5
```

Out[46]:

```
array([[ True, False],
       [ True,  True]], dtype=bool)
```

In [49]:

```
b = a[a>=5]    # creates a new object
b
```

Out[49]:

```
array([ 6.,  5.,  9.])
```

In [50]:

```
a              # 'a' remains unchanged
```

Out[50]:

```
array([[ 6.,  4.],
       [ 5.,  9.]])
```

In [51]:

```
cond = (a>=5)
a[cond]
```

Out[51]:

```
array([ 6.,  5.,  9.])
```

In [52]:

```
a[np.logical_and(a>5, a<9)]
```

Out[52]:

```
array([ 6.])
```

In [53]:

```
a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int)  # 'b' must be of int type only
a[b]   # indexing elements of 'a' with elements of 'b'
```

Out[53]:

```
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

In [54]:

```
a[[0,0,1,3,2,1]]
```

Out[54]:

```
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

Similarly, for multidimensional arrays, multiple one-dimensional integer arrays must be sent to the selection bracket, one for each axis.

In [55]:

```
a = np.array([[1, 4], [9, 16]], float)
b = np.array([0, 0, 1, 1, 0], int)
c = np.array([0, 1, 1, 1, 1], int)
a[b,c]
```

Out[55]:

```
array([  1.,   4.,  16.,  16.,   4.])
```

np.take also performs slection with integer arrays

In [56]:

```
a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int)
a.take(b)
```

Out[56]:

```
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

In [57]:

```
a = np.array([[0, 1], [2, 3]], float)
b = np.array([0, 0, 1], int)
a.take(b, axis=0)
```

Out[57]:

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

In [58]:

```
a.take(b, axis=1)
```

Out[58]:

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

np.put function will take values from a source array and place them at a specified indices in the array calling *put*

In [59]:

```
a = np.array([0, 1, 2, 3, 4, 5], float)
b = np.array([9, 8, 7], float)
```

In [60]:

```
a
```

Out[60]:

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

In [61]:

```
a.put([0, 3], b)
```

In [62]:

```
a
```

Out[62]:

```
array([ 9.,  1.,  2.,  8.,  4.,  5.])
```

In [63]:

```
a.put([1, 4], [99, 89])
a
```

Out[63]:

```
array([  9.,  99.,   2.,   8.,  89.,   5.])
```

In [64]:

```
a.put([0,3], 5)
```

In [65]:

```
a
```

Out[65]:

```
array([  5.,  99.,   2.,   5.,  89.,   5.])
```

Vector and matrix mathematics

In [67]:

```
a = np.array([1, 2, 3], float)
b = np.array([0, 1, 1], float)
np.dot(a,b)     # dot multiplication or matrix mulitplication
```

Out[67]:

```
5.0
```

In [68]:

```
a*b    # ordinary array multiplication
```

Out[68]:

```
array([ 0.,  2.,  3.])
```

In [69]:

```
a = np.array([[0, 1], [2, 3]], float)
b = np.array([2, 3], float)
c = np.array([[1, 1], [4, 0]], float)
```

In [70]:

```
a
```

Out[70]:

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

In [71]:

```
b
```

Out[71]:

```
array([ 2.,  3.])
```

In [72]:

```
np.dot(a,b)
```

Out[72]:

```
array([  3.,  13.])
```

In [73]:

```
np.dot(b,a)
```

Out[73]:

```
array([  6.,  11.])
```

In [74]:

```
np.dot(a,c)
```

Out[74]:

```
array([[  4.,   0.],
       [ 14.,   2.]])
```

In [75]:

```
np.dot(c,a)
```

Out[75]:

```
array([[ 2.,  4.],
       [ 0.,  4.]])
```

Getting the inner, outer and cross products of matrices and vectors

In [76]:

```
a = np.array([1, 4, 0], float)
b = np.array([2, 2, 1], float)
```

In [77]:

```
np.outer(a, b)
```

Out[77]:

```
array([[ 2.,  2.,  1.],
       [ 8.,  8.,  4.],
       [ 0.,  0.,  0.]])
```

In [78]:

```
np.inner(a, b)    # equivalent to np.dot(a,b)
```

Out[78]:

```
10.0
```

In [79]:

```
np.cross(a,b)
```

Out[79]:

```
array([ 4., -1., -6.])
```

np.linalg- numpy submodule for linear algebra calculations

In [80]:

```
a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
```

In [81]:

```
a
```

Out[81]:

```
array([[ 4.,  2.,  0.],
       [ 9.,  3.,  7.],
       [ 1.,  2.,  1.]])
```

In [82]:

```
np.linalg.det(a)  # determinant of matrix 'a'
```

Out[82]:

```
-48.000000000000028
```

In [83]:

```
vals, vecs = np.linalg.eig(a)  # eigen values, eigen vectors
```

In [84]:

```
vals
```

Out[84]:

```
array([ 8.85591316,  1.9391628 , -2.79507597])
```

In [85]:

```
vecs
```

Out[85]:

```
array([[-0.3663565 , -0.54736745,  0.25928158],
       [-0.88949768,  0.5640176 , -0.88091903],
       [-0.27308752,  0.61828231,  0.39592263]])
```

In [86]:

```
b = np.linalg.inv(a)  # Inverse of matrix 'a'
b
```

Out[86]:

```
array([[ 0.22916667,  0.04166667, -0.29166667],
       [ 0.04166667, -0.08333333,  0.58333333],
       [-0.3125    ,  0.125     ,  0.125     ]])
```

In [87]:

```
np.dot(a,b)
```

Out[87]:

```
array([[  1.00000000e+00,   0.00000000e+00,  -2.22044605e-16],
       [  0.00000000e+00,   1.00000000e+00,   0.00000000e+00],
       [  0.00000000e+00,   0.00000000e+00,   1.00000000e+00]])
```

In [88]:

```
np.dot(b,a)
```

Out[88]:

```
array([[  1.00000000e+00,  -1.11022302e-16,  -5.55111512e-17],
       [  0.00000000e+00,   1.00000000e+00,   0.00000000e+00],
       [  2.77555756e-17,   5.55111512e-17,   1.00000000e+00]])
```

In [89]:

```
a = np.array([[1, 3, 4], [5, 2, 3]], float)
U, s, Vh = np.linalg.svd(a)   # single value decomposition (analogous to diagonalizatio
n of non-square matrix)
```

In [90]:

```
U
```

Out[90]:

```
array([[-0.6113829 , -0.79133492],
       [-0.79133492,  0.6113829 ]])
```

In [91]:

```
s
```

Out[91]:

```
array([ 7.46791327,  2.86884495])
```

In [92]:

```
Vh
```

Out[92]:

```
array([[-0.61169129, -0.45753324, -0.64536587],
       [ 0.78971838, -0.40129005, -0.46401635],
       [-0.046676  , -0.79349205,  0.60678804]])
```

Statistics

Previously, we worked with *mean*, *var* and *std* functions

In [93]:

```
a = np.array([1, 4, 3, 8, 9, 2, 3], float)
np.median(a)
```

Out[93]:

```
3.0
```

In [94]:

```
np.corrcoef(a) # correlation coefficients
```

Out[94]:

```
1.0
```

In [95]:

```
a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
c = np.corrcoef(a)
```

In [96]:

```
c
```

Out[96]:

```
array([[ 1.        ,  0.72870505],
       [ 0.72870505,  1.        ]])
```

In [97]:

```
np.cov(a)   # covariance of matrix 'a'
```

Out[97]:

```
array([[ 0.91666667,  2.08333333],
       [ 2.08333333,  8.91666667]])
```

Random Numbers

Interview Question 3: what is the alogorithms used for random number generation, by numpy?

Numpy uses *Mersenne Twister* algorithm to generate pseudorandom numbers

In [98]:

```
np.random.seed(34534)  # setting the random number seed
```

In [99]:

```
np.random.rand(5)
```

Out[99]:

```
array([ 0.95729641,  0.74501966,  0.44852738,  0.66460777,  0.03405516])
```

In [100]:

```
np.random.rand(2,3)   # specifying the dimensions
```

Out[100]:

```
array([[ 0.23094279,  0.45053862,  0.64043204],
       [ 0.30279129,  0.0718453 ,  0.45795402]])
```

In [101]:

```
np.random.rand(6).reshape((2,3))
```

Out[101]:

```
array([[ 0.12194709,  0.84023691,  0.07016156],
       [ 0.61370052,  0.58303531,  0.44212102]])
```

In [102]:

```
np.random.random()
```

Out[102]:

```
0.4900647871780882
```

In [103]:

```
np.random.randint(5, 10)
```

Out[103]:

```
6
```

In [104]:

```
np.random.poisson(6.0)  # discrete Poisson distribution with lambda = 6.0
```

Out[104]:

```
6
```

In [105]:

```
np.random.normal(1.5, 4.0) # continuous normal (Gaussian) distribution with mean= 1.5 a
nd standard
deviation = 4.0
```

Out[105]:

```
-4.547567295390245
```

In [106]:

```
np.random.normal()  # standard normal distribution (mean=0, std. deviation = 1
```

Out[106]:

```
-0.5778105691198495
```

In [107]:

```
np.random.normal(size=5)
```

Out[107]:

```
array([-0.49419307,  0.53339334, -0.2970153 , -2.1170647 , -0.51534904])
```

In [108]:

```
l = range(10)
```

In [109]:

```
l
```

Out[109]:

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

In [110]:

```
np.random.shuffle(l)
```

In [111]:

```
l
```

Out[111]:

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