Numpy

```
import numpy as np
```

Array Create

- `np.array([1,2,3])`
- `np.arange(10)` - create numbers from 0 to 9
- `np.arange(1,11)` - create numbers from 1 to 10
- `np.arange(0,11,2)` - create numbers from 0 to 10 with stepsize as 2. so **[0, 2, 4, 6, 8, 10]**
- `np.arange(10, dtype='float64')` - create floating numbers
- `np.linspace(0,1,11)` - create linearly spaced array
- `np.ones(4)` or `np.ones((2,3))` - Matrix with ones. Send the matrix size as integer for 1 dimension and as tuple for n-dimension
- `np.zeros((2,3))` - Matrix with zeros. Send the matrix size as tuple
- `np.eye(3)` or `np.eye(3,2)` - Identity matrix 3x3 or matrix with ones only in the first 2 rows
- `a = np.diag([1,2,3,4])` - create diagonal matrix
- `np.diag(a)` - extract the diagonal matrix elts in a list
- `np.random.rand(4)` - random numbers of uniform distribution
- `np.random.randn(4)` - random number of standard deviation
- `np.random.randint(0, 20, 15)` - random integers [start index, end index, total number of elements]

Array's attributes

- `.ndim` - dimension
- `.shape` - shape as matrix
- `.dtype` - gives the data type
- `.T` - will give the transpose

Array's functions

- `len(nparray)` - returns the no of element in the first dimension
- `np.sum(a)` - returns the sum of all elements in an array
- `np.sum(a, axis=0/1)` - returns the array of sum of elements in each axis
- `np.shares_memory(a,b)` - Boolean : True if both **a** and **b** shares same RAM memory
- `np.sin(a)` - Elementwise apply **sin** function
- `np.log(a)` - Elementwise apply **log** function
- `np.exp(a)` - Elementwise apply **exp** function
- `np.min(a)` , `np.max(a)` - Find min & max of the array
- `np.argmin(a)` , `np.argmax(a)` - Find the argument of min & max element of the array
- `np.all(a)` - Boolean : true if all the values are true
- `np.any(a)` - Boolean : False if all the values are False
- `np.mean(x)` or `np.mean(x,axis=0/1)` - Find mean for 1D or 2D array

- `np.median(x)` or `np.median(x, axis=0/1)` - Find median for 1D or 2D array
- `np.std(x)` or `np.std(x, axis=0/1)` - Find standard deviation for 1D orr 2D array
- `array.ravel()` - Flatten the matrix by iterating each dimensions
- `array.reshape((1,2))` - Send a **tuple** as shape to reshape. The total elements in the matrix should match the product of nos. in the tuple. ***It may create a copy or create a view***. So be careful
- `array.resize((1,2))` - Send a **tuple** as shape to resize. The total elements in the matrix may not match the product of nos. in the tuple. So if needed, 0's will be added.
- `np.sort(x)` or `np.sort(x, axis=0/1)` - sort the numpy array **x**
- `np.argsort(x)` or `np.argsort(x, axis=0/1)` - return the **sorted indices**

```
a = np.array([0,1,2,3])
b = np.arange(10)
print(a)
print(b)
```

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

```
L = range(1000)
%timeit [i**2 for i in L]
```

```
1000 loops, best of 5: 262 µs per loop
```

```
L = np.arange(1000)
%timeit L**2
```

```
The slowest run took 24.59 times longer than the fastest. This could mean that an interme
1000000 loops, best of 5: 1.42 µs per loop
```

Creating Arrays

```
a = np.array([0, 1, 2, 3])
print(a)
```

```
[0 1 2 3]
```

```
a.ndim
```

```
1
```

```
a.shape
```

```
(4,)
```

```
len(a)
```

```
    4
```

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

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

```
b.ndim
```

```
    2
```

```
b.shape
```

```
    (2, 3)
```

```
print(b.dtype)
```

```
    int64
```

```
len(b) #returns size of first dimension
```

```
    2
```

```
c = np.array([[[0,1],[2,3]],[[4,5],[6,7]]])
c
```

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

           [[4, 5],
            [6, 7]]])
```

```
c.ndim
```

```
    3
```

```
c.shape
```

```
    (2, 2, 2)
```

```
a = np.arange(1,11,2) #start, end(exclusive), stepsize
a
```

```
    array([1, 3, 5, 7, 9])
```

```
a = np.linspace(0,1,11) #start, end, no of points
a
```

```
    array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

```
print(np.ones(4))
print(np.ones((4,)))
print('2D - ', np.ones((3,3)))
print('3D - ', np.ones((2,2,2)))
```

```
[1. 1. 1. 1.]
[1. 1. 1. 1.]
2D -  [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
3D -  [[[1. 1.]
  [1. 1.]]

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

```
print('2D - ', np.zeros((3,3)))
print('3D - ', np.zeros((2,2,2)))
```

```
2D -  [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
3D -  [[[0. 0.]
  [0. 0.]]

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

```
a = np.eye(3)
print(a)

b = np.eye(3,2) #3 rows, 2 cols
print(b)
```

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

```
a = np.diag([1,2,3,4]) #diagonal matrix
print(a)
```

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

```
np.diag(a) #Extract diagonals
```

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

```
a = np.random.rand(4) #random numbers of uniform distribution
print(a)
```

```
[0.5284632  0.12974267 0.2746634  0.23008551]
```

```
a = np.random.randn(4) #random number of standard deviation
print(a)
```

```
[ 0.62550268 -1.73983648 -0.82649306  0.44268514]
```

Basic datatypes

```
a = np.arange(10)
print(a.dtype)
```

```
int64
```

```
a = np.arange(10, dtype='float64')
print(a)
print(a.dtype)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
float64
```

```
a = np.zeros((3,3))
print(a)
print(a.dtype)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
float64
```

```
d = np.array([1+2j, 3+4j])
print(d)
print(d.dtype)
```

```
[1.+2.j 3.+4.j]
complex128
```

```
b = np.array([False, True])
print(b.dtype)
```

```
bool
```

```
s = np.array(["asd"])
print(s.dtype)
```

```
<U3
```

Indexing & Slicing

```
a = np.arange(10)
print(a[5])
```

```
5
```

```
b = np.diag([1,2,3,4,5])
print(b[2,2])
print(b[2][2])
```

```
3
3
```

```
b[2,1] = 10
print(b)
```

```
[[ 1  0  0  0  0]
 [ 0  2  0  0  0]
 [ 0 10  3  0  0]
 [ 0  0  0  4  0]
 [ 0  0  0  0  5]]
```

```
a = np.arange(11)
print(a)
```

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

```
print(a[2:10])
print(a[:10])
print(a[0:10:3]) #with step value
```

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

```
a[5:] = 10
print(a)
```

```
[ 0  1  2  3  4 10 10 10 10 10 10]
```

```
print(a)
b = np.arange(6)
a[5:] = b[:]
print(a)
a[5:] = b[::-1] #in reverse order
print(a)
```

```
[ 0  1  2  3  4 10 10 10 10 10 10]
[0 1 2 3 4 0 1 2 3 4 5]
[0 1 2 3 4 5 4 3 2 1 0]
```

Copies & Views

```
a = np.arange(10)
a
```

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

```
b = a[::2]
print(b)
```

```
[0 2 4 6 8]
```

```
b[0] = 10
print(b)
print(a)
```

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

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

```
True
```

```
a = np.arange(10)
c = a[::2].copy()
print(c)
c[0] = 10
print(a)
print(c)
```

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

Fancy Indexing

```
a = np.random.randint(0, 20, 15)
print(a)
```

```
[ 6 11 11 13  9  2  5  9  6 11  1 13 15 14 18]
```

```
mask = (a%2 == 0)
print(mask)
```

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

```
from_a = a[mask] #creates a copy but not view
from_a = a[mask] #creates a copy but not view
print(from_a)
```

```
[ 6  2  6 14 18]
```

```
from_a[0] = 11
print(from_a)
print(a)
```

```
[11  2  6 14 18]
[ 6 11 11 13  9  2  5  9  6 11  1 13 15 14 18]
```

```
a[mask] = -1
print(a)
```

```
[-1 11 11 13  9 -1  5  9 -1 11  1 13 15 -1 -1]
```

```
a = np.arange(0,100,10)
print(a)
```

```
[ 0 10 20 30 40 50 60 70 80 90]
```

```
new_range = [2,3,4,3,2] #choose the 2nd, 3rd, 4th, 3rd, 2nd and create a copy
new_a = a[new_range]
print(new_a)
```

```
[20 30 40 30 20]
```

```
new_a[0] = 200
print(new_a)
print(a)
```

```
[200  30  40  30  20]
[ 0 10 20 30 40 50 60 70 80 90]
```

```
a[[0, 1, 2, 3]] = [100, 200, 300, 400]
print(a)
```

```
[100 200 300 400  40  50  60  70  80  90]
```

Numpy Operations

```
a = np.array([1,2,3,4])
print(a+1) #Element wise operation as '1' is a scalar
print(a**2)
```

```
[2 3 4 5]
[ 1  4  9 16]
```

```
b = np.ones(4) + 1
```

```
a - b #Elementwise subtraction
```

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

```
a * b #elementwise multiplicationn
```

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

```
c = np.diag([1,2,3,4])
d = np.ones((4,4))+2
c[0,3] = 4
d[2:3] = d[2:3]+1
d[0,3] = 5

print(c)
print(d)
print("---------------------")
print(c*d) #elementwise multiplication provided both matrices have same shape
print(c.dot(d)) #matrix multiplication (i.e. for each elt in the result matrix, we'll do the d
```

```
[[1 0 0 4]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
[[3. 3. 3. 5.]
 [3. 3. 3. 3.]
 [4. 4. 4. 4.]
 [3. 3. 3. 3.]]
---------------------
[[ 3.  0.  0. 20.]
 [ 0.  6.  0.  0.]
 [ 0.  0. 12.  0.]
 [ 0.  0.  0. 12.]]
[[15. 15. 15. 17.]
 [ 6.  6.  6.  6.]
 [12. 12. 12. 12.]
 [12. 12. 12. 12.]]
```

```
c > d #Elementwise comparison
```

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

```
c < d
```

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

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

```
print(np.array_equal(a,b))
print(np.array_equal(a,c)) #all elts are same elementwise
```

```
False
True
```

```
a = np.array([1,1,0,0], dtype="bool")
b = np.array([1,0,1,0], dtype="bool")
print(np.logical_or(a,b))
print(np.logical_and(a,b))
```

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

NumPy Functions

```
a = np.arange(5)
np.sin(a)
```

```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

```
np.log(a)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: RuntimeWarning: divide by
  """Entry point for launching an IPython kernel.
array([      -inf, 0.        , 0.69314718, 1.09861229, 1.38629436])
```

```
np.exp(a)
```

```
array([ 1.        ,  2.71828183,  7.3890561 , 20.08553692, 54.59815003])
```

```
x = np.array([1,2,3,4])
print(np.sum(x))
```

```
10
```

```
x = np.array([[1,2,3,4],[5,6,7,8]])
print(np.sum(x))
print(np.sum(x, axis=0)) #columnwise addition, because column is the innermost
print(np.sum(x, axis=1)) #rowwise addition
```

```
36
[ 6  8 10 12]
[10 26]
```

```
print(np.min(x))
print(np.max(x))
print(np.argmin(x)) #Index
print(np.argmax(x))
```

```
1
8
0
7
```

```
print(np.all([True, True, False]))
print(np.any([True, False, False]))
```

```
False
True
```

```
a = np.zeros((50,50))
print(np.any(a>0))
```

```
False
```

```
a = np.array([1,2,3,2])
b = np.array([2,2,3,2])
c = np.array([6,4,4,5])
((a <= b) & (b <= c)).all()
```

```
True
```

Statistical Functions

```
x = np.array([1,2,3,1])
print(np.mean(x))
print(np.median(x))
print(np.std(x))
```

```
1.75
1.5
0.82915619758885
```

```
y = np.array([[1,2,3],[5,6,1]])
print(np.mean(y))
print(np.median(y, axis=1))
```

```
3.0
[2. 5.]
```

```
population = """# year  hare  lynx  carrot
1900  30e3  4e3 48300
1901  47.2e3  6.1e3 48200
1902  70.2e3  9.8e3 41500
1903  77.4e3  35.2e3  38200
1904  36.3e3  59.4e3  40600
1905  20.6e3  41.7e3  39800
1906  18.1e3  19e3  38600
1907  21.4e3  13e3  42300
1908  22e3  8.3e3 44500
```

```
1909  25.4e3  9.1e3 42100
1910  27.1e3  7.4e3 46000
1911  40.3e3  8e3 46800
1912  57e3  12.3e3  43800
1913  76.6e3  19.5e3  40900
1914  52.3e3  45.7e3  39400
1915  19.5e3  51.1e3  39000
1916  11.2e3  29.7e3  36700
1917  7.6e3 15.8e3  41800
1918  14.6e3  9.7e3 43300
1919  16.2e3  10.1e3  41300
1920  24.7e3  8.6e3 47300"""
f = open("populations.txt",'w')
f.write(population)
f.close()
```

```
data = np.loadtxt("populations.txt")
data
```

```
array([[ 1900., 30000.,  4000., 48300.],
       [ 1901., 47200.,  6100., 48200.],
       [ 1902., 70200.,  9800., 41500.],
       [ 1903., 77400., 35200., 38200.],
       [ 1904., 36300., 59400., 40600.],
       [ 1905., 20600., 41700., 39800.],
       [ 1906., 18100., 19000., 38600.],
       [ 1907., 21400., 13000., 42300.],
       [ 1908., 22000.,  8300., 44500.],
       [ 1909., 25400.,  9100., 42100.],
       [ 1910., 27100.,  7400., 46000.],
       [ 1911., 40300.,  8000., 46800.],
       [ 1912., 57000., 12300., 43800.],
       [ 1913., 76600., 19500., 40900.],
       [ 1914., 52300., 45700., 39400.],
       [ 1915., 19500., 51100., 39000.],
       [ 1916., 11200., 29700., 36700.],
       [ 1917.,  7600., 15800., 41800.],
       [ 1918., 14600.,  9700., 43300.],
       [ 1919., 16200., 10100., 41300.],
       [ 1920., 24700.,  8600., 47300.]])
```

```
year, hare, lynx, carrots = data.T
print(year)
```

```
[1900. 1901. 1902. 1903. 1904. 1905. 1906. 1907. 1908. 1909. 1910. 1911.
 1912. 1913. 1914. 1915. 1916. 1917. 1918. 1919. 1920.]
```

```
populations = data[:, 1:]
print(populations)
```

```
[[30000.  4000. 48300.]
 [47200.  6100. 48200.]
 [70200.  9800. 41500.]
 [77400. 35200. 38200.]
 [36300. 59400. 40600.]
 [20600. 41700. 39800.]
 [18100. 19000. 38600.]
```

```
       [21400. 13000. 42300.]
       [22000.  8300. 44500.]
       [25400.  9100. 42100.]
       [27100.  7400. 46000.]
       [40300.  8000. 46800.]
       [57000. 12300. 43800.]
       [76600. 19500. 40900.]
       [52300. 45700. 39400.]
       [19500. 51100. 39000.]
       [11200. 29700. 36700.]
       [ 7600. 15800. 41800.]
       [14600.  9700. 43300.]
       [16200. 10100. 41300.]
       [24700.  8600. 47300.]]
```

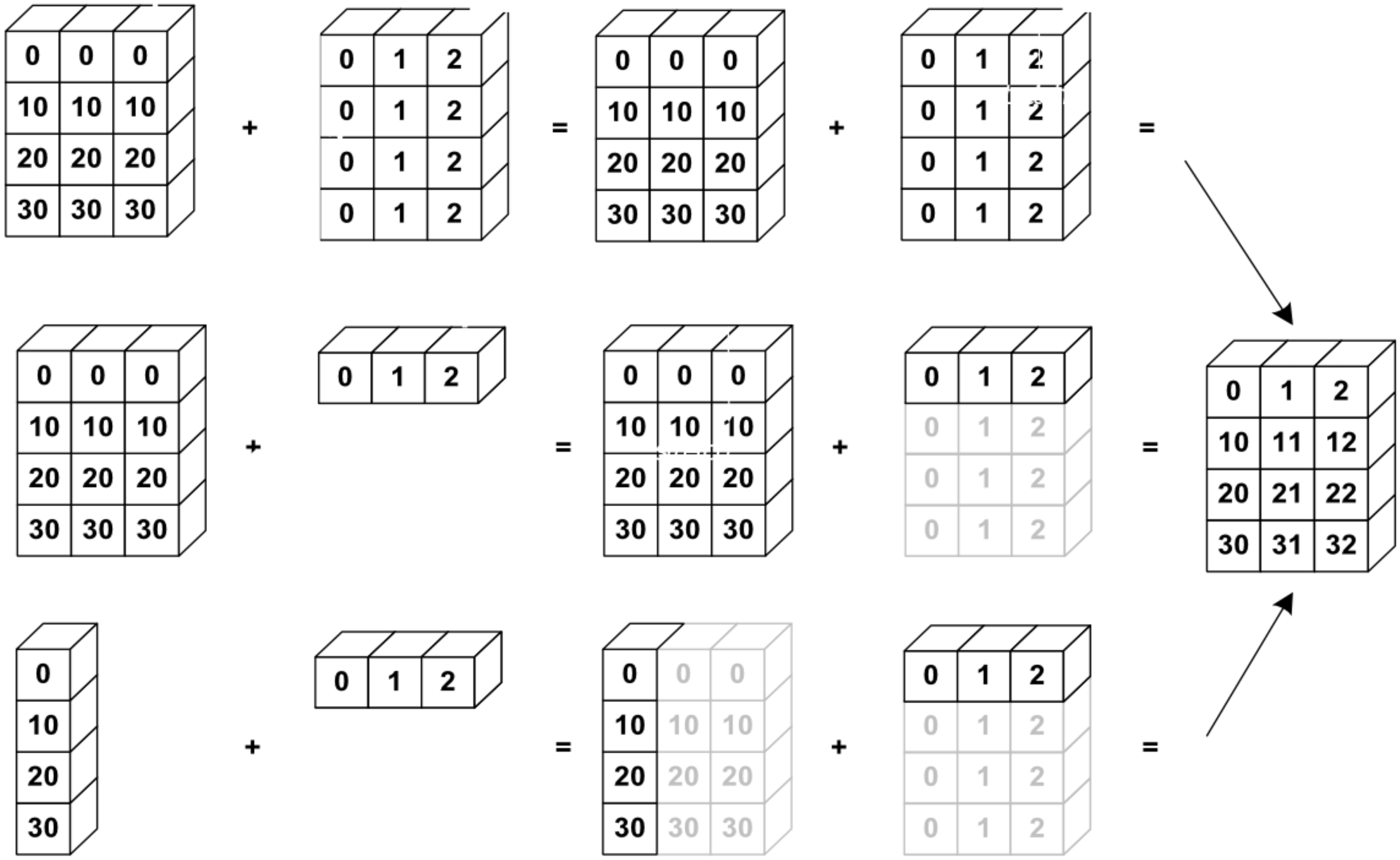
```
populations.std(axis=0)
```

```
array([20897.90645809, 16254.59153691,  3322.50622558])
```

```
#which species has high population each year
populations.argmax(axis=1)
```

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

Broadcasting



```
a = np.tile(np.array([0,10,20,30]), (3,1)) #in the new matrix, have entire row 3 times and ent
print(a)
```

```
[[ 0 10 20 30]
 [ 0 10 20 30]
 [ 0 10 20 30]]
```

```
b = np.array([0,1,2,3])
print(b)
```

```
[0 1 2 3]
```

```
a+b #array b is broadcasted to new rows (same no of rows as in a)
```

```
array([[ 0, 11, 22, 33],
       [ 0, 11, 22, 33],
       [ 0, 11, 22, 33]])
```

```
a.T + np.array([0,1,2])
```

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

```
np.array([[0],[10],[20],[30]]) + np.array([0,1,2])
```

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

```
a = np.arange(0, 40, 10)
print(a.shape)
print(a, '\n--------------')
a = a[:, np.newaxis]
print(a.shape)
print(a)

b = np.array([0, 1, 2])
print(b)

print(a+b)
```

```
(4,)
[ 0 10 20 30]
--------------
(4, 1)
[[ 0]
 [10]
 [20]
 [30]]
[0 1 2]
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

Flattening

```
a = np.array([[1,2,3,4],[5,6,7,8]])
print(a)
print(a.ravel()) #Flattening
```

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

```
print(a.T.ravel())
```

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

Reshaping

```
a.reshape((2,2,2))
```

```
array([[[1, 2],
        [3, 4]],

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

```
a.T.reshape((2,2,2))
```

```
array([[[1, 5],
        [2, 6]],

       [[3, 7],
        [4, 8]]])
```

```
b = a.reshape((2,2,2))
print(b)
b[0,0,0] = 100
print(a) #sometimes reshape may return copy or just a view. So be aware
```

```
[[[100   2]
  [  3   4]]

 [[  5   6]
  [  7   8]]]
[[100   2   3   4]
 [  5   6   7   8]]
```

```
a = np.arange(4*3*2).reshape((4,3,2))
print(a)
```

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

 [[ 6  7]
  [ 8  9]
  [10 11]]

 [[12 13]
  [14 15]
  [16 17]]

 [[18 19]
  [20 21]
  [22 23]]]
```

```
print(a[0,2,1])
```

```
5
```

Resizing

```
a = np.arange(4)
a.resize((8,))
print(a)
```

```
[0 1 2 3 0 0 0 0]
```

```
print(np.resize(a,(3,)))
```

```
[0 1 2]
```

```
a.resize((2,2,2))
print(a)
```

```
[[[0 1]
  [2 3]]

 [[0 0]
  [0 0]]]
```

```
a = np.arange(4)
b = a
a.resize((8,)) #Not allowed as a is referrenced by b
print(a)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-162-190b8fd856c6> in <module>()
      1 a = np.arange(4)
      2 b = a
----> 3 a.resize((8,))
      4 print(a)

ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
```

SEARCH STACK OVERFLOW

Sorting data

```
a = np.array([[5,4,6], [2,3,2]])
b = np.sort(a, axis=1)
print(b)
```

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

```
b = np.sort(a, axis=0)
print(b)
```

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

```
a = np.array([4,3,6,8,1,0])
c = np.argsort(a, axis=0)
print(c)
```

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

```
a[c]
```

```
array([0, 1, 3, 4, 6, 8])
```

✓ 0s completed at 6:46 PM