```
In [ ]: import numpy as np
```

Elementwise Operations

1. Basic Operations

with scalars

```
In [ ]: a = np.array([1, 2, 3, 4]) #create an array

        a + 1

Out[ ]: array([2, 3, 4, 5])
```

```
In [ ]: a ** 2

Out[ ]: array([ 1,  4,  9, 16])
```

All arithmetic operates elementwise

```
In [ ]: b = np.ones(4) + 1

        a - b

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

```
In [ ]: a * b

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

```
In [ ]: # Matrix multiplication

        c = np.diag([1, 2, 3, 4])

        print(c * c)
        print("*****************")
        print(c.dot(c))

        [[ 1  0  0  0]
         [ 0  4  0  0]
         [ 0  0  9  0]
         [ 0  0  0 16]]
        *****************
        [[ 1  0  0  0]
         [ 0  4  0  0]
         [ 0  0  9  0]
         [ 0  0  0 16]]
```

comparisions

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

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

```
In [ ]: a > b

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

```
In [ ]: #array-wise comparisions
        a = np.array([1, 2, 3, 4])
        b = np.array([5, 2, 2, 4])
        c = np.array([1, 2, 3, 4])

        np.array_equal(a, b)

Out[ ]: False
```

```
In [ ]: np.array_equal(a, c)

Out[ ]: True
```

Logical Operations

```
In [ ]: a = np.array([1, 1, 0, 0], dtype=bool)
        b = np.array([1, 0, 1, 0], dtype=bool)

        np.logical_or(a, b)

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

```
In [ ]: np.logical_and(a, b)

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

Transcendental functions:

```
In [ ]: a = np.arange(5)

        np.sin(a)

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

```
In [ ]: np.log(a)
```

```
/Users/satishatcha/.virtualenvs/course/lib/python2.7/site-packages/ipykernel_
launcher.py:1: RuntimeWarning: divide by zero encountered in log
  """Entry point for launching an IPython kernel.
```

```
Out[ ]: array([       -inf,  0.        ,  0.69314718,  1.09861229,  1.38629436])
```

```
In [ ]: np.exp(a)   #evaluates e^x for each element in a given input
```

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

Shape Mismatch

```
In [ ]: a = np.arange(4)

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-f4d5ac434765> in <module>()
      1 a = np.arange(4)
      2
----> 3 a + np.array([1, 2])

ValueError: operands could not be broadcast together with shapes (4,) (2,)
```

Basic Reductions

computing sums

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

```
Out[ ]: 10
```

```
In [ ]: #sum by rows and by columns

        x = np.array([[1, 1], [2, 2]])
        x
```

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

```
In [ ]: x.sum(axis=0)   #columns first dimension
```

```
Out[ ]: array([3, 3])
```

```
In [ ]: x.sum(axis=1)  #rows (second dimension)

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

Other reductions

```
In [ ]: x = np.array([1, 3, 2])
        x.min()

Out[ ]: 1
```

```
In [ ]: x.max()

Out[ ]: 3
```

```
In [ ]: x.argmin()# index of minimum element

Out[ ]: 0
```

```
In [ ]: x.argmax()# index of maximum element

Out[ ]: 1
```

Logical Operations

```
In [ ]: np.all([True, True, False])

Out[ ]: False
```

```
In [ ]: np.any([True, False, False])

Out[ ]: True
```

```
In [ ]: #Note: can be used for array comparisions
        a = np.zeros((50, 50))
        np.any(a != 0)

Out[ ]: False
```

```
In [ ]: np.all(a == a)

Out[ ]: True
```

```
In [ ]: 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()

Out[ ]: True
```

Statistics

```
In [ ]: x = np.array([1, 2, 3, 1])
        y = np.array([[1, 2, 3], [5, 6, 1]])
        x.mean()

Out[ ]: 1.75
```

```
In [ ]: np.median(x)

Out[ ]: 1.5
```

```
In [ ]: np.median(y, axis=-1) # last axis

Out[ ]: array([ 2.,  5.])
```

```
In [ ]: x.std()          # full population standard dev.

Out[ ]: 0.82915619758884995
```

Example:

Data in populations.txt describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years.

```
In [ ]: #Load data into numpy array object
        data = np.loadtxt('populations.txt')
```

```
In [ ]: data

Out[ ]: 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.]])
```

```
In [ ]: year, hares, lynxes, carrots = data.T #columns to variables
        print(year)
```

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

```
In [ ]: #The mean population over time
        populations = data[:, 1:]
        populations
```

```
Out[ ]: array([[ 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.]])
```

```
In [ ]: #sample standard deviations
        populations.std(axis=0)
```

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

```
In [ ]: #which species has the highest population each year?

        np.argmax(populations, axis=1)
```

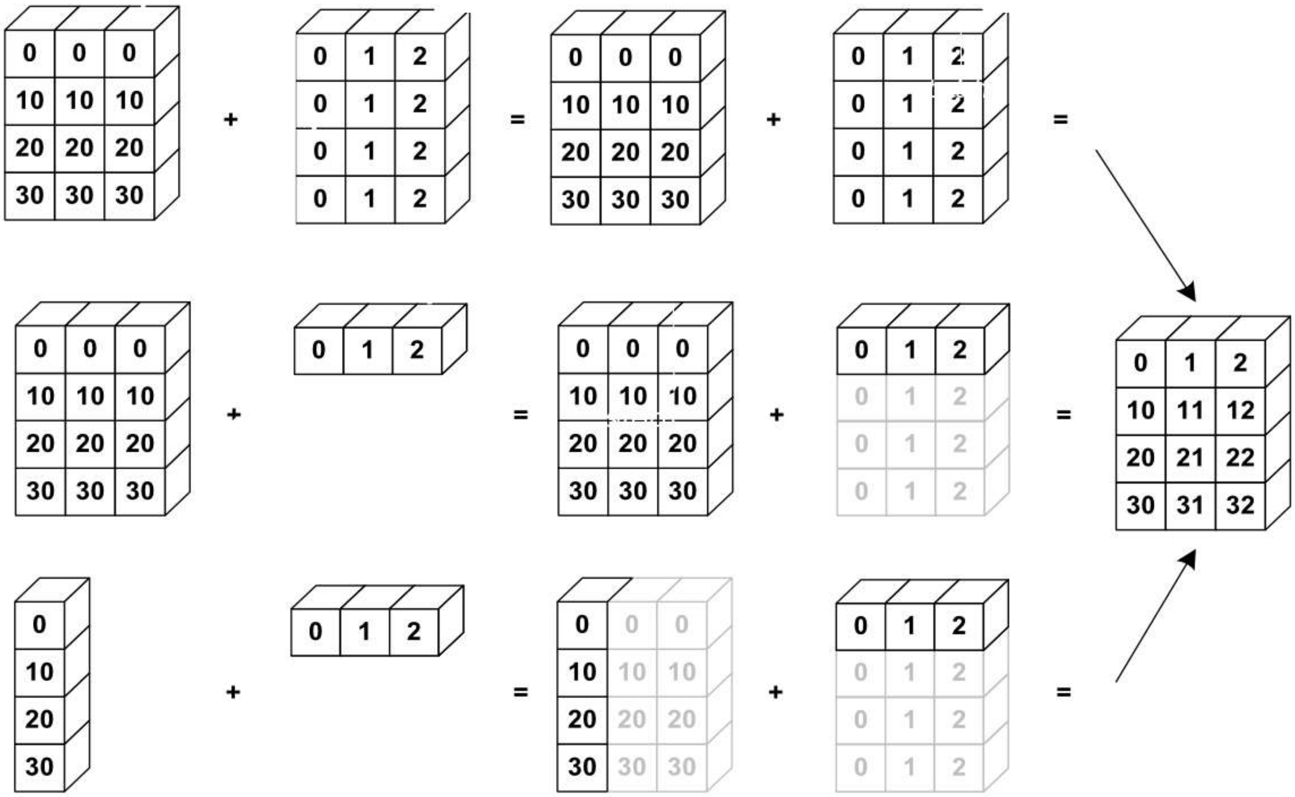
```
Out[ ]: array([2, 2, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 1, 2, 2, 2, 2, 2])
```

Broadcasting

Basic operations on numpy arrays (addition, etc.) are elementwise

This works on arrays of the same size. Nevertheless, It’s also possible to do operations on arrays of different sizes if NumPy can transform these arrays so that they all have the same size: this conversion is called broadcasting.

The image below gives an example of broadcasting:



```
In [ ]: a = np.tile(np.arange(0, 40, 10), (3,1))
        print(a)

        print("*************")
        a=a.T
        print(a)
```

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

```
In [ ]: b = np.array([0, 1, 2])
        b
```

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

```
In [ ]: a + b
```

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

```
In [ ]: a = np.arange(0, 40, 10)
        a.shape
```

```
Out[ ]: (4,)
```

```
In [ ]: a = a[:, np.newaxis]  # adds a new axis -> 2D array
        a.shape
```

```
Out[ ]: (4, 1)
```

```
In [ ]: a
```

```
Out[ ]: array([[ 0],
               [10],
               [20],
               [30]])
```

```
In [ ]: a + b
```

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

Array Shape Manipulation

Flattening

```
In [ ]: a = np.array([[1, 2, 3], [4, 5, 6]])
        a.ravel() #Return a contiguous flattened array. A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.
```

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

```
In [ ]: a.T #Transpose
```

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

```
In [ ]: a.T.ravel()
```

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

Reshaping

The inverse operation to flattening:

```
In [ ]: print(a.shape)
        print(a)
```

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

```
In [ ]: b = a.ravel()
        print(b)
```

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

```
In [ ]: b = b.reshape((2, 3))
        b
```

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

```
In [ ]: b[0, 0] = 100
        a
```

```
Out[ ]: array([[100,   2,   3],
               [  4,   5,   6]])
```

Note and Beware: reshape may also return a copy!:

```
In [ ]: a = np.zeros((3, 2))
        b = a.T.reshape(3*2)
        b[0] = 50
        a
```

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

Adding a Dimension

Indexing with the np.newaxis object allows us to add an axis to an array

newaxis is used to increase the dimension of the existing array by one more dimension, when used once. Thus,

1D array will become 2D array

2D array will become 3D array

3D array will become 4D array and so on

```
In [ ]: z = np.array([1, 2, 3])
        z
```

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

```
In [ ]: z[:, np.newaxis]
```

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

Dimension Shuffling

```
In [ ]: a = np.arange(4*3*2).reshape(4, 3, 2)
        a.shape
```

```
Out[ ]: (4, 3, 2)
```

```
In [ ]: a
```

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

```
In [ ]: a[0, 2, 1]
```

```
Out[ ]: 5
```

Resizing

```
In [ ]: a = np.arange(4)
        a.resize((8,))
        a
```

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

However, it must not be referred to somewhere else:

```
In [ ]: b = a
        a.resize((4,))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-68-702766c88583> in <module>()
      1 b = a
----> 2 a.resize((4,))

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

Sorting Data

In [ ]:

```
#Sorting along an axis:
a = np.array([[5, 4, 6], [2, 3, 2]])
b = np.sort(a, axis=1)
b
```

```
Out[ ]: array([[4, 5, 6],
               [2, 2, 3]])
```

In [ ]:

```
#in-place sort
a.sort(axis=1)
a
```

```
Out[ ]: array([[4, 5, 6],
               [2, 2, 3]])
```

In [ ]:

```
#sorting with fancy indexing
a = np.array([4, 3, 1, 2])
j = np.argsort(a)
j
```

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

In [ ]:

```
a[j]
```

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