Numpy Data X - BKHW

Author: Kunal Desai and Ikhlaq Sidhu 1/22/2017, midified June 2017

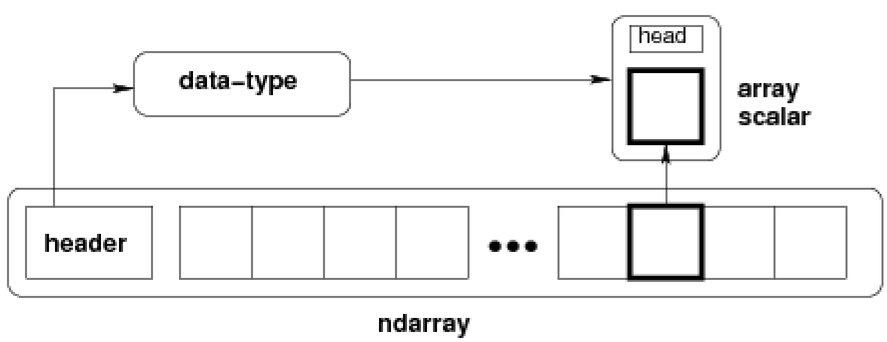
Introduction to NumPy

What is NumPy:

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

NumPy contains an array object that is "fast"



It stores:

- location of a memory block (allocated all at one time)
- a shape (3 x 3 or 1 x 9, etc)
- data type / size of each element

The core feauture that NumPy supports is its multi-dimensional arrays. In NumPy, dimensions are called axes and the number of axes is called a rank.

In [1]:
```
# written for Python 3.6
import numpy as np
```

In [2]:
```
# Creating a NumPy Arracy - simplest possible
# We use a list as an argument input in making a NumPy Array

list1 = [1, 2, 3, 4]
data = np.array(list1)
data
```

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

In [3]:
```
# it could be much longer
list2 = range(10000)
data = np.array(list2)
data
```

Out[3]:
```
array([   0,    1,    2, ..., 9997, 9998, 9999])
```

In [4]:
```
# data = np.array(1,2,3,4, 5,6,7,8,9) # wrong
data = np.array([1,2,3,4,5,6,7,8,9]) # right
data
```

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

In [5]:
```
#accessing elements - similar to slicing Python lists:
print(data[:])
print (data[0:3])
print (data[3:])
print (data[::-2])
```

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

Arrays are like lists, but different

In [6]:
```
# Arrays are faster and more efficient

x = list(range(10000))
# %timeit y = [i**2 for i in x]
y = [i**2 for i in x]
print (y[0:5])
```

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

In [7]:
```
z = np.array(x)
# %timeit y = z**2
y = z**2
print (y[0:5])
```

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

In [8]:
```
# Arrays are different than lists in another way:
# x and y are lists
x = list(range(5))
y = list(range(5,10))
print ("x = ", x)
print ("y = ", y)
print ("x+y = ", x+y)
```

```
x =  [0, 1, 2, 3, 4]
y =  [5, 6, 7, 8, 9]
x+y =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [9]:
```
# now lets try with NumPy arrays:
xn = np.array(x)
yn = np.array(y)
print (xn)
print (yn)
print ("xn + yn = ", xn + yn)
```

```
[0 1 2 3 4]
[5 6 7 8 9]
xn + yn =  [ 5  7  9 11 13]
```

In [10]:
```
# if you need to join to numpy arrays, try hstack, vstack, column_stack, or conc
print (np.hstack((xn,yn)))
print (np.concatenate((xn,yn)))
```

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

In [11]:
```
# An array is a sequence that can be manipulated easily
# An arithmatic operation is applied to each element individually
# When two arrays are added, they must have the same size; corresponding element
# are added in the result

print (3* x)
print (3 * xn)
```

```
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
[ 0  3  6  9 12]
```

In [12]:
```
# all elements must be the same type
# data = np.array([1,2,'cat', 4])
# print (data+1)  # results in error
```

Creating arrays with 2 axis:

In [13]:
```
# This list has two dimensions
list3 = [[1, 2, 3],
 [4, 5, 6]]
```

In [14]:
```
# data = np.array([[1, 2, 3], [4, 5, 6]])
data = np.array(list3)
print (data)
```

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

In [15]:
```
# You can also transpose an array Matrix
print ('Transpose: \n', data.T, '\n')
print ('Transpose: \n', np.transpose(data))

# print (list3.T) # note, this would not work
```

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

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

Remember that every time you declare an np.array, the argument must be in the form of a Python list. Ranges are a great tool to create these list arrays.

In [16]:
```
#Creates array from 0 to before end: np.arange(end)
# See that you don't have to make a list first

# A range is an array of consecutive numbers
# np.arange(end):

np.arange(10)
```

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

In [17]:
```
#Array increasing from start to end: np.arange(start, end)
np.arange(10, 20)
```

Out[17]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

In [18]:
```
#Array increasing from start to end by step: np.arange(start, end, step)
# The range always includes start but excludes end
np.arange(1, 10, 2)
```

Out[18]:
```
array([1, 3, 5, 7, 9])
```

Here is a quick example of a NumPy array and some helpful methods:

In [19]:
```
# Reshape is used to change the shape
a = np.arange(0, 15)
a = a.reshape(3, 5)
# a = np.arange(0, 15).reshape(3, 5)  # same thing
print (a)
```

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

In [20]:
```
# If you want to know the shape, use 'shape'
print (a.shape)
print (len(a.shape))
print (a.shape[1])
```

```
(3, 5)
2
5
```

In [21]:
```
# ndim tells us the number of dimensions of the array
a.ndim
```

Out[21]:
```
2
```

In [22]:
```
#dtype.name tells us what type is each element in the array
print (a.dtype.name)
```

```
int64
```

In [23]:
```
# And for total size:
a.size
```

Out[23]:
```
15
```

In [24]:
```
# Setting the data type
# default is float
d1 = np.array([1,2,3,4,5,6,7,8])
print (d1.dtype, d1)

d2 = np.array([1,2.0,3,4,5,6,7,8])
print (d2.dtype, d2)

d3 = np.array([1,2.0,3,4,5,6,7,8], dtype = np.uint)
print (d3.dtype, d3)

# can be complex, float, int (same as int64), uint.
```

```
int64 [1 2 3 4 5 6 7 8]
float64 [ 1.  2.  3.  4.  5.  6.  7.  8.]
uint64 [1 2 3 4 5 6 7 8]
```

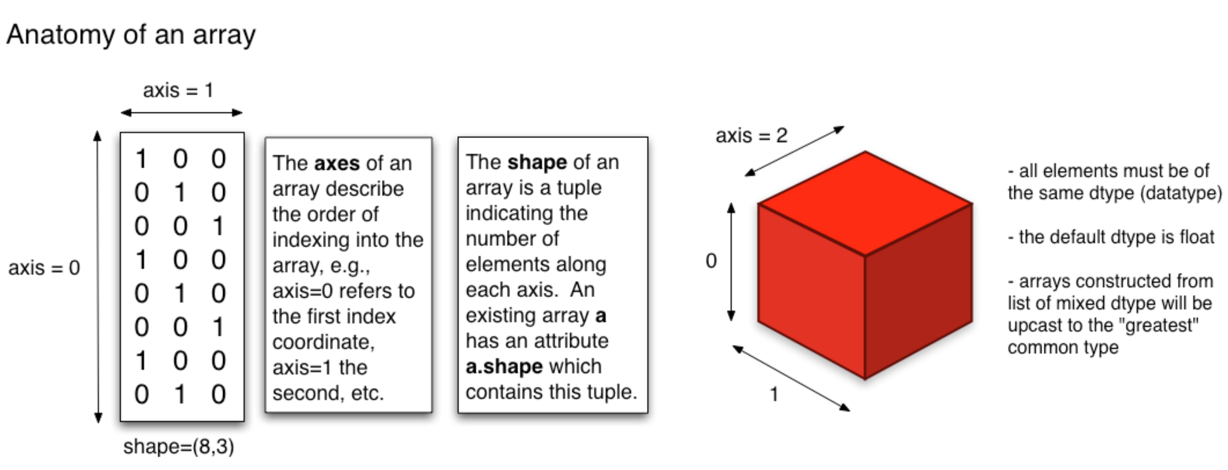
In [25]:
```
# sum, min, max, .. are easy
print (a)
print (a.sum())
print ((0+14)*15/2)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
105
105.0
```

In [26]:
```
print (a.sum(axis=0))
print (a.sum(axis=1))
```

```
[15 18 21 24 27]
[10 35 60]
```

Arrray Axis



To get the cumulative product:

In [27]:
```
print (np.arange(1, 10))
print (np.cumprod(np.arange(1, 10)))
```

```
[1 2 3 4 5 6 7 8 9]
[     1      2      6     24    120    720   5040  40320 362880]
```

To get the cumulative sum:

In [28]:
```
print (np.arange(1, 10))
np.cumsum((np.arange(1, 10)))
```

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

Out[28]:
```
array([ 1,  3,  6, 10, 15, 21, 28, 36, 45])
```

In [29]:
```
print (a[1,:])
print (np.cumsum(a[1,:]))
```

```
[5 6 7 8 9]
[ 5 11 18 26 35]
```

You can also compare arrays

In [30]:
```
#mask
# Does this array have any elements that are "3"?
data1 = np.array(range(10))
print (data1)
mask1 = (data1 > 3)
print (mask1)
```

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

In [31]:
```
# use the mask to get elements:
print (data1[mask1])
```

```
[4 5 6 7 8 9]
```

In [32]:
```
# again:
mask2 = data1 == 0
print (mask2)
print (data1[mask2])
```

```
[ True False False False False False False False False False]
[0]
```

In [33]:
```
# or directly in one step:
print (np.array(range(10))> 5)
print (np.array(range(10))[np.array(range(10)) > 5])
```

```
[False False False False False False  True  True  True  True]
[6 7 8 9]
```

In [34]:
```
# Does this array have any or all elements that are "1"?
print (np.array([1, 1, 0, 1]) == 1)
print (np.all(np.array([1, 1, 1, 1]) == 1))
```

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

Creating a 3D array:

In [35]:
```
a = np.arange(0, 96).reshape(2, 6, 8)
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]
  [24 25 26 27 28 29 30 31]
  [32 33 34 35 36 37 38 39]
  [40 41 42 43 44 45 46 47]]

 [[48 49 50 51 52 53 54 55]
  [56 57 58 59 60 61 62 63]
  [64 65 66 67 68 69 70 71]
  [72 73 74 75 76 77 78 79]
  [80 81 82 83 84 85 86 87]
  [88 89 90 91 92 93 94 95]]]
```

In [36]:
```
# The same methods typically apply in multiple dimensions
print (a.sum(axis = 0))
print ('---')
print (a.sum(axis = 1))
```

```
[[ 48  50  52  54  56  58  60  62]
 [ 64  66  68  70  72  74  76  78]
 [ 80  82  84  86  88  90  92  94]
 [ 96  98 100 102 104 106 108 110]
 [112 114 116 118 120 122 124 126]
 [128 130 132 134 136 138 140 142]]
---
[[120 126 132 138 144 150 156 162]
 [408 414 420 426 432 438 444 450]]
```

Basic Operations

One of the coolest parts of NumPy is the ability for you to run operations on top of arrays. Here are some basic operations:

In [37]:
```
a = np.arange(11, 21)
b = np.arange(0, 10)
print ("a = ",a)
print ("b = ",b)
print (a + b)
```

```
a =  [11 12 13 14 15 16 17 18 19 20]
b =  [0 1 2 3 4 5 6 7 8 9]
[11 13 15 17 19 21 23 25 27 29]
```

In [38]:
```
a * b
```

Out[38]:
```
array([  0,  12,  26,  42,  60,  80, 102, 126, 152, 180])
```

In [39]:
```
a ** 2
```

Out[39]:
```
array([121, 144, 169, 196, 225, 256, 289, 324, 361, 400])
```

You can even do things like matrix operations

In [40]:
```
a.dot(b)
```

Out[40]:
```
780
```

In [41]:
```
# Matrix multiplication
c = np.arange(1,5).reshape(2,2)
print ("c = ", c)
d = np.arange(5,9).reshape(2,2)
print ("d = ", d)
```

```
c =  [[1 2]
 [3 4]]
d =  [[5 6]
 [7 8]]
```

In [42]:
```
print (d.dot(c))
```

```
[[23 34]
 [31 46]]
```

In [43]:
```
# Other ways to create an array:
print (np.zeros(5))
print (np.ones(8).reshape(2,4))
```

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

In [44]:
```
# Radom numbers
rng = np.random.RandomState(0)  # the seed is zero
print(rng.uniform(1,5,10))   # 10 random uniform numbers from 1 to 5
print (rng.exponential(1,5)) # 5 random exp numbers with rate 1
```

```
[ 3.19525402  3.86075747  3.4110535   3.17953273  2.6946192   3.58357645
  2.75034885  4.567092    4.85465104  2.53376608]
[ 1.56889614  0.75267411  0.83943285  2.59825415  0.07368535]
```

In [45]:
```
print (np.random.random(8).reshape(2,4)) #8 random 0-1 in a 2 x 4 array
# https://docs.scipy.org/doc/numpy-1.12.0/reference/routines.random.html
```

```
[[ 0.79367369  0.11808361  0.08561092  0.16551397]
 [ 0.95734802  0.01997461  0.19161898  0.46989086]]
```

In [46]:
```
# linspace: this is how you fill a number an array
# with numbers from a to b with n equally spaced numbers (inclusive)

data = np.linspace(0,5,10)
print (data)
```

```
[ 0.          0.55555556  1.11111111  1.66666667  2.22222222  2.77777778
  3.33333333  3.88888889  4.44444444  5.        ]
```

In [47]:
```
from numpy import pi
x = np.linspace(0,2*pi, 10)
print ("x = ",x)
print ("sin(x) = ", np.sin(x))
```

```
x =  [ 0.          0.6981317   1.3962634   2.0943951   2.7925268   3.4906585
  4.1887902   4.88692191  5.58505361  6.28318531]
sin(x) =  [  0.00000000e+00   6.42787610e-01   9.84807753e-01   8.66025404e-01
   3.42020143e-01  -3.42020143e-01  -8.66025404e-01  -9.84807753e-01
  -6.42787610e-01  -2.44929360e-16]
```

In [48]:
```
# more slicing
x = np.array(range(25))
print (x)
print (x[5:15:2])
print (x[15:5:-1])
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
[ 5  7  9 11 13]
[15 14 13 12 11 10  9  8  7  6]
```

In [49]:
```
# take a slice from 10 to 19 and call it x1
x1 = x[10:20]
print (x1)

#x1 is. shallow copy, its just a window into the original x
x1[:] = 0
print (x1)
```

```
[10 11 12 13 14 15 16 17 18 19]
[0 0 0 0 0 0 0 0 0 0]
```

In [50]:
```
# what happens to x
print (x)
```

```
[ 0  1  2  3  4  5  6  7  8  9  0  0  0  0  0  0  0  0  0  0 20 21 22 23 24]
```

In [51]:
```
# if you actually need to delete a row or column, look up numpy.delete
x = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(x)
print ("---")
x = np.delete(x,0,axis=0)
print (x)
```

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

In [52]:
```
# same thing with assignment, its not a copy, its the same data
x = np.array(range(25))
print (x)
y = x
y[:] = 0
print (x)
x is y
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Out[52]: True

In [53]:
```
# If you want an actual copy: use a deep copy
x = np.array(range(25))
print (x)
y = x.copy()
y[:] = 0
print (x)
x is y
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

Out[53]: False

In [54]:
```
# flatten using ravel()
x = np.array(range(24))
x = x.reshape(4,6)
print(x)

x = x.ravel() # make it flat
print (x)

x = x.reshape(6,4)
print (x)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

HW Section

```
## Numpy Introduction

1a) Create two numpy arrays (a and b). a should be all integers between 10-19
(inclusive), and b should be ten evenly spaced numbers between 1-7. Print all
the results below:
* Square all the elements in both arrays (element-wise)
* Add both the squared arrays (e.g., [1,2] + [3,4] = [4,6])
* Sum the elements with even indices of the added array.
* Take the square root of the added array (element-wise square root)
```

In [31]:

```python
import numpy as np
a = np.arange(10,20)
b = np.linspace(1,7,10)
print('Square all elements')
print(np.square(a))
print(np.square(b))
print()
print('Add both the squared arrays')
add_square = np.add(np.square(a),np.square(b))
print(np.add(np.square(a),np.square(b)))

# M = [x for x.index() in S if x % 2 == 0]
i = [0,2,4,6,8]
sum_even = np.sum(add_square[i])

print()
print('Sum the elements with even indices of the added array')
print(sum_even)

print()
print('Take the square root of the added array (element-wise square root)')
print(np.sqrt(add_square))
```

```
Square all elements
[100 121 144 169 196 225 256 289 324 361]
[  1.           2.77777778   5.44444444   9.          13.44444444
  18.77777778  25.          32.11111111  40.11111111  49.        ]

Add both the squared arrays
[ 101.          123.77777778  149.44444444  178.          209.44444444
  243.77777778  281.          321.11111111  364.11111111  410.        ]

Sum the elements with even indices of the added array
1105.0

Take the square root of the added array (element-wise square root)
[ 10.04987562  11.12554618  12.22474721  13.34166406  14.47219556
  15.61338457  16.76305461  17.91957341  19.08169571  20.24845673]
```

1b) Append b to a, reshape the appended array so that it is a 5x4, 2d array and store the results in a variable called m. Print m.

In [38]:
```
m = np.append(a,b)
m = m.reshape(5,4)
print(m)
```

```
[[ 10.          11.          12.          13.        ]
 [ 14.          15.          16.          17.        ]
 [ 18.          19.           1.           1.66666667]
 [  2.33333333   3.           3.66666667   4.33333333]
 [  5.           5.66666667   6.33333333   7.        ]]
```

1c) Extract the second and the third column of the m matrix. Store the resulting 5x2 matrix in a new variable called m2. Print m2.

In [40]:
```
m2 = m[:,[1,2]]
print(m2)
```

```
[[ 11.          12.        ]
 [ 15.          16.        ]
 [ 19.           1.        ]
 [  3.           3.66666667]
 [  5.66666667   6.33333333]]
```

1d) Take the dot product of m2T and m store the results in a matrix called m3. Print m3.

In [44]:
```
m3 = np.dot(np.transpose(m),m2)
print(m3)
```

```
[[ 697.33333333  402.22222222]
 [ 748.11111111  437.88888889]
 [ 437.88888889  454.55555556]
 [ 482.33333333  489.88888889]]
```

1e) Round the m3 matrix to two decimal points. Store the result in place and print the new m3.

In [46]:
```
m3 = m3.round(2)
print(m3)
```

```
[[ 697.33  402.22]
 [ 748.11  437.89]
 [ 437.89  454.56]
 [ 482.33  489.89]]
```

1f) Sort the m3 array so that the highest value is at the top left, the next highest value to the right of the highest, and the lowest value is at the bottom right. Print the sorted m3 array.

In [165]:
```
kl = np.ravel(m3)
kl = -np.sort(-kl)
kl = kl.reshape(4,2)
print(kl)
```

```
[[ 748.11  697.33]
 [ 489.89  482.33]
 [ 454.56  437.89]
 [ 437.89  402.22]]
```

NumPy and Masks

2a) create an array called 'f' where the values are sin(x) for x from 0 to pi with 100 values in f
* print f
* use a 'mask' and print an array that is True when f >= 1/2 and False when f < 1/2
* create and print an array sequence that has only those values where f>= 1/2

In [93]:

```
x = np.linspace(0,np.pi,100)
f = np.sin(x)
print('f matrix')
print(f)
print()
mask = (f>=.5)
print('mask')
print(mask)

seq = f[np.where(mask==True)]
print()
print('create and print an array sequence that has only those values where f>= 1
print(seq)
```

```
f matrix
[  0.00000000e+00   3.17279335e-02   6.34239197e-02   9.50560433e-02
   1.26592454e-01   1.58001396e-01   1.89251244e-01   2.20310533e-01
   2.51147987e-01   2.81732557e-01   3.12033446e-01   3.42020143e-01
   3.71662456e-01   4.00930535e-01   4.29794912e-01   4.58226522e-01
   4.86196736e-01   5.13677392e-01   5.40640817e-01   5.67059864e-01
   5.92907929e-01   6.18158986e-01   6.42787610e-01   6.66769001e-01
   6.90079011e-01   7.12694171e-01   7.34591709e-01   7.55749574e-01
   7.76146464e-01   7.95761841e-01   8.14575952e-01   8.32569855e-01
   8.49725430e-01   8.66025404e-01   8.81453363e-01   8.95993774e-01
   9.09631995e-01   9.22354294e-01   9.34147860e-01   9.45000819e-01
   9.54902241e-01   9.63842159e-01   9.71811568e-01   9.78802446e-01
   9.84807753e-01   9.89821442e-01   9.93838464e-01   9.96854776e-01
   9.98867339e-01   9.99874128e-01   9.99874128e-01   9.98867339e-01
   9.96854776e-01   9.93838464e-01   9.89821442e-01   9.84807753e-01
   9.78802446e-01   9.71811568e-01   9.63842159e-01   9.54902241e-01
   9.45000819e-01   9.34147860e-01   9.22354294e-01   9.09631995e-01
   8.95993774e-01   8.81453363e-01   8.66025404e-01   8.49725430e-01
   8.32569855e-01   8.14575952e-01   7.95761841e-01   7.76146464e-01
   7.55749574e-01   7.34591709e-01   7.12694171e-01   6.90079011e-01
   6.66769001e-01   6.42787610e-01   6.18158986e-01   5.92907929e-01
   5.67059864e-01   5.40640817e-01   5.13677392e-01   4.86196736e-01
   4.58226522e-01   4.29794912e-01   4.00930535e-01   3.71662456e-01
   3.42020143e-01   3.12033446e-01   2.81732557e-01   2.51147987e-01
   2.20310533e-01   1.89251244e-01   1.58001396e-01   1.26592454e-01
   9.50560433e-02   6.34239197e-02   3.17279335e-02   1.22464680e-16]

mask
[False False False False False False False False False False False False
 False False False False False  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True False
 False False False False False False False False False False False False
 False False False False]

create and print an array sequence that has only those values where f>= 1/2
[ 0.51367739  0.54064082  0.56705986  0.59290793  0.61815899  0.64278761
  0.666769    0.69007901  0.71269417  0.73459171  0.75574957  0.77614646
  0.79576184  0.81457595  0.83256985  0.84972543  0.8660254   0.88145336
```

```
 0.89599377  0.909632    0.92235429  0.93414786  0.94500082  0.95490224
 0.96384216  0.97181157  0.97880245  0.98480775  0.98982144  0.99383846
 0.99685478  0.99886734  0.99987413  0.99987413  0.99886734  0.99685478
 0.99383846  0.98982144  0.98480775  0.97880245  0.97181157  0.96384216
 0.95490224  0.94500082  0.93414786  0.92235429  0.909632    0.89599377
 0.88145336  0.8660254   0.84972543  0.83256985  0.81457595  0.79576184
 0.77614646  0.75574957  0.73459171  0.71269417  0.69007901  0.666769
 0.64278761  0.61815899  0.59290793  0.56705986  0.54064082  0.51367739]
```

NumPy and 2 Variable Prediction

Lets make 2 numpy arrays each of size 100 x will be from 1 to 10 with a uniform noise (0,1/2) y will be from 1 to 20 with a uniform noise (0,1) This has already been done for you just below:

In [95]:
```
x = np.linspace(1,10,100)+ np.random.random(100) /2
y = np.linspace(1,20,100)+np.random.random(100)
print ('x = ',x)
print ('y= ',y)
```

```
x =  [  1.31526592   1.44522136   1.53760299   1.36370773   1.62301601
   1.61585609   1.92415066   1.69118523   2.09666643   2.04576694
   1.91350539   2.02917477   2.53799519   2.65590123   2.57611243
   2.59910548   2.51307632   2.80128654   2.707467     2.77233212
   3.05946438   3.33928794   3.30885136   3.21782272   3.45249516
   3.300006     3.69513896   3.52771777   3.71724207   3.9135878
   4.11892566   4.11355913   4.2426359    4.14521075   4.15539532
   4.64431615   4.56928856   4.6154429    4.93288151   4.57113117
   4.77757379   4.98008775   4.87372314   5.23408363   5.06484165
   5.18950333   5.38440082   5.58827443   5.49425195   5.64203993
   5.55364072   6.06059232   6.2186683    6.2782822    6.12774799
   6.10364541   6.37017008   6.21257754   6.41037773   6.62571865
   6.58367622   6.62553459   6.76756578   7.00582839   6.83463695
   7.15438404   7.13338649   7.1598925    7.62755683   7.4429219
   7.58874036   7.70764278   7.80145161   8.05643187   8.07045901
   7.92126074   8.32675073   8.1754256    8.58302516   8.49173526
   8.3883247    8.4070013    8.74183034   8.93570091   9.02429741
   9.06184623   9.27550728   8.9297271    9.34014182   9.51882464
   9.35225359   9.63890516   9.85687229   9.48528567   9.70241202
  10.10119938   9.77196609  10.20650208  10.04616781  10.06043924]
y=  [  1.79631686   1.20571301   2.0351115    2.26650268   2.16431142
   2.21095816   2.74437505   2.68754133   2.93099375   3.46909089
   2.99652462   3.82937445   3.97346591   3.92336396   3.7544406
   4.39156977   4.61843099   4.92963732   4.65526258   5.50734749
   5.15126674   5.61078624   5.46846097   5.91013383   6.5881338
   6.42143138   6.47649246   6.73882038   6.87439095   7.39984737
   7.08490409   7.47191184   7.49371039   7.35140243   8.37591827
   8.61572509   8.31045573   9.07584284   8.74078452   9.34246617
   8.94541616   9.40441155   9.78920258   9.93264185  10.11173116
  10.58517236  10.42860396  10.99560114  10.88555141  11.22219571
  11.51666072  11.02767264  11.31261135  11.75868758  11.78221543
  11.95915246  11.84847524  12.24209277  13.09842975  12.86944635
  13.03438443  13.26182927  12.96141647  13.31027802  13.47703889
  14.09712021  14.06488716  13.87771416  14.7366846   15.23654109
  14.93276173  15.29238884  15.79789206  15.74046459  15.3184951
  16.2292316   16.21558626  16.000564    15.97527281  16.97542649
  16.51930477  17.50878795  16.98904179  17.65045421  17.13337488
  17.44426298  17.87556359  17.95060831  18.7969806   18.9863357
  18.31825072  19.39889066  19.55154835  19.79171597  19.9685957
  19.94008473  19.76051342  20.14002474  20.66949567  20.0132135 ]
```

3a) Find Expected value of x and the expected value of y

In [96]:
```
exp_x  = np.mean(x)
exp_y = np.mean(y)
print('expected value of x')
print(exp_x)
print()
print('expected value of y')
print(exp_y)
```

```
expected value of x
5.73498510235

expected value of y
11.0325221602
```

```
3b) write code that uses a linear predictor to calculate a predicted value of
y for each x
ie y_predicted = f(x).
```

In [118]:
```
m = sum((x - exp_x)*(y - exp_y)) / sum(np.square(x - exp_x))
b = exp_y - m * exp_x
print('y = ({:>5f})x + ({:>5f})'.format(m,b))
```

```
y = (2.103002)x + (-1.028165)
```

3c) predict y for each value in x, pur the error into an array called y-error

In [125]:
```
pre_y = m*x+b
y_error = pre_y - y
print(y_error)
```

```
[ -5.84747018e-02   8.05425747e-01   1.70306049e-01  -4.26787276e-01
   2.20729901e-01   1.59025839e-01   2.73953167e-01  -1.59139956e-01
   4.50135548e-01  -1.95003334e-01  -5.83442354e-04  -5.90180279e-01
   3.35778825e-01   6.33837457e-01   6.34964784e-01   4.61900419e-02
  -3.61590699e-01  -6.66902442e-02   1.03817698e-02  -7.05291641e-01
   2.54628951e-01   3.83579054e-01   4.61896129e-01  -1.71210185e-01
  -3.55693448e-01  -5.09676097e-01   2.66228388e-01  -3.48186685e-01
  -8.51872023e-02  -1.97728091e-01   5.49041193e-01   1.50747630e-01
   4.00397820e-01   3.37820472e-01  -6.65277201e-01   1.23117645e-01
   2.70603811e-01  -3.97720613e-01   6.04911867e-01  -7.57531618e-01
   7.36677180e-02   4.05596654e-02  -5.67916386e-01   4.64833012e-02
  -4.88522279e-01  -6.99799681e-01  -1.33361395e-01  -2.71611874e-01
  -3.59291648e-01  -3.85137476e-01  -8.65506217e-01   6.89602274e-01
   7.37097724e-01   4.16389690e-01   7.62880272e-02  -1.51336784e-01
   5.19842439e-01  -2.05192579e-01  -6.45555269e-01   3.62905981e-02
  -2.17062816e-01  -3.56479416e-01   2.42625329e-01   3.94830627e-01
  -1.31946265e-01  -7.95986930e-02  -9.15235435e-02   1.51391656e-01
   2.75920441e-01  -6.12223763e-01  -1.78783669e-03  -1.11362857e-01
  -4.19585888e-01   1.74065681e-01   6.25534281e-01  -5.98966560e-01
   2.67425211e-01   1.64210362e-01   1.04668440e+00  -1.45452167e-01
   9.31968981e-02  -8.57009351e-01   3.66883085e-01   1.13180950e-01
   8.16578928e-01   5.84656078e-01   6.02685168e-01  -1.99536098e-01
  -1.82805244e-01   3.61005123e-03   3.21395715e-01  -1.56415297e-01
   1.49312411e-01  -8.72302759e-01  -5.92565254e-01   2.74596504e-01
  -2.38210604e-01   2.96108316e-01  -5.70545970e-01   1.15749064e-01]
```

3d) write code that calculates the mean square, that is average of y-error squared

In [133]:
```
rms_error = np.sqrt(sum(y_error ** 2)/len(y))
print('Root mean Square error')
print(rms_error)
```

```
Root mean Square error
0.422506100904
```