- Let's try using **arrays** to implement vectors.
- To use arrays in Python you must import the **numpy** module, usually as follows:

```
import numpy as np
```

- To create two vectors enter:

```
In [1]: a = np.array([1, 2, 3])
In [2]: print(type(a))
<class 'numpy.ndarray'>
In [3]: a
array([1, 2, 3])
In [4]: b = np.array([4, 5, 6])
In [5]: b
array([4, 5, 6])
```

```
In [1]: a
array([1, 2, 3])
In [2]: b
array([4, 5, 6])
In [3]: a + b
array([5, 7, 9])
In [4]: a - b
array([-3, -3, -3])
In [5]: a * b
array([ 4, 10, 18])
In [6]: a + 4
array([5, 6, 7])
In [6]: b**2
array([16, 25, 36])
```

- Suppose **ar** is an array.

```
In [1]: import numpy as np
In [2]: ar = np.array([1, 2, 3, 4])
In [3]: ar
array([1, 2, 3, 4])
In [4]: ax = ar
In [5]: ax
array([1, 2, 3, 4])
In [6]: ax[1] = 99
In [7]: ar[2] = -11
In [8]: ar
array([  1,  99, -11,   4])
In [9]: ax
array([  1,  99, -11,   4])
```

- **ax** and **ar** refer to the same array, a change to **ar** is a change to **ax** and a change to **ax** is a change to **ar**.

- To make a copy of the array **ar** named **ac** do the following:

```
In [1]: ac = ar.copy()
In [2]: ar
array([  1,  99, -11,   4])
In [3]: ac
array([  1,  99, -11,   4])
In [4]: ac[0] = 42
In [5]: ar[3] = 50
In [6]: ar
array([  1,  99, -11,  50])
In [7]: ac
array([ 42,  99, -11,   4])
```

- A change to **ar** does not affect **ac**.
- A change to **ac** does not affect **ar**.

- The assignment **a = (2*b**2 -5*b +10) / (b-1)** can be implemented using in place arithmetic by breaking up the expression into several statements.

```
a = b.copy()
a **= 2
a *= 2
a -= 5 * b
a += 10
a /= b - 1
```

- This creates 3 new arrays, one for the copy, the second for **5 * b** and the third for **b – 1**.
- The expression **(2*b**2 -5*b +10) / (b-1)** creates 7 new arrays.

- What values are assigned to **a** and **b** by the following statements?

```
In [1]: b
array([4, 5, 6, 7])
In [2]: a = b
In [3]: a += 5
In [4]: a
array([ 9, 10, 11, 12])
In [5]: b
array([ 9, 10, 11, 12])
```

- **a** and **b** refer to the same array (**a = b)**
- Therefore when the elements of **a** are updated in place so are the elements of **b**.

- To create an array **b** that is the same size and type as an array **a** do the following:

```
In [1]: a
array([ 9, 10, 11, 12])
In [2]: b = np.zeros(a.shape, a.dtype)
In [3]: b
array([0, 0, 0, 0])
```

- The new array **b** is of the same type and size as **a** and each array element has been initialized to zero.

```
In [4]: a.shape
(4,)
In [5]: a.dtype
dtype('int64')
```

- **shape** is the size of an array as a tuple (more on this later)
- **dtype** is the data type of the elements in the array

- The ***asarray*** function can be used to ensure a variable refers to an array.

```
In [1]: a = [1, 2, 3]
In [2]: print(type(a))
<class 'list'>
In [3]: a = np.asarray(a)
In [4]: print(type(a))
<class 'numpy.ndarray'>
In [5]: a
array([1, 2, 3])
```

- If **a** is already an array **a = np.asarray(a)** does nothing, but if **a** is a list or tuple **a** is converted into an array.
- Often used for parameters that must be arrays but could be passed lists or tuples.

- A list of integers or an array of integers can be used as an index to an array.

```
In [1]: a = np.arange(.1, .8, .1)
In [2]: a
array([ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7])
In [3]: b = range(1, 5)
In [4]: list[b]
[1, 2, 3, 4]
In [5]: a[b]
array([ 0.2,  0.3,  0.4,  0.5])
In [6]: c = np.arange(2,6)
In [7]: c
array([2, 3, 4, 5])
In [8]: a[c]
array([ 0.3,  0.4,  0.5,  0.6])
```

- Boolean arrays can be used to generate a list of indices.
- The indices correspond to the indices for which the elements of the boolean array are True.

```
In [1]: a
array([ 0.1, 0.2, -1. , -2. , -3. , 0.6, 0.7])
In [2]: a<0
array([False, False,  True,  True,  True,
 False, False], dtype=bool)
In [3]: a[a<0]
array([-1., -2., -3.])
In [4]: a[a<0] = len(a)
In [5]: a
array([ 0.1, 0.2,  7. ,  7. ,  7. , 0.6, 0.7])
```

- Assume ***numpy*** has been imported as ***np***.
- How do we determine if a variable refers to an array?

```
In [1]: a
array([ 0.1, 0.2,  7. ,  7. ,  7. , 0.6, 0.7])
In [2]: print(type(a))
<class 'numpy.ndarray'>
In [3]: print(type(a) is np.ndarray)
True
In [4]: print(isinstance(a, np.ndarray))
True
In [5]: print(isinstance(a, (float, int)))
False
```

- Notice that we test for a type of ***np.ndarray*** as ***numpy*** was imported as ***np***.
- The function ***isinstance*** can check to see if a variable is one of many types.

```
# typeTest.py
import numpy as np

def two(x):
    """(float, int) -> (float, int); array ->
 array"""
    if isinstance(x, (float, int)):
        result = 2
    elif isinstance(x, np.ndarray):
        result = np.zeros(x.shape, x.dtype) + 2
    else:
        result = 'ERROR: Invalid type!'
    return result
```

```
In [1]: two(0)
2
In [2]: two(np.arange(3.))
Array([2., 2., 2.])
In [3]: two(np.arange(5))
array([2, 2, 2, 2, 2])
In [4]: two('hello')
'ERROR: Invalid type!'
```

- Every array has a *shape* attribute, which is a tuple, that specifies the dimensions of the array.
- So far we have been working with 1 dimensional arrays which are often referred to as ***vectors***.

```
In [1]: ar = np.arange(-.5, .76, .25)
In [2]: ar
array([-0.5 , -0.25,  0.  ,  0.25,  0.5 ,
  0.75])
In [3]: print(ar.shape)
(6,)
In [4]: print(ar.size)
6
In [5]: ar = ar.reshape(2, 3) # 2 dimensional
  array
In [6]: ar
array([[-0.5 , -0.25,  0.  ],
       [ 0.25,  0.5 ,  0.75]])
In [7]: print(ar.shape)
(2, 3)
```

- ar is now a 2 dimensional array with 2 rows and 3 columns.

```
In [1]: ar
array([[-0.5 , -0.25,  0.  ],
       [ 0.25,  0.5 ,  0.75]])
In [2]: print(ar.shape)
(2, 3)
In [3]: print(ar.size)
6
In [4]: print(len(ar))
2
```

- The value returned by the *len* function is the length of the first dimension, which for **ar** is 2, the number of rows.
- A 2 dimensional array (or an array of rank 2) is often referred to as a table or a matrix.

- Another way to create a two dimensional array is to convert a list of lists into an array.

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

In [3]: ar.shape
(2, 3)

In [4]: ar = np.array([[1,2,3],[4,5],[6]]) # 1D
  array of
                                          # lists
In [5]: print(ar)
[[1, 2, 3] [4, 5] [6]]

In [6]: ar.shape
(3,)
```

- There is no restriction on the number of dimensions an array may have.
- Arithmetic operations can be performed on arrays with 2 or more dimensions, just as for vectors, with the operations being performed by position.

```
In [1]: ar
array([[-0.5 , -0.25,  0.  ],
       [ 0.25,  0.5 ,  0.75]])
In [2]: ar + 2
array([[ 1.5 ,  1.75,  2.  ],
       [ 2.25,  2.5 ,  2.75]])
In [3]: ar + ar
array([[-1. , -0.5,  0. ],
       [ 0.5,  1. ,  1.5]])
```

- An array with two dimensions is similar to a list of lists, but the storage of the array in memory is much more efficient than for lists. This is one of the reasons using arrays is faster than using lists.
- Referring to an element in a 2 dimensional array is similar to a list of lists, but the syntax can be different.

```
In [1]: ar
array([[-0.5 , -0.25,  0.  ],    # Row 0
       [ 0.25,  0.5 ,  0.75]])   # Row 1
In [2]: print(ar[1][1])
0.5
In [3]: print(ar[1,1])   # preferred syntax ...
0.5                      # ... more efficient
In [4]: ar[1]  # row 1 of the array
array([ 0.25,  0.5 ,  0.75])
```

- Display all of the elements in a 2 dimensional array.

```
# displayArray1.py
import numpy as np
def display2Darray(array):
    for row in range(array.shape[0]):
        for column in range(array.shape[1]):
            print('array[%d,%d]=%g' %
                (row, column, array[row,
 column]))
In [1]: ar
array([[-0.5 , -0.25,  0.  ],
       [ 0.25,  0.5 ,  0.75]])
In [2]: display2Darray(ar)
array[0,0]=-0.5
array[0,1]=-0.25
array[0,2]=0
array[1,0]=0.25
array[1,1]=0.5
array[1,2]=0.75
```

- Display all of the elements in a 2 dimensional array.

```
# displayArray2.py
import numpy as np
def display2Darray(array):
    for indices, element in np.ndenumerate(array):
        print('array[%d,%d]=%g' %
              (indices[0], indices[1], element))
In [1]: ar
array([[-0.5 , -0.25,  0.  ],
       [ 0.25,  0.5 ,  0.75]])
In [2]: display2Darray(ar)
array[0,0]=-0.5
array[0,1]=-0.25
array[0,2]=0
array[1,0]=0.25
array[1,1]=0.5
array[1,2]=0.75
```

- Slices work on arrays of any number of dimensions.
- There is one set of slice parameters (start, stop, step) for each dimension.

```
In [1]: ar = np.linspace(1, 12, 12).reshape(3,4)
In [2]: ar
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
In [3]: ar[0:, 2]   # column 2
array([  3.,   7.,  11.])
In [4]: ar[:, 0]    # column 0
array([ 1.,  5.,  9.])
In [5]: ar[1,:]     # row 1
array([ 5.,  6.,  7.,  8.])
```

```
In [1]: ar
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
In [2]: ar[1:,:]
array([[  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
In [3]: ar[1:,1:-1]
array([[  6.,   7.],
       [ 10.,  11.]])
In [4]: ar[1:,:-1:2]
array([[  5.,   7.],
       [  9.,  11.]])
```

- For a 2 dimensional array A with **m** rows and **n** columns, A*A means to multiply A_{ij} by A_{ij}, where $0 \leq i < m$ and $0 \leq j < n$.
- This is shown below:

```
In [1]: ar = np.linspace(1,9,9).reshape(3,3)
In [2]: ar
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
In [3]: ar*ar
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.],
       [ 49.,  64.,  81.]])
```

- To multiply two 2 dimensional arrays as if they were matrices use the ***dot*** function.
- Must follow the standard rules for number of rows and columns for matrix multiplication.

```
In [1]: ar
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
In [2]: np.dot(ar, ar)
array([[  30.,   36.,   42.],
       [  66.,   81.,   96.],
       [ 102.,  126.,  150.]])
In [3]: vec  # acts as a column vector below
array([ 1.,  2.,  3.])
In [4]: np.dot(ar, vec)
array([ 14.,  32.,  50.])
```

- Let M1 and M2 be 2-dimensional arrays.
- Matrix multiplication, M1 × M2, can be performed only if the number of columns in M1 is equal to the number of rows of M2.
- Let M3 = M1 × M2
- If M1 has **n** rows and **m** columns and M2 has **m** rows and **k** columns then M3 has **n** rows and **k** columns.
- $M3_{n\times k} = M1_{n\times m} \times M2_{m\times k}$
- For example:

- The program to do this without using *dot* follows.

```
# matrixMultiply.py

def multiply(M1, M2):
    M3 = np.zeros((M1.shape[0], M2.shape[1]), int)
    for row in range(M1.shape[0]):
        for column in range(M2.shape[1]):
            M3[row,column] = \
                np.sum(M1[row]*M2[:,column])
    return M3

def displayMatrix(MM):
    for row in range(MM.shape[0]):
        for column in range(MM.shape[1]):
            print(MM[row, column], end=' ')
        print()
```

```
import numpy as np

M1 = np.array([1,2,3,3,2,1]).reshape(2,3)
M2 = np.array([1,4,3,2,2,1,4,3,3,2,1,4]).\
                    reshape(3,4)

if M1.shape[1] == M2.shape[0]:
    M3 = multiply(M1,M2)
    print('\nThe first matrix is:')
    displayMatrix(M1)
    print('\nThe second matrix is:')
    displayMatrix(M2)
    print('\nThe product of the matrices is:')
    displayMatrix(M3)
else:
    print('The matrices cannot be multiplied!')
```

- The output from this program is:

```
The first matrix is:
1 2 3
3 2 1

The second matrix is:
1 4 3 2
2 1 4 3
3 2 1 4

The product of the matrices is:
14 12 14 20
10 16 18 16
```

- In Python matrix objects are supported.

```
In [1]: xx = np.arange(1.,4.)
In [2]: xx
array([ 1.,  2.,  3.])
In [3]: mm = np.matrix(xx) # row vector
In [4]: print(type(mm))
<class 'numpy.matrixlib.defmatrix.matrix'>
In [5]: mm = np.mat(xx)
In [6]: print(type(mm))
<class 'numpy.matrixlib.defmatrix.matrix'>
In [7]: mm  # row vector
matrix([[ 1.,  2.,  3.]])
In [8]: mm = mm.transpose() # column vector
In [9]: mm
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```

- Perform some matrix multiplication:

```
In [1]: mm = np.mat(np.linspace(1., 9., 9).reshape(3,3))
In [2]: rr = np.mat(np.arange(1.,4.))
In [3]: cc = rr.transpose()
In [4]: mm
matrix([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.]])
In [5]: rr
matrix([[ 1.,  2.,  3.]])
In [6]: cc
matrix([[ 1.],
        [ 2.],
        [ 3.]])
In [7]: mm*cc
matrix([[ 14.],
        [ 32.],
        [ 50.]])
In [8]: rr*mm
matrix([[ 30.,  36.,  42.]])
```

- The *eye* function creates an identity matrix of the specified size.

```
In [1]: ii = np.eye(3)
In [2]: ii
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
In [3]: mm
matrix([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.]])
In [4]: ii * mm  # result should be same
 as mm
matrix([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.]])
```

- What happens if you multiply an array and a matrix?

```
In [1]: aa = np.linspace(1.,9., 9).reshape(3,3)
In [2]: aa
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
In [3]: print(type(aa))
<class'numpy.ndarray'>
In [4]: mm
matrix([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.]])
In [5]: print(type(mm))
<class 'numpy.matrixlib.defmatrix.matrix'>
In [6]: aa*mm   # does matrix multiplication
matrix([[  30.,   36.,   42.],
        [  66.,   81.,   96.],
        [ 102.,  126.,  150.]])
In [7]: mm*aa   # does matrix multiplication
matrix([[  30.,   36.,   42.],
        [  66.,   81.,   96.],
        [ 102.,  126.,  150.]])
```

- To solve n equations in n unknowns use matrices.
- The set of equations:

$3x + 2y + z = 2$

$x + y + z = 2$

$2x – y + z = 7$

can be represented as:

- One matrix and two column vectors are required.

- To find the values for x, y and z invert the matrix and multiply by the constants.

$$x = 0.4(2) - 0.6(2) + 0.2(7) = 1$$
$$y = 0.2(2) + 0.2(2) - 0.4(7) = -2$$
$$z = -0.6(2) + 1.4(2) + 0.2(7) = 3$$

```
# nUnknowns.py (solve n equations in n unknowns)
import numpy as np
from numpy import linalg

def solveUnknowns(aMatrix, aVector):
    return linalg.inv(aMatrix) * aVector

def displaySolution(coefficients, constants, solution):
    indices = range(len(coefficients))
    last = indices[-1]
    for row in indices:
        for column in indices:
            print('%g(%g)' % (coefficients[row, column],
                              solution[column]), end=' ')
            if column != last:
                print('+', end=' ')
        print('= %g' % constants[row])

coefficients = np.mat(np.array([3., 2., 1., 1., 1., 1.,
                        2., -1., 1.]).reshape(3,3))
constants = np.mat(np.array([2., 2., 7.])).transpose()
solution = solveUnknowns(coefficients, constants)
displaySolution(coefficients, constants, solution)
```

- The output from the program is:

```
3(1) + 2(-2) + 1(3) = 2
1(1) + 1(-2) + 1(3) = 2
2(1) + -1(-2) + 1(3) = 7
```

- The solution is x = 1, y = -2 and z = 3