Numpy Review

Background review:

The documentation for numpy arrays seems incomplete. The assumption is you know matlab and the convention comes from there.

numpy arrays

source: http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

Images are represented as numpy arrays. A numpy array is an ndimensional array or np.ndarray.

1) a numpy array consists of a sequence of numbers/data/objects stored in consecutive memory locations called a data buffer.
   Taken from: http://stackoverflow.com/questions/22053050/difference-between-numpy-array-shape-r-1-and-r

1. The meaning of shapes in NumPy

You write, "I know literally it's list of numbers and list of lists where all list contains only a number" but that's a bit of an unhelpful way to think about it.

The best way to think about NumPy arrays is that they consist of two parts, a *data buffer* which is just a block of raw elements, and a *view* which describes how to interpret the data buffer.

For example, if we create an array of 12 integers:

```
>>> a = numpy.arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Then `a` consists of a data buffer, arranged something like this:

0	1	2	3	4	5	6	7	8	9	10	11

and a view which describes how to interpret the data:

```
>>> a.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> a.dtype
dtype('int64')
>>> a.itemsize
8
>>> a.shape
(12,)
```

Here the *shape* `(12,)` means the array is indexed by a single index which runs from 0 to 11. Conceptually, if we label this single index `i`, the array `a` looks like this:

```
i= 0   1   2   3   4   5   6   7   8   9   10  11
```

If we reshape an array, this doesn't change the data buffer. Instead, it creates a new view that describes a different way to interpret the data. So after:

```
>>> b = a.reshape((3, 4))
```

the array `b` has the same data buffer as `a`, but now it is indexed by *two* indices which run from 0 to 2 and 0 to 3 respectively. If we label the two indices `i` and `j`, the array `b` looks like this:

```
i= 0   0   0   0   1   1   1   1   2   2   2   2
j= 0   1   2   3   0   1   2   3   0   1   2   3
```

which means that:

```
>>> b[2,1]
9
```

You can see that the second index changes quickly and the first index changes slowly. If you prefer this to be the other way round, you can specify the `order` parameter:

```
>>> c = a.reshape((3, 4), order='F')
```

which results in an array indexed like this:

```
i= 0   1   2   0   1   2   0   1   2   0   1   2
j= 0   0   0   1   1   1   2   2   2   3   3   3
```

which means that:

```
>>> c[2,1]
5
```

It should now be clear what it means for an array to have a shape with one or more dimensions of size 1. After:

```
>>> d = a.reshape((12, 1))
```

the array `d` is indexed by two indices, the first of which runs from 0 to 11, and the second index is always 0:

```
i= 0   1   2   3   4   5   6   7   8   9  10  11
j= 0   0   0   0   0   0   0   0   0   0   0   0
```

0	1	2	3	4	5	6	7	8	9	10	11

and so:

```
>>> d[10,0]
10
```

A dimension of length 1 is "free" (in some sense), so there's nothing stopping you from going to town:

```
>>> e = a.reshape((1, 2, 1, 6, 1))
```

giving an array indexed like this:

```
i= 0   0   0   0   0   0   0   0   0   0   0   0
j= 0   0   0   0   0   0   1   1   1   1   1   1
k= 0   0   0   0   0   0   0   0   0   0   0   0
l= 0   1   2   3   4   5   0   1   2   3   4   5
m= 0   0   0   0   0   0   0   0   0   0   0   0
```

0	1	2	3	4	5	6	7	8	9	10	11

and so:

```
>>> e[0,1,0,0,0]
6
```

See the NumPy internals documentation for more details about how arrays are implemented.

2) You can use np.zeros(shape, dtype, XX) to create an array with a specific shape. Shape is a tuple; (num elements in array, array dimensions). For example arr=np.zeros(2,3,3) gives you 2 3x3 arrays.

```
>>> arr=np.zeros((2,3,3))
>>> arr
array([[[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]],

       [[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]]])
```

The shape tuple embeds additional arrays as you add more entries to the tuple
for: [[0],[0],[0],[0]] shape=(4,1)
if we add another 1 to the shape tuple we get another nested array. Be careful: this is a little inconsistent from the above blog post which shows you can add indexes to allocated buffers at will. This may be true but there are various nestings of array which are not shown.

shape=(4,1,1) = [[[0]],[[0]],[[0]],[[0]]]

this still contains the same data but we have nested the arrays. Adding additional dimensions introduces 0 rows into the array.

2) we can reshape the arrays using the reshape/transpose commands as seen in example 1) above. when reading the CIFR array we get a dictionary with the following keys:

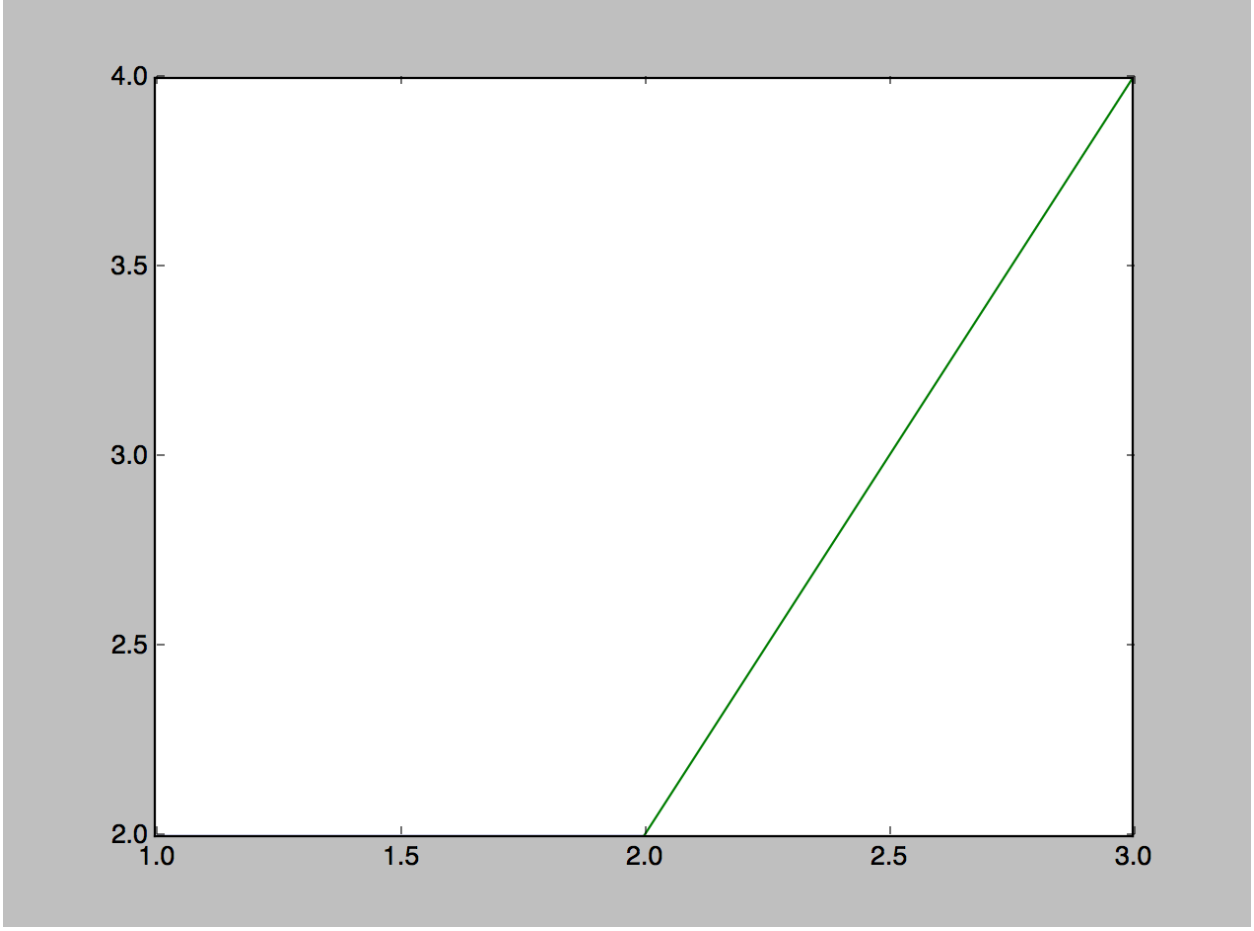
Python Plotting
the class uses the matplotlib which closely resembles the matlab plotting libraries.
Disclaimer: haven't verified this

you can either embed the plot inside an iPyhton notebook or start a separate window.

We cover some basic notes for creating a separate window to plot the CIFAR images
as a test case

the default is when given an array or tuples it will draw lines:

```
>>> a=plt.plot((1,2),(2,2),(3,2),(4,2))
>>> plt.show()
```



Plotting points:
>>> a=plt.scatter((1,2),(2,2),(3,2),(4,2))
>>> plt.show()

Plotting images:
An image is represented as an array of RGB values where each RGB value is an 8 bit binary value ranging from 0-255. One pixel is [0,0,0] for white or [255,255,255] for black.

There are 2 image libraries,
1) the python PIL Image library
2) the numpy/matplotlib image object based in scimage.

The conventions for both of these libraries are slightly different.

PIL image:
to load an image in PIL;
>>> im=Image.open('/Users/dc/Downloads/image_tutorial-5.png')
>>>im.show()

Create a white 100x100 image

```
>>> im1 = Image.new("RGB", (100,100),'white')
>>> im1.show()
>>> im1.getpixel((10,10))
(255, 255, 255)
```

The PIL image pixels take the format of (R,G,B) for value. This isn't clear in the docs.

white is (255,255,255)
put some black pixels in the white

```
>>> im1.putpixel((10,11),(0,0,0))
>>> im1.putpixel((10,12),(0,0,0))
>>> im1.putpixel((10,13),(0,0,0))
```

put some red pixels in the white

```
>>> im1.putpixel((10,13),(255,0,0))
>>> im1.putpixel((10,14),(255,0,0))
>>> im1.putpixel((10,15),(255,0,0))
```


Numpy Image library:
The numpy library uses scikit-image.

To load an image in scikit/numpy

Images are supported using the Pillow library. Default are PNG images. There are differences between PIL and Numpy Images.

To display an PNG image.

Assuming image_tutorial-5.png exists;
```
>>> img=plt.imread("/Users/dc/Downloads/image_tutorial-5.png")
>>> plt.imshow(img)
<matplotlib.image.AxesImage object at 0x1147944d0>
>>> plt.show()
```

im.flags to give stats on the image.
im.shape
im.size

Numpy and Python arrays:

To create an array using python, this is known as array.array:
```
a=([[0,1,2],[3,4,5],[6,7,8]])

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

To create an array using numpy:
There are 2 numpy array functions np.array and np.ndarray. A call to np.array creates a ndarray but the behavior of the 2 functions is very different.

np.array(a)

```
>>> np.array(10)
array(10)
>>> x=np.array(10)
>>> type(x)
<type 'numpy.ndarray'>
>>> np.ndim(x)
0
>>> x
array(10)
```

A scalar object in np.array has 0 dimension. It creates an ndarray object.

On the other hand using ndarray also creates an ndarray for a scalar object. ndarray creates a scalar object of dimension 1. ndarray expects the argument to be a sequence so it creates an array of 10 objects.

```
>>> x=np.ndarray(10)
>>> type(x)
<type 'numpy.ndarray'>
>>> np.ndim(x)
1
>>> x
array([  2.12261997e-314,   0.00000000e+000,   2.14845100e-314,
        2.14852420e-314,   2.14850390e-314,   2.14852378e-314,
        2.12199580e-314,   0.00000000e+000,   2.12337453e-314,
        0.00000000e+000])
```

The values in the x array are not initialized. This is confusing; most arrays aren't created using a single scalar value. You have to use other methods like np.zeros(10), np.ones(10), np.arange(10) to create an initialized array with 10 elements.

ndarray takes a shape, type and order:
```
>>> np.ndarray(shape=(3,3),dtype=int, order='C')
```

A python array is displayed using brackets:

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

A numpy array uses the word array:

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

As a nicely inconsistent language, from a language with no explicit type decls; if you print the numpy array, it shows you the python representation

```
>>> print d
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

numpy stores array in memory in a contiguous block. It uses shape() to form a view into that data. You can use resize() to alter the memory representation. (depends)

Python array ranges:
```
img[i,:] = [0,0,0,0] #set the values of row i
img[a:b] = [1,1,1,1]  # set the values of row a to b
img[:c]=[2,2,2,2]  #set the first c rows of image
img[,-1] = [255,255,255,255] # set the last column
img[100] = [4,4,4,4] $set row 100;
```

To access the first column, add : to indicate rows are complete

img[:,1]
or img[:,1:3] for columns 1-3.

Fortran format vs. C format.

For an array [i,j,k] there are different storage conventions for arrays. Fortran and C use different standards/conventions. Fortran format iterates i the fastest and is a column oriented format. For multidimensional arrays, Fortran allocates the first column first followed by the second column. To maximize cache hits in Fortran format arrays, increment the first array index first progressing towards the end. C format stores rows of the array first. C format iterates k the fastest to minimize cache misses.

int arr[i,j,k]

```
int numValues=0;
for (indexi=0; indexi < 10; indexi++)
  for(indexj=0; indexj < 10; indexj++)
    for(indexk=0; indexk<10; indexk++)
      arr[i,j,jk] = numValues;
      numValues++;
```

Writing sequential values in C; we index the last index the fastest for sequential cache reads/ writes.

Contrary to most documentation, numpy arrays support both Fortran and C ordering. Most docs indicate numpy arrays are stored in Fortran order only.

```
>>> f=np.array([[0,1,2], [3,4,5], [6,7,8],[9,10,11]], order='F')
>>> c=np.array([[0,1,2], [3,4,5], [6,7,8],[9,10,11]], order='C')
```

Be careful of the reshaping. If you reshape a Fortran formatted array without the order='F' option then numpy will reshape using C format.

Correct behavior:

```
>>> f.reshape(3,4, order='F')
array([[ 0,  9,  7,  5],
       [ 3,  1, 10,  8],
       [ 6,  4,  2, 11]])
>>> c.reshape(3,4,order='C')
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Below is bad:

```
>>> f.reshape(3,4)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
```

Axis

A numpy axis is defined as removing the dimension of the axis

```
>>> y.shape
(3, 32, 32)
```

```
>>> y.shape
(3, 32, 32)
>>> y.sum(axis=0).shape
(32, 32)
>>> y.sum(axis=1).shape
(3, 32)
>>> y.sum(axis=2).shape
(3, 32)
```

Transpose

2 ways to transpose

1) a=arange(20), a.shape(10,2), np.transpose(a)
2) a=arange(20); a.shape(10,2), b=a.T

A transpose changes the strides for the axes reordering and reshaping the elements or creating a separate view.

numpy transposes change the stride and the shape.

```
>>> y=np.array([[[1,2,3],[4,5,6]]])
>>> y.shape
(1, 2, 3)
>>> y.strides
(48, 24, 8)
>>> y.shape
(1, 2, 3)
>>> np.transpose(y,(0,2,1))
array([[[1, 4],
        [2, 5],
        [3, 6]]])
>>> np.transpose(y,(2,1,0))
```

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

       [[2],
        [5]],

       [[3],
        [6]]])
>>> np.transpose(y,(2,1,0)).strides
(8, 24, 48)
>>> np.transpose(y,(2,1,0)).shape
(3, 2, 1)
```

Transpose when you get empty rows!!! Because there is another array starting. An array boundary is a blank line. Doesn't affect array elements. Note a transpose of 0,1,2 to 1,0,2 doesn't do anything because there is no i index.

```
>> y=np.array([[[1,2,3],[4,5,6]]])
>>> y
array([[[1, 2, 3],
        [4, 5, 6]]])
>>> np.transpose(y,(1,0,2))
array([[[1, 2, 3]],

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

We can test this by adding more array braces around a single element. As we add more we can see on a print more empty lines are added.

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

       [[2]]])
>>> x=np.array([[[[1,2]]]])
>>> y=np.transpose(x)
>>> y
array([[[[1]]],


       [[[2]]]])
>>>
```

For a 2 dimensional array, a transpose can be implemented by swapping the i and j indexes. For a N-dim array, the default implementation of a transpose is:

Resizing vs Reshape;
a resize reallocates data, resizing to a smaller dimension deletes data whereas a reshape keeps the original data/buffer, can also add data by increasing the buffer.

As an exercise let's resize a C format array to a F format array. This is important because we have this code in the CIFAR dataset.

```
>>> a.reshape(10,3,32,32).transpose(0,2,3,1)
array([[[[   0,  1024,  2048],
         [   1,  1025,  2049],
         [   2,  1026,  2050],
         ...,
         [  29,  1053,  2077],
         [  30,  1054,  2078],
         [  31,  1055,  2079]],

        [[  32,  1056,  2080],
         [  33,  1057,  2081],
         [  34,  1058,  2082],
         ...,
         [  61,  1085,  2109],
         [  62,  1086,  2110],
         [  63,  1087,  2111]],

        [[  64,  1088,  2112],
         [  65,  1089,  2113],
         [  66,  1090,  2114],
         ...,
         [  93,  1117,  2141],
         [  94,  1118,  2142],
         [  95,  1119,  2143]],


        ...,
        [[ 928,  1952,  2976],
         [ 929,  1953,  2977],
         [ 930,  1954,  2978],
         ...,
         [ 957,  1981,  3005],
         [ 958,  1982,  3006],
         [ 959,  1983,  3007]],
```

```
 [[  960,  1984,  3008],
  [  961,  1985,  3009],
  [  962,  1986,  3010],
  ...,
  [  989,  2013,  3037],
  [  990,  2014,  3038],
  [  991,  2015,  3039]],

 [[  992,  2016,  3040],
  [  993,  2017,  3041],
  [  994,  2018,  3042],
  ...,
  [ 1021,  2045,  3069],
  [ 1022,  2046,  3070],
  [ 1023,  2047,  3071]]],


[[[ 3072,  4096,  5120],
  [ 3073,  4097,  5121],
  [ 3074,  4098,  5122],
  ...,
  [ 3101,  4125,  5149],
  [ 3102,  4126,  5150],
  [ 3103,  4127,  5151]],

 [[ 3104,  4128,  5152],
  [ 3105,  4129,  5153],
  [ 3106,  4130,  5154],
  ...,
  [ 3133,  4157,  5181],
  [ 3134,  4158,  5182],
  [ 3135,  4159,  5183]],

 [[ 3136,  4160,  5184],
  [ 3137,  4161,  5185],
  [ 3138,  4162,  5186],
  ...,
  [ 3165,  4189,  5213],
  [ 3166,  4190,  5214],
  [ 3167,  4191,  5215]],

 ...,
 [[ 4000,  5024,  6048],
  [ 4001,  5025,  6049],
  [ 4002,  5026,  6050],
  ...,
  [ 4029,  5053,  6077],
  [ 4030,  5054,  6078],
```

```
 [ 4031,  5055,  6079]],

[[ 4032,  5056,  6080],
 [ 4033,  5057,  6081],
 [ 4034,  5058,  6082],
 ...,
 [ 4061,  5085,  6109],
 [ 4062,  5086,  6110],
 [ 4063,  5087,  6111]],

[[ 4064,  5088,  6112],
 [ 4065,  5089,  6113],
 [ 4066,  5090,  6114],
 ...,
 [ 4093,  5117,  6141],
 [ 4094,  5118,  6142],
 [ 4095,  5119,  6143]]],


[[[ 6144,  7168,  8192],
 [ 6145,  7169,  8193],
 [ 6146,  7170,  8194],
 ...,
 [ 6173,  7197,  8221],
 [ 6174,  7198,  8222],
 [ 6175,  7199,  8223]],

[[ 6176,  7200,  8224],
 [ 6177,  7201,  8225],
 [ 6178,  7202,  8226],
 ...,
 [ 6205,  7229,  8253],
 [ 6206,  7230,  8254],
 [ 6207,  7231,  8255]],

[[ 6208,  7232,  8256],
 [ 6209,  7233,  8257],
 [ 6210,  7234,  8258],
 ...,
 [ 6237,  7261,  8285],
 [ 6238,  7262,  8286],
 [ 6239,  7263,  8287]],

...,
[[ 7072,  8096,  9120],
 [ 7073,  8097,  9121],
 [ 7074,  8098,  9122],
 ...,
 [ 7101,  8125,  9149],
```

```
  [ 7102,  8126,  9150],
  [ 7103,  8127,  9151]],

 [[ 7104,  8128,  9152],
  [ 7105,  8129,  9153],
  [ 7106,  8130,  9154],
  ...,
  [ 7133,  8157,  9181],
  [ 7134,  8158,  9182],
  [ 7135,  8159,  9183]],

 [[ 7136,  8160,  9184],
  [ 7137,  8161,  9185],
  [ 7138,  8162,  9186],
  ...,
  [ 7165,  8189,  9213],
  [ 7166,  8190,  9214],
  [ 7167,  8191,  9215]]],


...,
[[[21504, 22528, 23552],
  [21505, 22529, 23553],
  [21506, 22530, 23554],
  ...,
  [21533, 22557, 23581],
  [21534, 22558, 23582],
  [21535, 22559, 23583]],

 [[21536, 22560, 23584],
  [21537, 22561, 23585],
  [21538, 22562, 23586],
  ...,
  [21565, 22589, 23613],
  [21566, 22590, 23614],
  [21567, 22591, 23615]],

 [[21568, 22592, 23616],
  [21569, 22593, 23617],
  [21570, 22594, 23618],
  ...,
  [21597, 22621, 23645],
  [21598, 22622, 23646],
  [21599, 22623, 23647]],

 ...,
 [[22432, 23456, 24480],
  [22433, 23457, 24481],
  [22434, 23458, 24482],
```

```
  ...,
  [22461, 23485, 24509],
  [22462, 23486, 24510],
  [22463, 23487, 24511]],

 [[22464, 23488, 24512],
  [22465, 23489, 24513],
  [22466, 23490, 24514],
  ...,
  [22493, 23517, 24541],
  [22494, 23518, 24542],
  [22495, 23519, 24543]],

 [[22496, 23520, 24544],
  [22497, 23521, 24545],
  [22498, 23522, 24546],
  ...,
  [22525, 23549, 24573],
  [22526, 23550, 24574],
  [22527, 23551, 24575]]],


[[[24576, 25600, 26624],
  [24577, 25601, 26625],
  [24578, 25602, 26626],
  ...,
  [24605, 25629, 26653],
  [24606, 25630, 26654],
  [24607, 25631, 26655]],

 [[24608, 25632, 26656],
  [24609, 25633, 26657],
  [24610, 25634, 26658],
  ...,
  [24637, 25661, 26685],
  [24638, 25662, 26686],
  [24639, 25663, 26687]],

 [[24640, 25664, 26688],
  [24641, 25665, 26689],
  [24642, 25666, 26690],
  ...,
  [24669, 25693, 26717],
  [24670, 25694, 26718],
  [24671, 25695, 26719]],

 ...,
 [[25504, 26528, 27552],
  [25505, 26529, 27553],
```

```
  [25506, 26530, 27554],
  ...,
  [25533, 26557, 27581],
  [25534, 26558, 27582],
  [25535, 26559, 27583]],

 [[25536, 26560, 27584],
  [25537, 26561, 27585],
  [25538, 26562, 27586],
  ...,
  [25565, 26589, 27613],
  [25566, 26590, 27614],
  [25567, 26591, 27615]],

 [[25568, 26592, 27616],
  [25569, 26593, 27617],
  [25570, 26594, 27618],
  ...,
  [25597, 26621, 27645],
  [25598, 26622, 27646],
  [25599, 26623, 27647]]],


[[[27648, 28672, 29696],
  [27649, 28673, 29697],
  [27650, 28674, 29698],
  ...,
  [27677, 28701, 29725],
  [27678, 28702, 29726],
  [27679, 28703, 29727]],

 [[27680, 28704, 29728],
  [27681, 28705, 29729],
  [27682, 28706, 29730],
  ...,
  [27709, 28733, 29757],
  [27710, 28734, 29758],
  [27711, 28735, 29759]],

 [[27712, 28736, 29760],
  [27713, 28737, 29761],
  [27714, 28738, 29762],
  ...,
  [27741, 28765, 29789],
  [27742, 28766, 29790],
  [27743, 28767, 29791]],

 ...,
 [[28576, 29600, 30624],
```

```
         [28577, 29601, 30625],
         [28578, 29602, 30626],
         ...,
         [28605, 29629, 30653],
         [28606, 29630, 30654],
         [28607, 29631, 30655]],

        [[28608, 29632, 30656],
         [28609, 29633, 30657],
         [28610, 29634, 30658],
         ...,
         [28637, 29661, 30685],
         [28638, 29662, 30686],
         [28639, 29663, 30687]],

        [[28640, 29664, 30688],
         [28641, 29665, 30689],
         [28642, 29666, 30690],
         ...,
         [28669, 29693, 30717],
         [28670, 29694, 30718],
         [28671, 29695, 30719]]])
>>> a.reshape(10,3,32,32).transpose(0,2,3,1) .shape
(10, 32, 32, 3)
>>> a.reshape(10,3,32,32).shape
(10, 3, 32, 32)
>>> a=np.arange(3072)
>>> a.reshape(1,3,32,32).transpose(0,2,3,1) .shape
(1, 32, 32, 3)
>>> a.reshape(1,3,32,32).transpose(0,2,3,1)
array([[[[   0, 1024, 2048],
         [   1, 1025, 2049],
         [   2, 1026, 2050],
         ...,
         [  29, 1053, 2077],
         [  30, 1054, 2078],
         [  31, 1055, 2079]],

        [[  32, 1056, 2080],
         [  33, 1057, 2081],
         [  34, 1058, 2082],
         ...,
         [  61, 1085, 2109],
         [  62, 1086, 2110],
         [  63, 1087, 2111]],

        [[  64, 1088, 2112],
         [  65, 1089, 2113],
         [  66, 1090, 2114],
```

```
        ...,
        [  93, 1117, 2141],
        [  94, 1118, 2142],
        [  95, 1119, 2143]],

       ...,
       [[ 928, 1952, 2976],
        [ 929, 1953, 2977],
        [ 930, 1954, 2978],
        ...,
        [ 957, 1981, 3005],
        [ 958, 1982, 3006],
        [ 959, 1983, 3007]],

       [[ 960, 1984, 3008],
        [ 961, 1985, 3009],
        [ 962, 1986, 3010],
        ...,
        [ 989, 2013, 3037],
        [ 990, 2014, 3038],
        [ 991, 2015, 3039]],

       [[ 992, 2016, 3040],
        [ 993, 2017, 3041],
        [ 994, 2018, 3042],
        ...,
        [1021, 2045, 3069],
        [1022, 2046, 3070],
        [1023, 2047, 3071]]])
>>>
```

Some Fun exercises:
numpy includes graphing for histograms, create a color histogram for an image

Matlab is obsolete. Do your matlab work in Keras or some GPU enabled framework for matrices and anything related to matricies.

Keras Review:
1) add input as a layer
2) add learning layer via compile
3) add training via fit

```
model=Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Anaconda review:
Install Anaconda and then cd into assignment1 and run ipython notebook.
conda run ipython notebook starts the Jupyter server, select the notebook from the webpage

The start_ipython_osx script si fro the virtualenv setup option. You do not need to use this if using Anaconda.

Assignment 1 Review

Assignment2 Review

Assignment 3 Review: