March 9th: Class 6 Numpy and Recursion

I. Introduction of `numpy` and `matplotlib`

A. Numpy

Numpy is an external package in Python for making multidimensional arrays (very similar to Matlab)

- Why do we use it? **Speed!**
- To learn more about numpy: http://www.numpy.org/
- If you've downloaded Anaconda, you already have the numpy package downloaded.
- Two rules in Numpy help accomplish this speed
    1. Homogeneous: all elements of an array are of same type
    2. Fixed length

1) First import numpy: `import numpy`

2) 6 ways to create an array in `numpy`

1. function `array(var)`: can put any one sequential type variable (list, tuple, set) in the parentheses to cast it into numpy array
2. function `zeros(x)`: return an array of length x (all elements 0)
   Use a tuple as an argument instead of int to create multidimensional arrays

e.g.

```
In: import numpy
In: s = numpy.zeros(10)
In: s
Out: [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]

In: s = numpy.zeros((3,2))
In: s
Out: array([[ 0.,  0.],
            [ 0.,  0.],
            [ 0.,  0.]])
```

3. function `ones(x)` : same as above with 1.0s instead of 0.s
4. function `arange()` : calling the array function on `range()`

```
In: import numpy as np
In: np.arange(3)
Out: array([0, 1, 2])
```

5. function `linspace(starting point, ending point, number of points I want in between)`: carves up the range into n-1 pieces for you

```
In: np.linspace(1,10,2)
Out: array([  1.,  10.])
In: np.linspace(1,10,5)
Out: array([  1.  ,   3.25,   5.5 ,   7.75,  10.  ])
```

6. function `random.rand(n)`: array of n random numbers from 0 to 1

```
In: np.random.rand(3)
Out: array([ 0.63040386,  0.27029223,  0.46125486])

In: np.random.rand(3,3)
Out: [[ 0.61925984  0.97407503  0.02655378]
      [ 0.51168271  0.46085259  0.27524427]
      [ 0.77434942  0.64746699  0.49072881]]
```
(2D array of random#)

3) Some constants for numpy arrays

1. `dtype`: returns datatype of the array elements
2. `shape`: returns a tuple that shows length in each dimension
3. `ndim`: returns dimension
4. `size`: returns number of elements

e.g.

```
a = np.zeros((3,4))
In: a.shape
Out:(3,4)
In: a.ndim
Out: 2
In: a.size
Out: 12
In: a.dtype
Out: dtype('int64')
```

e.g.

```
In: d1 = np.zeros(3)
In: d1.shape
Out: (3,)  # because one dimensional

In: d2 = np.zeros((1,3))
In: d2.shape
Out: (1,3) # now 2D although d1 and d2 look same to us
```

4) Indexing through numpy arrays and assigning elements

```
In: import numpy as np
In: s3 = np.zeros((2,2,2))  #3D array:2 floors of 2row*2column matrix
In: s3[0][0][0]
Out: 0.

In: s3[0,0,0]  #different syntax for same meaning as above (both work)
Out: 0.

In: s3[1,:,:] #colon means everything in that dimension
Out: array([[[ 0.,  0.],
             [ 0.,  0.]]])

In: s3[1,1,1] = 100 #assigning elements → this modifies array
In: s3
Out: array([[[   0.,    0.],
             [   0.,    0.]],

            [[   0.,    0.],
             [   0.,  100.]]])

In: s3[0,:,:] = 1 #slicing works(set everything in first floor to 1)
Out: array([[[   1.,    1.],
             [   1.,    1.]],
```

```
           [[   0.,     0.],
            [   0.,   100.]]])
```

5) Methods

- `.astype(`*`datatype`*`)`: returns a brand new numpy array with elements casted

```
In: s3.astype(int)
Out: array([[[  1,    1],
             [  1,    1]],

            [[  0,    0],
             [  0, 100]]])
```

- `.reshape(`*`tuple for size`*`)`: returns new, reformatted version of original
  ** should keep size (number of elements) the same like below

```
In: s3.reshape((2,1,4))
Out: array([[[   1.,     1.,     1.,     1.]],

            [[   0.,     0.,     0.,   100.]]])
In: s3
Out: array([[[   1.,     1.],
             [   1.,     1.]],

            [[   0.,     0.],
             [   0.,   100.]]])   # s3 unmodified
```

- `.resize()`: same as `reshape` but modifies the original array instead of creating new one

```
In: s3.resize((2,1,4))
In: s3
Out: array([[[   1.,     1.,     1.,     1.]],

            [[   0.,     0.,     0.,   100.]]])
```

- perform arithmetic operations on all elements

```
In: s3 = s3/2
In: s3
Out: array([[[  0.5,    0.5,    0.5,    0.5]],
             [[  0. ,    0. ,    0. ,   50. ]]])

In: s3 = s3 > .5
In: s3
Out: array([[[False, False, False, False]],

             [[False, False, False,  True]]], dtype=bool)

In: s3 = s3.astype(int) #cast the Boolean matrix as int
In: s3
Out: array([[[0, 0, 0, 0]],
             [[0, 0, 0, 1]]])
```

B. MATPLOTLIB

As always, import!

```
import matplotlib.pyplot as plt
```

1) Visual grid

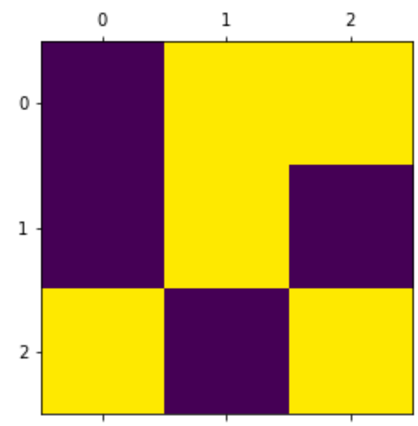
`plt.matshow(`*`numpy_array`*`)`**:** function to generate visual display from array

`plt.show()`**:** show visual display

e.g.

```
import numpy as np
import matplotlib.pyplot as plt
a = np.random.rand(3,3)
a = a<.7 #each element is True with .7 chance, False with .3 chance
a = a.astype(int) #now 1 with .7 chance, 0 with .3 chance
print a
plt.matshow(a)
plt.show()
```

```
→ [[0 1 1]
   [0 1 0]
   [1 0 1]]
```



2) Plot on Graph

```
var1 = np.linspace(start,stop,#stops) #determine range and inverval of x
```

1) Generate 1 plot

```
plt.plot(var1,var2) # line plot of var2 vs var1
plt.plot(var2,var1,'o') # dot plot
```

2) Generate nice graph with title, axis title

```
fig = plt.figure() # generate figure
gr1 = fig.add_subplot(111)
            #Among 1x1 graphs of fig, gr1 is the 1st graph(see below for detail)
gr1.plot(var1, var2)  # Plot var2 vs var1 in sutplot gr1
gr1.set_title('string')
gr1.set_xlabel('string')
gr1.set_ylabel('Percolation probability')
```

3) Generate various subplots on one display

```
fig = plt.figure() # generate figure
```

```
gr1 = fig.add_subplot(221)
      #Among 2x2 subplots of fig, gr1 is the 1st graph(top left)
gr2 = fig.add_subplot(222) # gr2 is the 2nd graph (top right)
gr3 = fig.add_subplot(223) # gr2 is 3rd graph (bottom left)
gr4 = fig.add_subplot(224) # gr2 is 4th graph (bottom right)
gr1.plot(var1, var2)
gr2.plot(var2, var3), etc. # must plot and title each subplot separately
```

e.g.

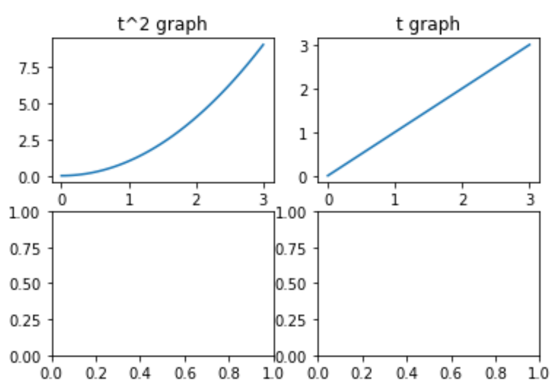
```
def f(t):
    return t**2

t = np.linspace(0,3,50)
y = np.zeros(len(t))
for i in range(len(t)):
    y[i] = f(t[i])
fig = plt.figure()
gr1 = fig.add_subplot(221)
gr1.plot(t,y)
gr1.set_title('t^2 graph')

gr2 = fig.add_subplot(222)
gr2.plot(t,t)
gr2.set_title('t graph')

gr3 = fig.add_subplot(223)
gr4 = fig.add_subplot(224) #gr3, gr4 are empty
plt.show()
```

Result:



II. Recursion

- A function that calls itself
- RULES OF RECURSION!
  - Must have a simple base case (always write this first when coding recursive functions)
  - Subproblems should be similar to big problem, but smaller
  - Operations should not overlap

- Example 1> reversing a string

```
def reverse(s):
    if len(s) == 1:
        return s        # base case
    else:
        new_string = s[len(s)-1] + reverse(s[0:len(s)-1])
        return new_string
```