Lab 1 - Basic iPython Tutorial (EE 126 Fall 2014)

modified from Berkeley Python Bootcamp 2013 https://github.com/profjsb/python-bootcamp

and Python for Signal Processing http://link.springer.com/book/10.1007%2F978-3-319-01342-8

and EE 123 iPython Tutorial http://inst.eecs.berkeley.edu/~ee123/sp14/lab/python_tutorial.ipynb

- Rishi Sharma

General iPython Notebook usage instructions (overview)

- Click the `Play` button to run and advance a cell. The short-cut for it is `shift-enter`
- To add a new cell, either select `"Insert->Insert New Cell Below"` or click the white down arrow button
- You can change the cell mode from code to text in the pulldown menu. I use `Markdown` for text
- You can change the texts in the `Markdown` cells by double-clicking them.
- To save your notebook, either select `"File->Save and Checkpoint"` or hit `Command-s` for Mac and `Ctrl-s` for Windows
- To undo in each cell, hit `Command-z` for Mac and `Ctrl-z` for Windows
- To undo `Delete Cell`, select `Edit->Undo Delete Cell`
- `Help->Keyboard Shortcuts` has a list of keyboard shortcuts

Installing Python

Follow the instructions on the class website to install python (if you're reading this, you've probably already done that):

http://ipython.org/install.html

Make sure you install the notebook dependencies for iPython in addition to the basic package

Tab Completion

One useful feature of iPython is tab completion

In [ ]:
```
x = 1
y = 2
x_plus_y = x+y

# type `x_` then hit TAB will auto-complete the variable
# then press shift + enter to run the cell
print x_
```

Help

Another useful feature is the help command. Type any function followed by `?` and run the cell returns a help window. Hit the `x` button to close it.

In [ ]:
```
abs?
```

Hit `Tab` after a left parenthesis also brings up a help window. Press Tab twice to see a more detailed Help window. Press Tab four times to have the same help window pop up that does if you use `?`. Some people may need to use `Shift+Tab` rather than just `Tab`. You can also just wait a few seconds after typing a left parenthesis and a help window will appear.

In [ ]:
```
abs(
```

Floats and Integers

Doing math in python is easy, but note that there are `int` and `float` in Python. Integer division returns the floor. Always check this when debugging!

In [ ]:
```
59 / 87
```

In [ ]:
```
59 / 87.0
```

However, this will change in the future (Python 3.0)... If you import division from the future, then everything works fine

In [ ]:
```
from __future__ import division
59 / 87
```

Strings

Double quotes and single quotes are the same thing. `'+'` concatenates strings

In [ ]:
```
# This is a comment
"Hi " + 'Bye'
```

Lists

A list is a mutable array of data, ie can change stuff. They can be created using square brackets [ ]

Important functions:

- `'+'` appends lists.
- `len(x)` to get length

In [ ]:
```
x = [1, 2, "asdf"] + [4, 5, 6]

print x
```

In [ ]:
```
print len(x)
```

Tuples

A tuple is an unmutable list. They can be created using round brackets ().

They are usually used as inputs and outputs to functions

In [ ]:
```
t = (1, 2, "asdf") + (3, 4, 5)
print t
```

In [ ]:
```
# cannot do assignment
t[0] = 10

# errors in ipython notebook appear inline
```

Arrays (Numpy)

Numpy array is like a list with multidimensional support and more functions. We will be using it a lot.

Arithmetic operations on numpy arrays correspond to elementwise operations.

Important functions:

- `.shape` returns the dimensions of the array
- `.ndim` returns the number of dimensions.
- `.size` returns the number of entries in the array
- `len()` returns the first dimension

To use functions in numpy, we have to import numpy to our workspace. This is done by the command `import numpy`. By convention, we rename `numpy` as `np` for convenience

In [ ]:
```
import numpy as np # by convention, import numpy as np

x = np.array( [ [1, 2, 3], [4, 5, 6] ] )

print x
```

In [ ]:
```
print "Number of dimensions:", x.ndim
```

In [ ]:
```
print "Dimensions:", x.shape
```

In [ ]:
```
print "Size:", x.size
```

In [ ]:
```
print "Length:", len(x)
```

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

```
print "a=", a

print "a*a=", a * a   #elementwise
```

In []:
```
b = np.array(np.ones((3,3)))*2
print "b=", b
c = np.array(np.ones((3,3)))
print "c=", c
```

multiply elementwise

In []:
```
print "b*c = ", b*c
```

Now multiply as matrices

In []:
```
print "b*c = ", np.matrix(b)*np.matrix(c)
print "b*c = ", np.dot(b,c)
```

Alternatively, array can be created as a matrix

In []:
```
d = np.matrix([[1,1j,0],[1,2,3]])
e = np.matrix([[1],[1j],[0]])

print d*e
```

Slicing for numpy arrays

Numpy uses pass-by-reference semantics so it creates views into the existing array, without implicit copying. This is particularly helpful with very large arrays because copying can be slow.

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

We slice an array from a to b-1 with `[a:b]`

In []:
```
y = x[0:4]
print y
```

Since slicing does not copy the array, changing `y` changes `x`

In []:
```
y[0] = 7
print x
print y
```

To actually copy x, we should use .copy()

In []:
```
x = np.array([1,2,3,4,5,6])
```

```
y = x.copy()
y[0] = 7
print x
print y
```

Commonly used Numpy functions: r_ and c_

We use `r_` to create integer sequences

`r_[0:N]` creates an array listing every integer from 0 to N-1

`r_[0:N:m]` creates an array listing every `m` th integer from 0 to N-1

In [ ]:
```
from numpy import r_ # import r_ function from numpy directly, so that we can call r_ directly instead of np.r_

print r_[-5:5] # every integer from -5 ... 4

print r_[0:5:2] # every other integer from 0 ... 4

print abs( r_[-5:5] )
```

r_ stands for row concatenation, and the function r_[-5:5] is saying to row concatenate every element generated in the range [-5:5]. This is just one use case for row concatenation, but as you can imagine there are many others. The same goes for its cousin the c_ function, which performs a column concatenation.

In [ ]:
```
from numpy import r_
from numpy import c_

row1 = [[1,0,0]]
row2 = [[0,1,0]]
row3 = [[0,0,1]]

# we want to stack these three rows to create a 3x3 identity matrix
# this is where the r_ function comes in handy

print np.r_[row1,row2,row3] # 3x3 identity matrix appending vectors as rows
```

What would have happened above if you had used `c_` instead of `r_` on `row1`, `row2`, and `row3`? You should try it in the box below

In [ ]:
```
print np.c_[# fill in # ]     # vector created by appending elements as new columns
```

Some more examples:

In [ ]:
```
X = np.eye(3) # 3x3 Identity Matrix
Y = np.ones([3,3]) # 3x3 Matrix of ones
print "X = "
print X
print "Y = "
print Y

Z = r_[X,Y] # concatenate y to x as rows
print "\n Row Concatenated [X ; Y] : "
print Z

W = c_[X,Y] # concatenate y to x as columns
print "\n Column Concatenated [X Y] : \n "
print W
```

Plotting

In this class, we will use `matplotlib.pyplot` to plot signals and images.

To begin with, we import `matplotlib.pyplot` as `plt`

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt # by convention, we import pyplot as plt
from numpy import r_ # import r_ function from numpy



x = r_[:1:0.01] # if you don't specify a number before the colon, the starting
 index defaults to 0
a = np.exp( -x )
b = np.sin( x*10.0 )/4.0 + 0.5

# plot in browser instead of opening new windows
%matplotlib inline
```

`plt.plot(x, a)` plots `a` against `x`

In [ ]:
```
plt.figure()
plt.plot( x, a )
```

Once you started a figure, you can keep plotting to the same figure

In [ ]:
```
plt.figure()
plt.plot( x, a )
plt.plot( x, b )
```

To plot different plots, you can create a second figure

In [ ]:
```
plt.figure()
```

In [ ]:
```
plt.figure()
plt.plot( x, a )
plt.figure()
plt.plot( x, b )
```

To label the axes, use `plt.xlabel()` and `plt.ylabel()`

In [ ]:
```
plt.figure()
plt.plot( x, a )
plt.plot( x, b )

plt.xlabel( "time" )
plt.ylabel( "space" )
```

You can also add title and legends using `plt.title()` and `plt.legend()`

In [ ]:
```
plt.figure()
plt.plot( x, a )
plt.plot( x, b )
plt.xlabel( "time" )
plt.ylabel( "space" )

plt.title( "Most important graph in the world" )

plt.legend( ("blue", "red") )
```

There are many options you can specify in `plot()`, such as color and linewidth. You can also change the axis using `plt.axis`

In [ ]:
```
plt.figure()
plt.plot( x, a ,':r',linewidth=20)
plt.plot( x, b ,'--k')
plt.xlabel( "time" )
plt.ylabel( "space" )

plt.title( "Most important graph in the world" )

plt.legend( ("blue", "red") )



plt.axis( [0, 4, -2, 3] )
```

There are many other plotting functions. For example, we will use `plt.imshow()` for showing images and `plt.stem()` for plotting discretized signal

In [ ]:
```
# image
plt.figure()
```

```
data = np.outer( a, b ) # plotting the outer product of a and b

plt.imshow(data)
```

In []:
```
# stem plot
plt.figure()
plt.stem(a[::5]) # subsample by 5
```

In []:
```
# xkcd style
plt.xkcd()
plt.plot( x, a )
plt.plot( x, b )
plt.xlabel( "time" )
plt.ylabel( "space" )

plt.title( "Most important graph in the world" )

plt.legend( ("blue", "red") )
```

To turn off xkcd style plotting, restart the kernel or run the command `plt.rcdefaults()`

Logic

For loop

Indent matters in python. Everything indented belongs to the loop

In []:
```
for i in [4, 6, "asdf", "jkl"]:
    print i
```

In []:
```
for i in r_[0:10]:
    print i
```

If Else statement

Same goes to If Else

In []:
```
if 1 != 0:
    print "1 != 0"
else:
    print "1 = 0"
```

Random Library

The numpy random library should be your resource for all Monte Carlo simulations which require generating instances of random variables.

The documentation for the library can be found here: http://docs.scipy.org/doc/numpy/reference/routines.random.html

The function \(\operatorname{rand}()\) can be used to generates a uniform random number in the range \([0,1)\)

In [ ]:
```
from numpy import random

print random.rand() # random number
print random.rand(5) # random vector
print random.rand(3,3) # random matrix
```

Let's see how we can use this to generate a fair coin toss (i.e. a discrete \(\mathcal{B}\left(\frac{1}{2}\right)\) random variable)

In [ ]:
```
x = round(random.rand()) # Bernoulli(1/2) random variable
print x
```

Now let's generate several fair coin tosses and plot a histogram of the results

In [ ]:
```
k = 100
x = [round(random.rand()) for _ in xrange(k)]
# we could also use numpy's round function to element-wise round the vector
x = np.round(random.rand(k))

plt.hist(x)
```

We can do something similar for several other distributions, and allow the histogram to give us a sense of what the distribution looks like. As we increase the number of samples we take from the distribution \(k\), the more and more our histogram looks like the actual distribution.

In [ ]:
```
k = 1000

discrete_uniform = random.randint(0,10,size=k) # k discrete uniform random var
iables between 0 and 9
plt.figure(figsize=(6,3))
plt.hist(discrete_uniform)
plt.title('discrete uniform')

continuous_uniform = random.rand(k)
plt.figure(figsize=(6,3))
plt.hist(continuous_uniform)
plt.title('continous uniform')
```

```
std_normal = random.randn(k) # randn generates elements from the standard norm
al
plt.figure(figsize=(6,3))
plt.hist(std_normal)
plt.title('standard normal')

# To generate a normal distribution with mean mu and standard deviation sigma,
 we must mean shift and scale the variable
mu = 100
sigma = 40
normal_mu_sigma = mu + random.randn(k)*sigma
plt.figure(figsize=(6,3))
plt.hist(normal_mu_sigma)
plt.title('N(' + str(mu) + ',' + str(sigma) + ')')
```

^ We could do this all day with all sorts of distributions. I think you get the point.

Specifying a Discrete Probability Distribution for Monte Carlo Sampling

The following function takes n sample from a discrete probability distribution specified by the two arrays `distribution` and `values`.

As an example, let us suppose a random variable X follows the following distribution:

$$ X = \begin{cases} 1 \ \text{w.probability 0.1} \\ 2 \ \text{w.probability 0.4} \\ 3 \ \text{w.probability 0.2} \\ 4 \ \text{w.probability 0.2} \\ 5 \ \text{w.probability 0.05} \\ 6 \ \text{w.probability 0.05} \end{cases} $$

Then we would have: `distribution` = $\begin{bmatrix} 0.1 , 0.4 , 0.2 , 0.2 , 0.05 , 0.05 \end{bmatrix}$ and `values` = $\begin{bmatrix} 1, 2, 3, 4, 5, 6 \end{bmatrix}$

In []:
```
def nSample(distribution, values, n):
    if sum(distribution) != 1:
        distribution = [distribution[i] / sum(distribution) for i in range(len
(distribution))]
    rand = [random.rand() for i in range(n)]
    rand.sort()
    samples = []
    samplePos, distPos, cdf = 0, 0, distribution[0]
    while samplePos < n:
        if rand[samplePos] < cdf:
            samplePos += 1
            samples.append(values[distPos])
        else:
            distPos += 1
            cdf += distribution[distPos]
    return samples
```

In []:
```
# collect k samples from X and plot the histogram
```

In [ ]:
```
# collect k samples from X and plot the histogram
samplesFromX = nSample([0.1, 0.4, 0.2, 0.2, 0.05, 0.05], [1, 2, 3, 4, 5, 6], k
)
plt.hist(samplesFromX)
plt.ylim((0,1000))
print "Wow, if we normalized the y-axis that would be a PMF. Incredible! I sho
uld try that."
```

Printing

Here are some fancy ways of printing:

In [ ]:
```
print "There are currently %d students enrolled in EE126.\n" % 109
```

In [ ]:
```
print "At least %d of you will probably end up dropping. That is %f percent" %
 ( 25, 25.0/109.0 * 100 )
```

\(\mathcal{Q}\)uestion 1: Monty Hall Proof and Simulation

Provide a rigorous proof of the probability of winning the prize given a strategy of switching doors vs. a strategy of staying put. Simulate the Monty Hall problem for the case when you do switch doors and when you don't. Run the simulation 100,000 times for each case and determine the simulated probability of winning given each strategy. You may use the function below as a guide (or not). Make sure your simulation results match the analytical result you expect. If you need a refresher on the Monty Hall problem, the following video may be of some help:

In [ ]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('Zr_xWfThjJ0')
```

Provide a proof that switching is a better strategy on the Monty Hall game show in this cell

You can write in latex (cell type = markdown) or you can attach inline an image of the work if you choose. Make sure the proof ends up inline **in this cell**. We highly encourage you use latex. For inline latex (e.g. \(x=5\)) use single $ around your math to render. For block equations use $$.

In [ ]:
```
# Code to simulate Monty Hall goes here
from __future__ import division

def MontyHall(switch, n):
    # switch is a boolean which tells you whether or not to switch from the or
iginal door chosen
    # n is the number of times you want to simulate the action

    # Your beautiful code here... #
```

```
print "Probability of Winning if You Switch Doors: ", MontyHall(True,100000)
print "Probability of Winning if You Don't Switch: " , MontyHall(False,100000)
```