Lab 2: Histograms, law of large numbers, simulating simple games

Please begin by running the code in the following cell to import the packages that are used in this notebook.

In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
import scipy.stats as st
print "Modules Imported!"
```

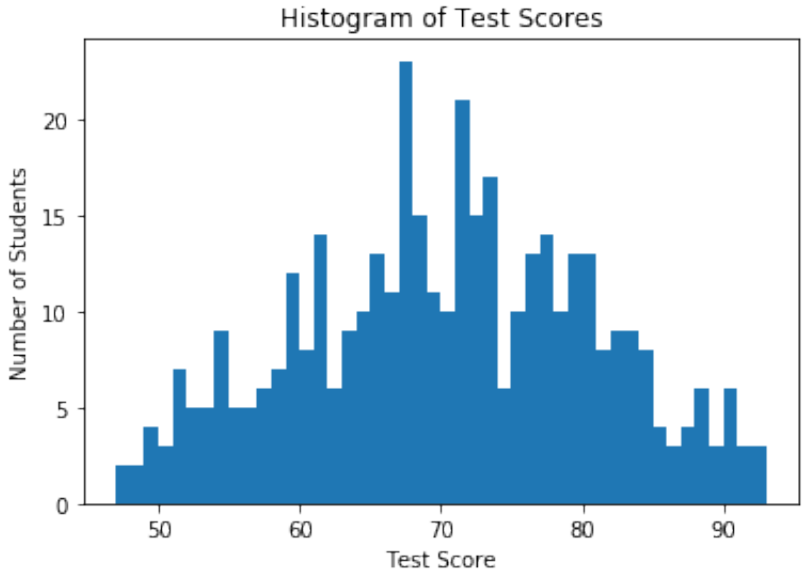
```
Modules Imported!
```

Plotting a Histogram:

A histogram is a graphical representation of a distribution. It shows the frequency at which values occur. Suppose a probability class has 400 students in it. We can simulate random tests scores (with an average of 70) and plot a histogram. The np.random.rand() function returns a number between 0 and 1. It is from a continuous uniform distribution on this interval. It means that for any c between 0 and 1, the probability the sample number is less than or equal to c is equal to c.

In [7]:
```
x = np.zeros(400) #Generates an initial array with 400 students
for i in range(400): #Loops through each of the students
    x[i] = (int)(25*(np.random.rand()-np.random.rand())+70) #Generates a random test score for that student
plt.hist(x,bins=(int)(np.max(x)-np.min(x))) #Plots a Histogram in range of all valid test scores
plt.title("Histogram of Test Scores")
plt.xlabel('Test Score')
plt.ylabel('Number of Students')
```

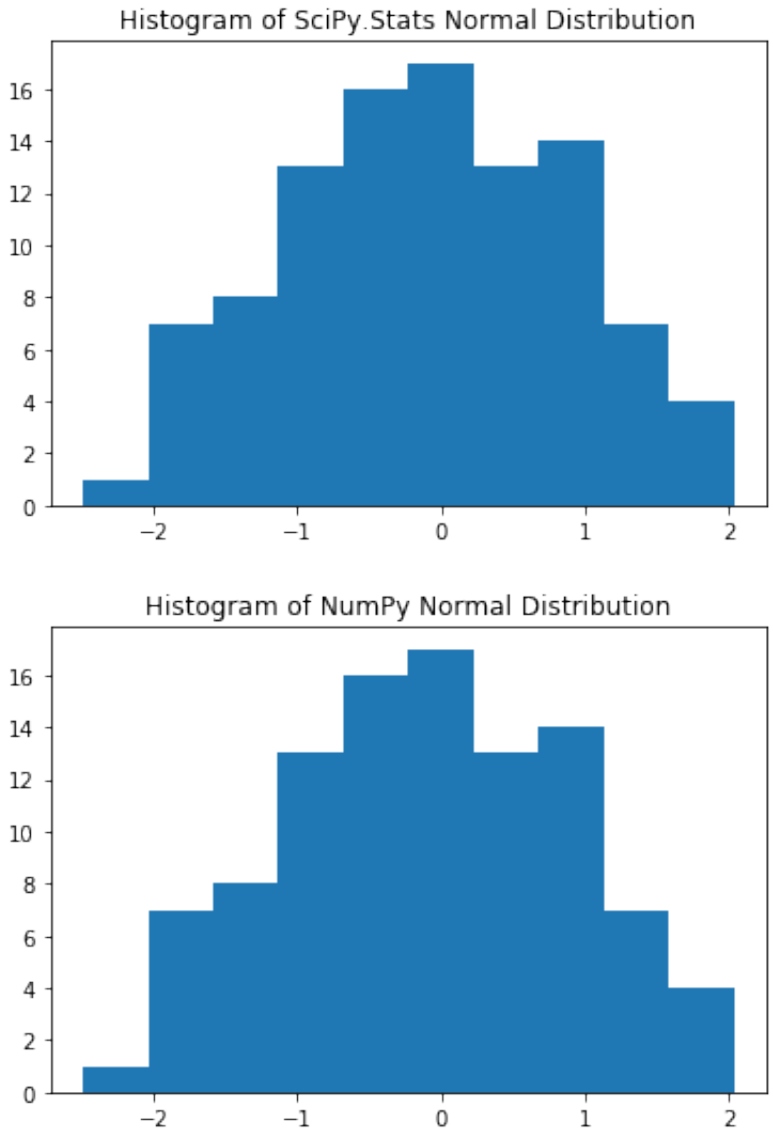
Out[7]: <matplotlib.text.Text at 0x10d2d7450>



For simulation purposes, it is useful to be able to get a value after a single trial of an RV based on a certain distribution. These are called random variates. From the stats module we've used, you simply need to call the rvs function with a size as an input. NumPy also has an easy way to do this through the random module. When you run the code below, you should see two histograms from the same normal distributions.

In [12]:
```
#Simulates a Gaussian RV 100 times in two different ways and creates a
histogram

X = st.norm()
np.random.seed(100)
x = X.rvs(size = 100) #Generates a vector with the results of 100 outp
uts or trials based on the standard Gaussian distribution
np.random.seed(100)
y = np.random.normal(size = 100) #Generates a vector with the results
of 100 outputs based on the standard Gaussian distribution
plt.hist(x); #Creates a histogram of those results, the default value
for the number of bins is bins=10
plt.title('Histogram of SciPy.Stats Normal Distribution')
plt.figure()
plt.hist(y);
plt.title('Histogram of NumPy Normal Distribution')
```

```
Out[12]: <matplotlib.text.Text at 0x10dd0e3d0>
```

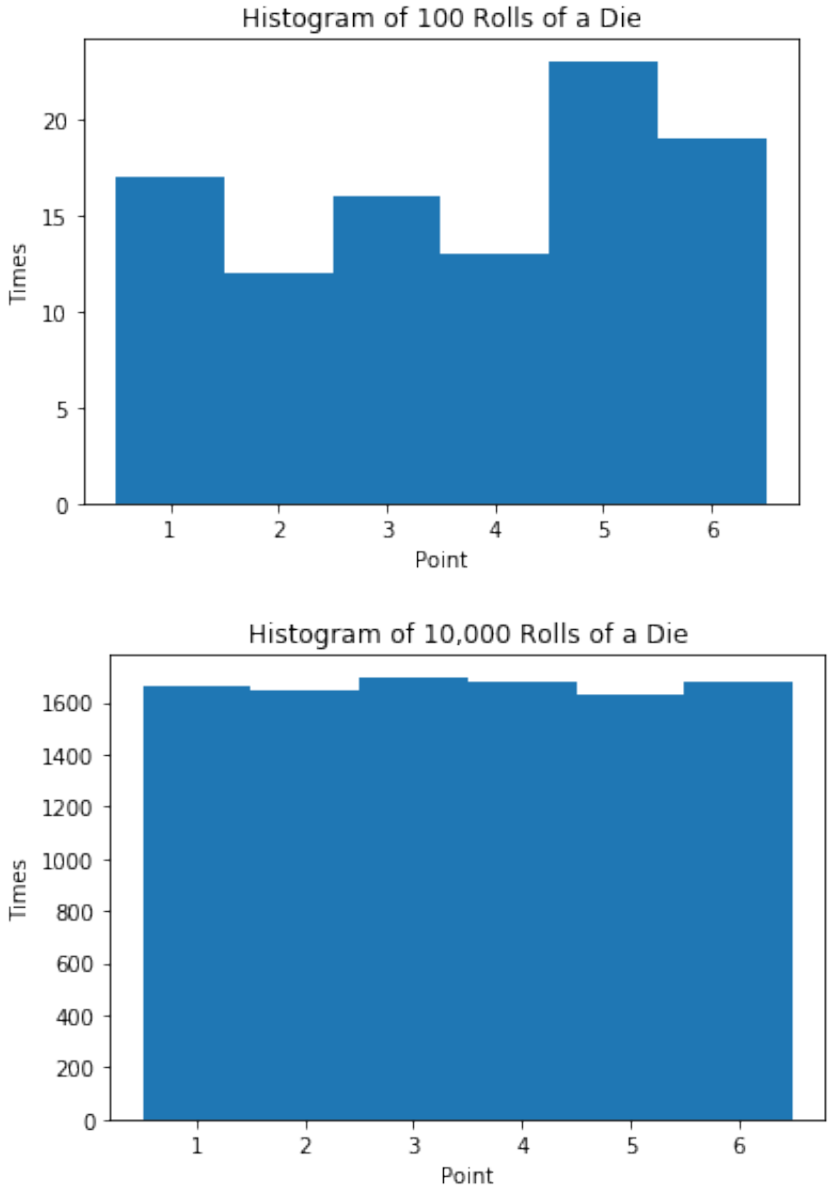


The scipy and numpy methods for getting these variates should produce the same results, but the two histograms are different. This is due to the random number generator. If we seed the random number generator before running the methods, they should produce consistent results. In the code there are two commented lines with np.random.seed(100). Uncomment these, and run the code again. You should see the histograms are the same now. Essentially, seeding the random number generator, tells it where to start in it's sequence.

Problem 1: Using the custom discrete variable you made for a roll of the die in Lab 1, or some other method (include a text cell or a comment in the code to explain your method): (a) Simulate 100 rolls using the .rvs() method in the stats module. Plot a histogram of the results. Is it what you would expect? (b) Now simulate 10,000 rolls of the die and again plot a histogram. Observe any difference and explain your observation.

In [92]:
```
########Student Answer##############
p = (1./6,1./6,1./6,1./6,1./6,1./6)
c = (1,2,3,4,5,6)
Xdie = st.rv_discrete(values=(c,p)) #Simulate of roll a die
x = Xdie.rvs(size = 100) #Roll 100 times
plt.hist(x, bins=(range(1,8)), align='left') #Plots the histogram
plt.title("Histogram of 100 Rolls of a Die")
plt.xlabel('Point')
plt.ylabel('Times')
plt.figure()
y = Xdie.rvs(size = 10000) #Roll 10,000 times
plt.hist(y, bins=(range(1,8)), align='left') #Plots the histogram
plt.title("Histogram of 10,000 Rolls of a Die")
plt.xlabel('Point')
plt.ylabel('Times')
####################################
```

```
Out[92]: <matplotlib.text.Text at 0x114068610>
```



Since we have equal probability of getting each points on a die, we should get equal number of times of getting a point.

When we roll 100 times, each point should get $\frac{100}{6} = 16.67$ times. According to the graph, there's some variation.

When we roll 10,000 times, each point should get $\frac{10000}{6} = 1666.67$ times. Compare with the previous graph, each point is closer to the theoretical value. The graph is more flat.

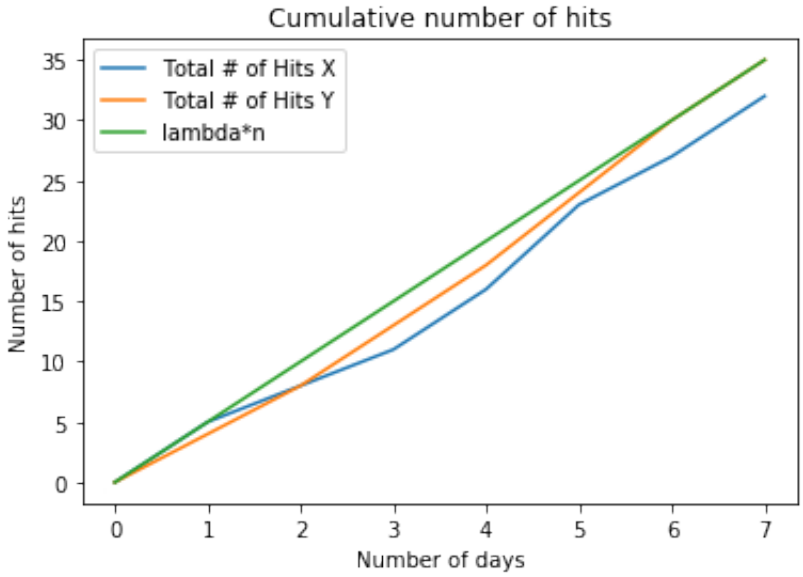
The Law of Large Numbers:

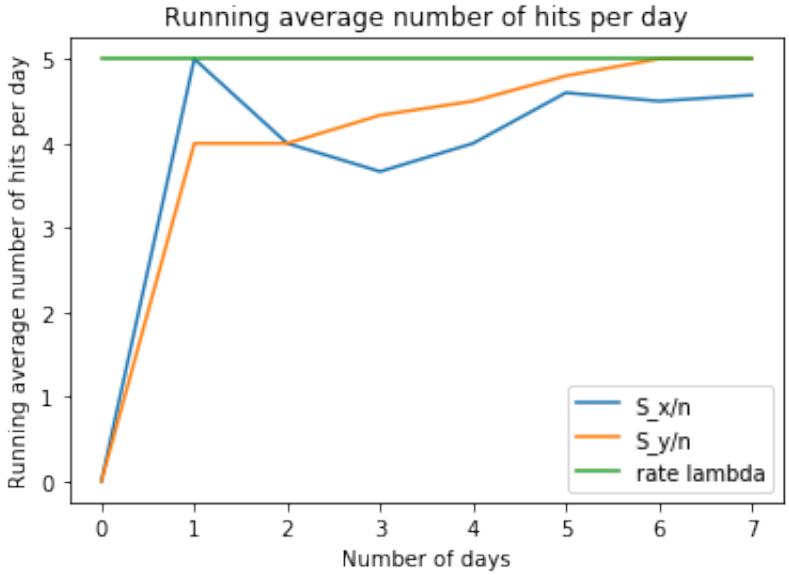
The law of large numbers describes the result of performing the same experiment a large number of times. Let $X_1, X_2, \ldots, X_n$ be a set of uncorrelated random variables, each with a finite mean of μ. Let S_n represent the sum of these random variables: $S_n = X_1 + X_2 + \ldots + X_n$. The law of large numbers states that $P\left\{\left|\frac{S_n}{n} - \mu\right| \geq \delta\right\} \leq \frac{C}{n\delta^2} \to 0$ as $n \to \infty$. Put another way this simply states that the sum of the RVs divided by n converges in some sense to the mean ($\frac{S_n}{n} \to \mu$). A proof of this is given in the ECE 313 textbook, using a bounded variance assumption and the Chebychev inequality.

To observe this behavior by simulation, let's look at the Poisson distribution. A Poisson RV has a single parameter λ which represents the mean number of occurances or counts, such as the mean number of hits your Youtube video gets in a day. Then the probability that your video gets i hits in one day is given by the pmf: $p(i) = \frac{\lambda^i e^{-\lambda}}{i!}$. So what if you totalled the number of hits you received each day for a week, month, or year and divided by that many days? The law of large numbers asserts that as n gets large this should go to the mean λ. Below, we create a Poisson distribution where $\lambda = 5$ hits per day. We retrieve a random variate each day and model the law of large numbers over time. Try running the cell multiple times and see how much variation there is for different runs.

In [97]:
```
lamb = 5. #Our rate of hits per day
N = 7 #Number of days we are going to simulate over
x = [st.poisson.rvs(lamb)]    #Gets the number of hits on the first da
y and stores it in a vector
y = [np.random.poisson(lamb)] #Same using numpy instead of scipy.stats
Sx = [0] #Sx will represent cumulative sums of the random variates at
different times
Sy = [0]
Ax = [0.0] #Ax will be an array of averages, Sx/n, at different times
Ay = [0.0]

for n in range(1,N+1): #Simulates over N days
    x.append(st.poisson.rvs(lamb)) #Appends the number of hits of the
next day to the vector
    y.append(np.random.poisson(lamb))
    Sx.append(Sx[n-1]+x[n]) #Appends the sum at n to the vector
    Sy.append(Sy[n-1]+y[n])
    Ax.append(Sx[n]/(1.*n)) #Appends the sum over n to the vector
    Ay.append(Sy[n]/(1.*n))

n = np.linspace(0,N,N+1) #Generates an array of N+1 points from 0 to N
for graphing purposes

plt.plot(Sx, label = 'Total # of Hits X') #Plots the total number of h
its
plt.plot(Sy, label = 'Total # of Hits Y')
plt.plot(n, lamb*n, label = 'lambda*n') #Plots a line based on the rat
e given
plt.title('Cumulative number of hits')
plt.ylabel('Number of hits')
plt.xlabel('Number of days')
plt.legend()

plt.figure() #Creates a new figure
plt.plot(Ax, label = 'S_x/n') #Plots the average number of hits
plt.plot(Ay, label = 'S_y/n')
plt.plot(lamb*np.ones(N+1), label = 'rate lambda') #Plots the expected
average
plt.title('Running average number of hits per day')
plt.ylabel('Running average number of hits per day')
plt.xlabel('Number of days')
plt.legend()
```

```
Out[97]: <matplotlib.legend.Legend at 0x1140de790>
```



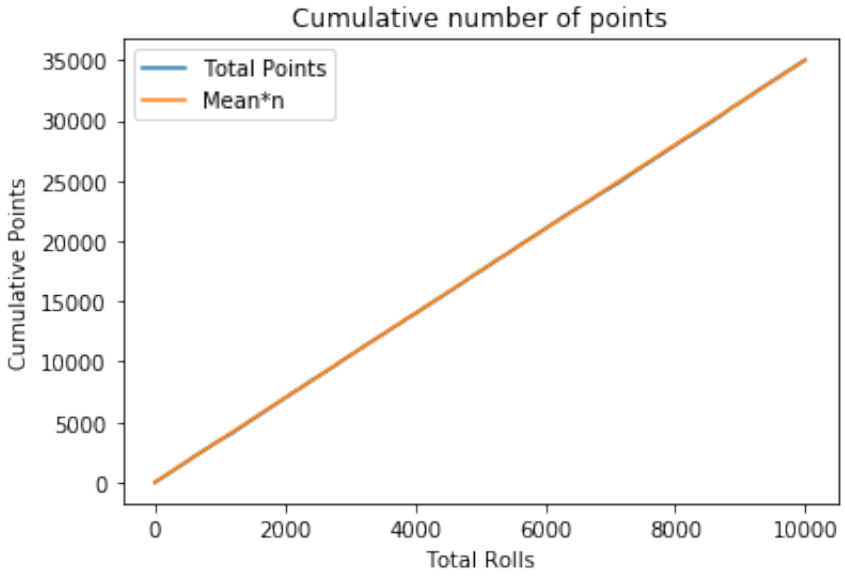


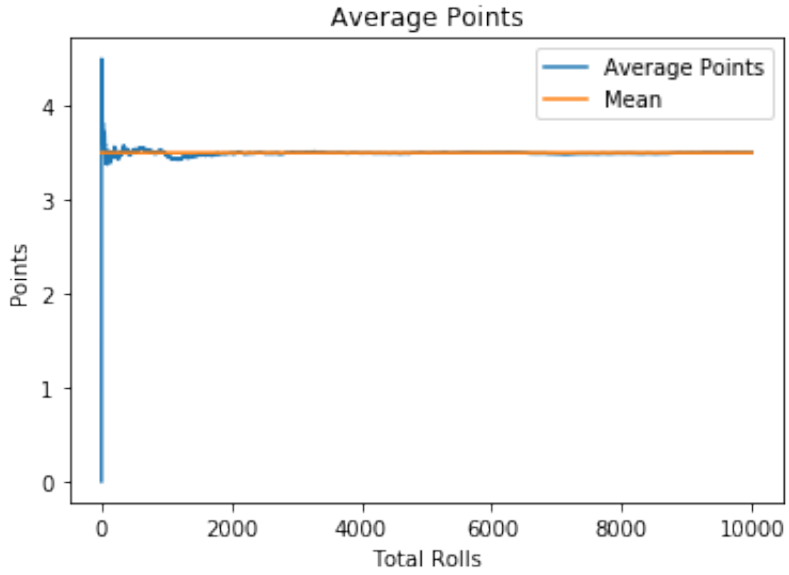
You can play with the above code. Change λ or change the number of days we simulate over. You should see that $\frac{S_n}{n} \rightarrow \lambda$ as the number of days, n, increases.

Problem 2: Using your discrete die rolling RV, once again simulate 10000 rolls of the die. Show graphically that the law of large numbers is maintained for the average of the numbers of the first n rolls as $n \rightarrow \infty$. What would be a rough approximation of the sum? Of the average? (Hint: There is no parameter λ associated with the roll of a die. But the mean value of a roll plays the same part.)

In [115]:
```
########Student Answer##############
N = 10000 #Roll 10,000 times

p = (1./6,1./6,1./6,1./6,1./6,1./6)
c = (1,2,3,4,5,6)
Xdie = st.rv_discrete(values=(c,p)) #Simulate of roll a die
Mdie = Xdie.mean() #Mean value in a roll
x = [0]
Sx = [0]
Ax = [0.0]

for n in range(1,N+1):
    x.append(Xdie.rvs(size = 1))
    Sx.append(Sx[n-1]+x[n]) #Calculate sum and append to Sx
    Ax.append(Sx[n]/(1.*n)) #Calculate average and append to Ax

n = np.linspace(0,N,N+1) #Generates an array of N+1 points from 0 to N
for graphing purposes

plt.plot(Sx, label = 'Total Points') #Plots the total number of points
obtained
plt.plot(n, Mdie*n, label = 'Mean*n') #Plots the mean value of rolling
a die
plt.title('Cumulative number of points')
plt.ylabel('Cumulative Points')
plt.xlabel('Total Rolls')
plt.legend()

plt.figure() #Creates a new figure
plt.plot(Ax, label = 'Average Points') #Plots the average number of po
ints obtained
plt.plot(Mdie*np.ones(N+1), label = 'Mean') #Plots the expected averag
e
plt.title('Average Points')
plt.ylabel('Points')
plt.xlabel('Total Rolls')
plt.legend()
####################################
```

```
Out[115]: <matplotlib.legend.Legend at 0x114eee950>
```





A rough approximation to the sum would be
$Mean\ Value\ of\ Each\ Roll \times Numbers\ of\ Rolls$.

A rough approximation to the average would simply be the $Mean\ Value\ of\ Each\ Roll$.

Additional Lab Questions:

For the completeion of this lab, make sure to do questions 1-2 as well as these two additional questions:

Problem 3: If you thought you could get out of your first probability lab without some questions on poker hands, I'm sorry, but they're too good to pass up. Suppose you draw five cards from a standard 52 card deck.

1. Caclulate the probabilities of getting a TWO PAIR, THREE OF A KIND, FULLHOUSE, and a FLUSH.
2. Simulate 1,000,000 poker hands, count the number of times you get for each of the above hands and find the empirical probability of getting each of the above hands. Your simulated hands should include all possible poker hands. (Hint: One way to represent a random poker hand is to use a 4 by 13 binary array with ones placed at five randomly chosen locations. The np.sum command can be used to compute row or column sums.)
3. Do the probabilities match up relatively well?

In [363]:
```
########Student Answer###############
# Theoretical Probability Calculation
Pick5 = sp.misc.comb(52,5) # All possible hands when we pick 5 cards

TwoPair_1 = (sp.misc.comb(13,2)*sp.misc.comb(4,2)**2*sp.misc.comb(11,1
)*sp.misc.comb(4,1))/Pick5
ThreeOfAKind_1 = (sp.misc.comb(13,1)*sp.misc.comb(4,3)*sp.misc.comb(12
,2)*sp.misc.comb(4,1)**2)/Pick5
FullHouse_1 = (sp.misc.comb(13,1)*sp.misc.comb(4,3)*sp.misc.comb(12,1)
*sp.misc.comb(4,2))/Pick5
Flush_1 = (sp.misc.comb(13,5)*sp.misc.comb(4,1))/Pick5

# Simulate Probability Calculation
N = 1000000 # Try 1,000,000 times

TP_count = 0 # Number of times of Two Pair
TOAK_count = 0 # Number of times of Three Of A Kind
FH_count = 0 # Number of times of Full House
F_count = 0 # Number of times of Flush
OTHER_count = 0 # Number of other conditions

# Get N hands of poker
for i in range(1,N+1,1):
    # Ready to get a new hand of poker
    cards = np.zeros((4,13))
    j = 1
    while j < 6: # Randomly pick 5 different cards
        suit = np.random.randint(1,5) # Return a random integer betwee
n [1,5)
        rank = np.random.randint(1,14) # Return a random integer betwe
en [1,13)
```

```
        if cards[suit-1,rank-1] == 0: # We can not get the same card t
wice
            cards[suit-1,rank-1] = 1
            j = j + 1

    # Check Two Pair, Three Of A Kind, and Full House
    rankCount = np.sum(cards,0)
    TP_check = 0 # If it satisfy 2 + 2 + 1, it will count to 9
    TOAK_check = 0 # If it satisfy 3 + 1 + 1, it will count to 11
    FH_check = 0 # If it satisfy 3 + 2, it will count to 13
    for k in range(13):
        if rankCount[k] == 1:
            TP_check = TP_check + 1
            TOAK_check = TOAK_check + 1
            FH_check = FH_check - 100
        elif rankCount[k] == 2:
            TP_check = TP_check + 4
            TOAK_check = TOAK_check - 100
            FH_check = FH_check + 4
        elif rankCount[k] == 3:
            TP_check = TP_check - 100
            TOAK_check = TOAK_check + 9
            FH_check = FH_check + 9
        elif rankCount[k] == 4:
            TP_check = TP_check - 100
            TOAK_check = TOAK_check - 100
            FH_check = FH_check - 100

    # Check Flush
    suitCount = np.sum(cards,1)
    F_check = 0 # If it satisfy 5 cards in a suit, it will count to 25
    for k in range(4):
        if suitCount[k] == 5:
            F_check = 25

    # Write conclusion about this hand of poker
    if TP_check == 9:
        TP_count = TP_count + 1
    elif TOAK_check == 11:
        TOAK_count = TOAK_count + 1
    elif FH_check == 13:
        FH_count = FH_count + 1
    elif F_check == 25:
        F_count = F_count + 1
    else:
        OTHER_count = OTHER_count + 1

Total = (float)(TP_count + TOAK_count + FH_count + F_count + OTHER_cou
nt)
```

```
TwoPair_2 = format(TP_count/Total, '.13f') # Calculate probability of
Two Pair
ThreeOfAKind_2 = format(TOAK_count/Total, '.13f') # Display 13 digits
after decimal
FullHouse_2 = format(FH_count/Total, '.13f')
Flush_2 = format(F_count/Total, '.13f')

print "Theoreotical Probability"
print "Two Pair:         ", TwoPair_1
print "Three of a Kind:  ", ThreeOfAKind_1
print "Full House:       ", FullHouse_1
print "Flush:            ", Flush_1
print ''
print "Empirical Probability"
print "Two Pair:         ", TwoPair_2
print "Three of a Kind:  ", ThreeOfAKind_2
print "Full House:       ", FullHouse_2
print "Flush:            ", Flush_2
####################################
```

```
Theoreotical Probability
Two Pair:         0.0475390156062
Three of a Kind:  0.0211284513806
Full House:       0.00144057623049
Flush:            0.00198079231693

Empirical Probability
Two Pair:         0.0476500000000
Three of a Kind:  0.0208680000000
Full House:       0.0014470000000
Flush:            0.0020160000000
```

Theoretical Probability and Empirical Probability match up relatively well!

Problem 4: A classic problem when being introduced to probability is the Monty Hall problem. If you've ever seen "Let's Make a Deal" on television, this problem takes from that show. You're the contestant. The host of the show gives you three doors to choose from. One door chosen at random holds a grand prize and the other two hold worthless items. You choose your door, and then the host reveals one of the doors you didn't choose such that it always holds a worthless item. (If you initially choose the door with the grand prize, the host reveals either of the other doors with equal probability.) So now there are two doors left and the host asks you whether you would like to switch. What should you do?

1. Write down your first reaction? Would you switch doors or keep the one you have? Why?
2. Create this scenario and simulate the strategy of sticking with the same door 1,000,000 times. What percentage of time did you win?
3. Simulate the strategy of switching doors 1,000,000 times. What percentage of time did you win?
4. Which strategy would you use now? Explain why this is the case.

In [494]:
```
########Student Answer##############
print "First Reaction: Switch. Because I think either choice have equa
l probability to win."
print ''

N = 1000000 # Simulate 1,000,000 times

Switch_Win = 0
Keep_Win = 0

for i in range(N):
    doors = np.zeros(3)
    door_prize = np.random.randint(0,3) # Hidden prize is in this door
    doors[door_prize] = 1 # Set it to 1
    choose_1 = np.random.randint(0,3) # Choose one door

    # Open a door with worthless items
    switch = 0 # The door left over
    if choose_1 == door_prize:
        if doors[0] == 1:
            door_open = np.random.randint(1,3)
        elif doors[1] == 1:
            door_open = np.random.choice((0,2))
        else:
            door_open = np.random.randint(0,2)
        doors[door_open] = -1 # Opened door = -1
        switch = 3 - door_open - door_prize
    elif choose_1 != door_prize:
        doors[3 - choose_1 - door_prize] = -1
        switch = door_prize

    if switch == door_prize: # If we win after we switch
        Switch_Win = Switch_Win + 1
    else: # If we win when we keep
        Keep_Win = Keep_Win + 1

Total = (float)(Switch_Win + Keep_Win)

print "Probability of Win when Keep:   ", format((float)(Switch_Win/To
tal), '.13f')
print "Probability of Win when Switch: ", format((float)(Keep_Win/Tota
l), '.13f')
####################################
```

```
First Reaction: Switch. Because I think either choice have equal pro
bability to win.

Probability of Win when Keep:    0.6667490000000
Probability of Win when Switch:  0.3332510000000
```

I'll switch now. It's more likely to win the prize. That's because when we choose a door, we got $\frac{1}{3}$ chance of win, and there're $\frac{2}{3}$ of chance that the other two doors have the prize. But after the host open a door, the door left over still have $\frac{2}{3}$ of chance, while our original choice still have $\frac{1}{3}$ chance to win. So we should switch.