9B. Random Simulations

Topics:
- The class `random`
- Estimating probabilities
- Estimating averages
- More occasions to practice iteration

The `random` Module

Contains functions that can be used in the design of random simulations.

We will practice with these:

```
random.randint(a,b)
random.uniform(a,b)
random.normalvariate(mu,sigma)
```

And as a fringe benefit, more practice with for-loops

Generating Random Integers

If `a` and `b` are initialized integers with `a < b` then

```
i = random.randint(a,b)
```

assigns to `i` a "random" integer that satisfies

```
a <= i <= b
```

That is, we randomly select an element from the set {a,a+1,…,b} and assign it to n

What Does "Random" Mean?

```
import random
for k in range(1000000):
   i = random.randint(1,6)
   print i
```

The output would "look like" you rolled a dice one million times and recorded the outcomes.

No discernible pattern.

Roughly equal numbers of 1's, 2's, 3's, 4's, 5's, and 6's.

Renaming Imported Functions

```
import random
for k in range(1000000):
   i = random.randint(1,6)
   print i
```

```
from random import randint as randi
for k in range(1000000):
   i = randi(1,6)
   print i
```

Handy when the names are long or when you just want to name things your way.

Random Simulation

We can use `randint` to simulate genuinely random events, e.g.,

Flip a coin one million times and record the number of heads and tails.

Coin Toss

```
from random import randint as randi

N = 1000000
Heads = 0
Tails = 0
for k in range(N):
    i = randi(1,2)
    if i==1:
      Heads = Heads+1
    else:
      Tails = Tails+1
print  N, Heads, Tails
```

The "count" variables **Heads** and **Tails** are initialized

randi returns 1 or 2

Convention: "1" is heads

Convention: "2" is tails

A Handy Short Cut

Incrementing a variable is such a common calculation that Python supports a shortcut.

These are equivalent:

```
x += 1

x = x+1
```

x += c   is equivalent to x = x+c

Coin Toss

```
from random import randint as randi

N = 1000000
Heads = 0
Tails = 0
for k in range(N):
    i = randi(1,2)
    if i==1:
      Heads+=1
    else:
      Tails+=1
print  N, Heads, Tails
```

The "count" variables **Heads** and **Tails** are initialized

randi returns 1 or 2

Convention: "1" is heads

Convention: "2" is tails

Sample Outputs

```
        N = 1000000
Heads =   500636
Tails =   499364
```

```
        N = 1000000
Heads =   499354
Tails =   500646
```

Different runs produce different results.

This is consistent with what would happen if we physically tossed a coin one million times.

Estimating Probabilities

You roll a dice. What is the probability that the outcome is "5"?

Of course, we know the answer is 1/6. But let's "discover" this through simulation.

Dice Roll

```
from random import randint as randi
N = 6000000
count = 0
for k in range(N):
    i = randi(1,6)
    if i==5:
      count+=1
prob = float(count)/float(N)
print prob
```

N is the number of "experiments".

i is the outcome of an experiment

prob is the probability the outcome is 5

Dice Roll

```
from random import randint as randi
N = 6000000
count = 0
for k in range(N):
    i = randi(1,6)
    if i==5:
      count+=1
prob = float(count)/float(N)
print prob
```

```
Output:

         .166837
```

Discovery Through Simulation

Roll three dice.

What is the probability that the three outcomes are all different ?

If you know a little math, you can do this without the computer. Let's assume that we don't know that math.

Solution

```
N = 1000000
count = 0
for k in range(1,N+1):
    d1 = randi(1,6)
    d2 = randi(1,6)
    d3 = randi(1,6)
    if d1!=d2 and d2!=d3 and d3!=d1:
        count +=1
    if k%100000==0:
       print k,float(count)/float(k)
```

Note the 3 calls to randi

Prints snapshots of the probability estimates every 100,000 trials

Sample Output

```
     k          count/k
----------------------
  100000       0.554080
  200000       0.555125
  300000       0.555443
  400000       0.555512
  500000       0.555882
  600000       0.555750
  700000       0.555901
  800000       0.556142
  900000       0.555841
 1000000       0.555521
```

Note how we say "sample output" because if the script is run again, then we will get different results.

Educated guess true prob = 5/9

Generating Random Floats

Problem:

Randomly pick a float in the interval [0,1000].

What is the probability that it is in [100,500]?

Answer = (500-100)/(1000-0) = .4

Generating Random Floats

If `a` and `b` are initialized floats with `a < b` then

```
x = random.uniform(a,b)
```

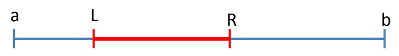
assigns to `x` a "random" float that satisfies

```
a <= x <= b
```

The actual probability that x is equal to a or b is basically 0.

The Uniform Distribution

Picture:



The probability that

L <= `random.uniform(a,b)` <= R

is true is

(R-L) / (b-a)

Illustrate the Uniform Distribution

```
from random import uniform as randu
N = 1000000
a = 0; b = 1000; L = 100; R = 500
count = 0
for k in range(N):
    x = randu(a,b)
    if L<=x<=R:
        count+=1
prob = float(count)/float(N)
fraction = float(R-L)/float(b-a)
print prob,fraction
```

Pick a float in the interval [0,1000]. What is the prob that it is in [100,500]?

Sample Output

```
Estimated probability:  0.399928

          (R-L)/(b-a) :  0.400000
```

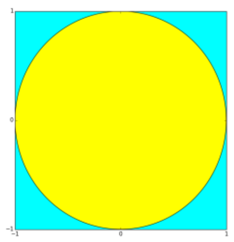
Estimating Pi Using random.uniform(a,b)

Idea:

Set up a game whose outcome tells us something about pi.

This problem solving strategy is called Monte Carlo. It is widely used in certain areas of science and engineering.

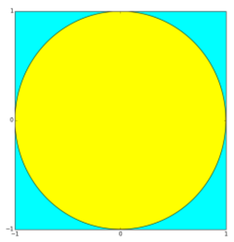
The Game



Throw darts at the 2x2 cyan square that is centered at (0,0).

If the dart lands in the radius-1 disk, then count that as a "hit".

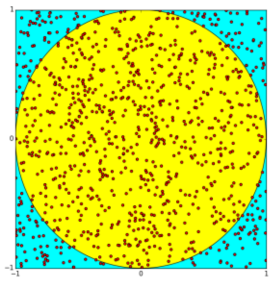
3 Facts About the Game



1. Area of square = 4
2. Area of disk is pi since the radius is 1.
3. Ratio of hits to throws should approximate pi/4 and so

   4*hits/throws "=" pi

Example



1000 throws

776 hits

Pi = 4*776/1000
 = 3.104

When Do We Have a Hit?

The boundary of the disk is given by

x**2 + y**2 = 1

If (x,y) is the coordinate of the dart throw, then it is inside the disk if

```
x**2+y**2 <= 1
```

is `True`.

Solution

```
from random import uniform as randu
N = 1000000
Hits = 0
for throws in range(N):
    x = randu(-1,1)
    y = randu(-1,1)
    if x**2 + y**2 <= 1 :
        # Inside the unit circle
        Hits += 1
piEst = 4*float(Hits)/float(N)
```

Note the 2 calls to randu

Repeatability of Experiments

In science, whenever you make a discovery through experimentation, you must provide enough details for others to repeat the experiment.

We have "discovered" pi through random simulation. How can others repeat our computation?

random.seed

What we have been calling random numbers are actually pseudo-random numbers.

They pass rigorous statistical tests so that we can use them as if they are truly random.

But they are generated by a program and are anything but random.

The `seed` function can be used to reset the algorithmic process that generates the pseudo random numbers.

Repeatable Solution

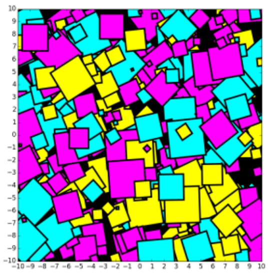
```
from random import uniform as randu
from random import seed
N = 1000000; Hits = 0
seed(0)
for throws in range(N):
    x = randu(-1,1); y = randu(-1,1)
    if x**2 + y**2 <= 1 :
        Hits += 1
piEst = 4*float(Hits)/float(N)
```

Now we will get the same answer every time

Another Example

Produce this "random square" design.

Think: I toss post-its of different colors and sizes onto a table.



Solution Framework

Repeat:

1. Position a square randomly in the figure window.
2. Choose its side length randomly.
3. Determine its tilt randomly
4. Color it cyan, magenta, or, yellow randomly.

Getting Started

```
from random import uniform as randu
from random import randint as randi
from SimpleGraphics import *
n = 10
MakeWindow(n,bgcolor=BLACK)
for k in range(400):
    # Draw a random colored square
    pass
ShowWindow()
```

Note the 3 calls to randi

"pass" is a necessary place holder. Without it, this script will not run

Positioning the square

The figure window is built from **MakeWindow(n)**.

A particular square with random center (x,y) will be located using **randu** :

```
x = randu(-n,n)
y = randu(-n,n)
```

The Size s of the square

Let's make the squares no bigger than n/3 on a side.

```
s = randu(0,n/3.0)
```

The tilt of the square

Pick an integer from 0 to 45 and rotate the square that many degrees.

```
t = randi(0,45)
```

The Color of the square

With probability 1/3, color it cyan
With probability 1/3 color it magenta
With probability 1/3, color it yellow.

```
i = randi(1,3)
if i==1;
   c = CYAN
elif i==2:
   c = MAGENTA
else:
   c = YELLOW
```

The Final Loop Body

```
x = randu(-n,n)          The center
y = randu(-n,n)
s = randu(0,n/3.0)       The side
t = randi(0,45)          The tilt
i = randi(1,3)
if i==1:
   c = CYAN
elif i==2:               The color
   c = MAGENTA
else:
   c = YELLOW
DrawRect(x,y,s,s,tilt=t,FillColor=c)
```

Stepwise Refinement

Appreciate the problem-solving methodology just illustrated.

It is called stepwise refinement.

We started at the top level. A for-loop strategy was identified first. Then, one-by-one, we dealt with the location, size, tilt, and color issues.

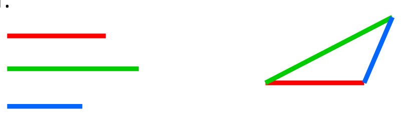
Another Example: TriStick

Pick three sticks each having a random length between zero and one.

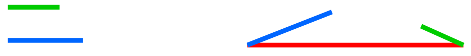
You win if you can form a triangle whose sides are the sticks. Otherwise you lose.

TriStick

Win:



Lose:



The Problem to Solve

Estimate the probability of winning a game of TriStick by simulating a million games and counting the number of wins.

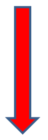
We proceed using the strategy of step-wise refinement…

Pseudocode

```
Initialize running sum variable.
Repeat 1,000,000 times:
    Play a game of TriStick by picking
        the three sticks.
    If you win
        increment the running sum
Estimate the probability of winning
```

Pseudocode: Describing an algorithm in English but laying out its parts in python style

The Transition



Pseudocode

via stepwise refinement

Finished Python Code

First Refinement

```
Initialize running sum variable.
Repeat 1,000,000 times:
    Play a game of TriStick by picking
        the three sticks.
    If you win
        increment the running sum
Estimate the probability of winning
```

Turn the loop-related stuff into Python

Next, Refine the Loop Body

```
# Initialize running sum variable.
wins = 0
for n in range(1000000):
    Play the nth game of TriStick by
        picking the three sticks.
    If you win
        increment the running sum.
# Estimate the prob of winning
p = float(wins)/1000000
```

Refine the Loop Body

```
Play the nth game of TriStick by picking
  the three sticks.
If you win
    increment the running sum.
```

```
a = randu(0,1)
b = randu(0,1)
c = randu(0,1)
if a<=b+c and b<=a+c and c<=a+b:
    wins +=1
```

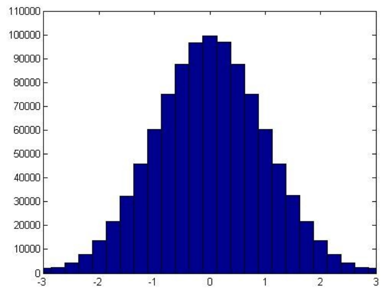
The 3 sticks

Key Problem-Solving Strategy

Progress from pseudocode to Python through a sequence of refinements.

Comments have an essential role during the transitions. They remain all the way to the finished code.

Final "Random" Topic: The Normal Distribution



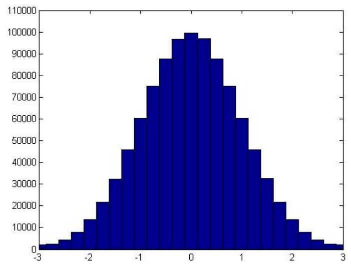
Generating floats from the Normal Distribution

If `mu` and `sigma` (positive) are floats, then

```
x = random.normalvariate(mu,sigma)
```

assigns to `x` a "random" float sampled from the normal distribution with mean mu and standard deviation sigma

Normal Distribution Mean = 0, Standard Deviation = 1



Typical Situation: Test Scores

```
from random import normalvariate as randn
for k in range(450):
    x = randn(70,7)
    print round(x)
```

This would look like a report of test scores from a class of 450 students.

The mean is approximately 70 and the standard deviation is approximately 7.

More on Standard Dev

Generate a million random numbers using

```
random.normalvariate(mu,sigma)
```

and confirm that the generated data has mean mu and std sigma

Checking Out `randn`

```
N = 1000000; sum1 = 0; sum2 = 0
mu = 70; sigma = 7
for k in range(N):
    x = randn(mu,sigma)
    sum1 += x
    sum2 += (x-mu)**2
ApproxMean = float(sum1)/float(N)
ApproxSTD  = sqrt(float(sum2)/float(N))
```

Sample Output:

```
70.007824    6.998934
```

Final Reminder

randi, **randu**, and **randn** are RENAMED versions of

```
random.randint
random.uniform
random.normalvariate
```