15-110: Principles of Computing, Spring 2018

Lab 8 – Thursday, March 22

Goals

This lab is aimed at helping you understand the use of random numbers in Python. When you are done, you should be able to:

1. Use randrange() to randomly generate integers in a given range.
2. Write functions that simulate a large number of trials of a random process.

Part 1: TA Demonstrations (Random Numbers in Python)

In class, you learned how to generate random numbers in Python.

```
import random
random.randrange(15,110)  # returns random integer 15 <= x < 110
random.random()           # returns random float 0.0 <= x < 1.0
random.uniform(1,10)      # returns random float 1.0 <= x < 10.0
random.randint(1,10)      # returns random integer 1 <= x <= 10
```

Class discussion:

1. How would you generate the roll of one die?

2. To generate a random floating point value between -1.0 and 1.0 inclusive, you could use `random.uniform`, but this does not include the upper bound. If you want to include the upper bound, you can do the following instead:

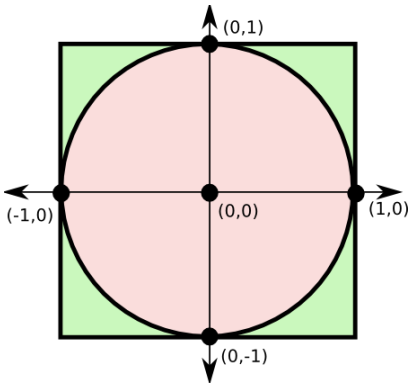
- Generate a random integer between 0 and 2000000000 (2 billion) inclusive.
- Subtract 1000000000 (1 billion) from the random integer.
  (Now the integer is between -1000000000 and 1000000000 inclusive.)
- Divide this result by 1000000000.0 to get your random floating point value between -1.0 and 1.0 inclusive.

```
(randint(0, 2000000000) - 1000000000) / 1000000000.0

(randrange(0, 2000000001) - 1000000000) / 1000000000.0
```

Self-Directed Activities

1. The Monte Carlo method is a computational technique that works by calculating a statistical summary of a large number of random operations. One simple use of the Monte Carlo Method is to approximate the value of pi.



Consider a unit circle (a circle of radius 1 whose center is at (0,0)) on a Cartesian plane, and a square whose corners are at (-1,1), (1,1), (1,-1), and (-1,-1). (Shown above.). If a point is chosen randomly within the square, the probability of it also being within the circle is determined by the ratio of the area of the circle (pi) to the area of the square (4).
We can use this fact to build a procedure to estimate the value of pi:

1. Let *n* be the number of points to be randomly generated.
2. Set a *counter* to 0.
3. Repeat the following *n* times:
    a) Generate a random point in the square:
    Let x be a randomly generated value between -1.0 and 1.0 inclusive.
    Let y be a randomly generated value between -1.0 and 1.0 inclusive.
    b) Determine whether the distance of this point from the origin (i.e. sqrt(x^2+y^2)) is less than or equal to 1.0. If so, the point lies "inside" the unit circle so add 1 to the counter.
4. Compute the *ratio* of the total number of points inside the unit circle to the total number of points generated.
5. Multiply this *ratio* by 4 to obtain an estimate of pi. Return this final value.

Write a Python function `estimate_pi(n)` in the file `estimate_pi.py` that implements the algorithm above. Once you have it implemented, test it with n = 1000. How many digits of your estimate of pi are accurate? Repeat the experiment with n = 10000, 100000, 1000000. Share your results with classmates to compare your answers.

2. Suppose you want to make a batch of 25 raisin cookies. How many raisins should you use to make sure nearly every cookie has at least 1 raisin? Assume that raisins are randomly distributed in the batter, so each raisin is equally likely to go into each cookie.

   First, write a Python function `make_batch(c, r)` in the file `raisin_cookies.py` where c represents the number of cookies to be made and r represents the number of raisins to mix into the batter. This function returns `True` if and only if every one of cookies has at least 1 raisin. Implement the function as follows:

   1. Create a list of *c* zeros called *cookie*. This list represents a set of counters that keep track of the number of raisins in each cookie.

   2. Distribute the raisins randomly (i.e. do this *r* times):
      - Compute a random cookie *index*.
        (Think: Generate a random integer between what two values?)
      - Increment *cookies[index]* by 1.

   3. For debugging purposes, print out this list.

   4. See if every cookie has at least one raisin:
      - Use a linear search on the *cookie* list to see if any counter is zero; if any counter is zero, immediately return False.
      - If the search completes without finding a zero, return True.

   Test your function with the following function calls in the Python3 interpreter:

```
make_batch(10, 10)
make_batch(10, 20)
make_batch(10, 50)
```

   See that you get the correct answers. You may need to run each several times since you're using random numbers here. Once you are sure it works, comment out step 3 above.

   Next, write a function `probability(c, r)` in the same file to estimate the probability that for *c* cookies and *r* raisins, every cookie will have at least one raisin:

   1. Set *count* = 0
   2. Do this 1000 times:
      - Call *make_batch(c, r)* and if the returned result is True, increment count by 1.
   3. After the loop, return count / 1000, the fraction of times every cookie has at least one raisin.

   Call `probability(25, 100)` to estimate the probability that every cookie in a batch of 25 cookies will have at least 1 raisin if there are 100 raisins in all.
   Call `probability(25, 100)` again. Make sure you understand why you do not get the same answer. Discuss briefly with another student in lab.

FURTHER WORK [OPTIONAL]:

Now we want to answer a different question: how many raisins do we have to use to get a good probability that every cookie has a raisin? Write a Python function `raisins_needed(num_cookies, error)` in the file `raisin_cookies.py`. The idea is to start with the same number of raisins as cookies (we need at least that many), and call your `probability` function to get the probability that every cookie has a raisin. Keep increasing the number of raisins until the answer from your `probability` function is close enough to 1 (certainty). The uncertainty to allow is given by error. For example, an error of 0.1 means that we want an answer with certainty of 99% (i.e. a probability of 0.99).

The algorithm:

1. Set *num_raisins* = *num_cookies*.
2. Set *diff* to 1.0 - *probability(num_cookies, num_raisins)*
3. While *diff* is greater than *error*, increase *num_raisins* by one and recalculate *diff* as above.
4. Once *diff* is less than or equal to *error*, return *num_raisins*.

Example usage (note that for large numbers of cookies and/or small error values this will take a while to run):

```
>>> raisins_needed(1, .1)
1
>>> raisins_needed(2, .1)
5
>>> raisins_needed(10, .1)
45
```

Experiment with other error values and cookie amounts.

Submission

When you finish the lab, you should be inside the `lab8` folder, which is inside the `private/15110` directory. When you type `ls` and press the Enter key, you should see the following files: `estimate_pi.py`, `raisin_cookies.py`. Once you see all files, please type `cd ..` to move up one folder and press the Enter key. Then, zip your `lab8` folder by typing `zip –r lab8.zip lab8` and you should see a `lab8.zip` file in the current folder if you type `ls`. Please submit the zipped file `lab8.zip` on Autolab under 'lab 8'.