COM110: Fall 2019 - Lab 8
Simulation using random numbers, image processing

1) Collision avoidance. During Lab07 and Tuesday's class you were asked to revisit the lab 4 house animation exercise in lab4starlist-v2.py. Locate it on today's lab folder and test it. Again, review the part of the code that achieves the "star-house-collision avoidance" and make sure you understand it. (No need to explain the details of the insideHouse() method itself. Just how the method is used with the while loop.)

2) Note that you could use a similar collision avoidance technique when avoiding putting the multiple words in your PA04 word cloud in the same location. (This is just one option.) Let's practice this technique by generating 6 *distinct* random integers between 1 and 10. (I.e., no repeats – no "collisions" allowed). How? Read on…

a. In a new module file, start by generating 6 random integers between 1 and 10, not checking for repeats for now, and adding each of them to a list. Print the list to check that your code is working. Notice that if you run your program repeatedly, you will often get repeated numbers among the six.

☺ **Get check 1** ☺

b. Now add a *while loop*, similar to lab4starlist-v2.py above, so that if the randomly generated value has previously been added to your list of numbers, it tries again until it finds one that is new. Only after you're sure you have a new number (not already in the list) should you append it to the list.

☺ **Get check 2** ☺

3) Monte Carlo Simulation. The Monte Carlo method is a way to approximate a value using repeated random sampling. We did this, for example, when we approximated the probability of rolling a 7 when tossing two dice in check 3 above. We will use the Monte Carlo method to estimate the value of the world's most famous transcendental number: π.

For this checkpoint you will simulate the act of throwing thousands of darts onto a square (2x2) dart board. (No graphical output required.) As the "darts" are being "thrown" onto uniform-randomly generated points in the square, we will count how many of the darts "land" inside the circle that is inscribed by our 2x2 square. Since we know a 2x2 square has an area of exactly 4, we can use the percentage of darts that land inside the circle to estimate the area of the circle!

As an example, if 50% of the darts end up inside the circle, it would mean that the area of the circle is roughly 50% of 4, which is 2. (If you draw yourself a picture of the circle inscribed by our square, you should be able to visually see whether this is lower or higher than the actual answer.)

Note that this circle, since it's inscribed in a 2x2 square, ***has a radius of 1***. Now consider the formula for the area of a circle (look it up online if you don't remember it). Can you see why approximating the area of this circle is the same as approximating π?

A. Have your program prompt the user for how many dart throws s/he wants to simulate. Use a loop to simulate that number of dart throws, keeping track of how many darts

"land" inside the circle. (See implementation tips below.) For now just print out how many darts land in the circle. Implementation tips:

- A framework for the code is provided for you in **pi.py**.
- You should imagine a hypothetical 2x2 square dartboard that goes from x = -1 to x = 1, and from y = -1 to y = 1, it is centered at (0,0). Each dart's randomly generated landing point (x,y) needs to be within this 2x2 square, and all points in the square must have uniform probability of being "hit." With a little thought, this can be achieved by calls to the random() function. (Recall our in-class practice exercises… for this checkpoint we're asking: how would you generate a random **floating point number** between -1 and 1?)
- To check if your randomly generated point (x,y) is inside the circle of radius 1, you might recall that the formula for a circle is $x^2+y^2 = r^2$. Hence if x^2+y^2 is <= 1, that means the dart landed inside the circle.
- Use an accumulator variable to count the number of darts that land in the circle.

☺Get check3 ☺

B. Now compute the fraction of darts that landed in the circle, then use it to approximate area of the circle, which is also the approximate value of π. See how many dart-throws you need to simulate to get close to the correct value of π!

☺Get check 4 ☺

5) Image processing: Refer to section 4.8.4 of Zelle for this part of the lab. Pause and take a moment to read and understand section 4.8.4. Let us know if you any part is confusing or if you have any questions.

☺Get check 5 ☺

a. Open and review the code in imageTest.py - it has operations that we learned in class on Thursday, the code starts by opening an image (camelsLogo.gif), print some of its details, and then changes a whole column of pixels in the image to red, we finally save the new image data into a new file - camelsLogo2.gif. After reviewing and understanding the code, run it, see the printouts and image changes, also see the new created image file in the same folder.

b. Now modify the code to ADD more red columns all the way across the image, so it looks something like below:

(hint) use range() in the **for** loop, consider using the third step parameter for column spacing.

☺Get check 6☺

c. Note that the Zelle Image class allows you to save your Image objects out to a new .gif files. Do this with your modified camelsLogo image, calling it camelsLogoBars.gif. Check that the new gif file was created in the same directory as your python program.

☺Get check 7☺

d. Open program, greenImage.py, and review and test it. This program will create a graphical window (800x600 size), create an image object called `flowers` from the file pink.gif, and draw it on the left side of the window. It also creates an empty image object, `noFlowers`, with the same size as pink.gif and draws it on the right side of the window. This new blank image is where we will create a copy of the first image, but with all the flowers removed!

e. In a nested for loop that goes through each pixel of `flowers` (for each `x` value and for each `y` value), get the (r,g,b) color values of each pixel. Per info on page 120 of Zelle, use something like `[r,g,b] = flowers.getPixel(x,y).`

f. For each pixel, use the (r,g,b) info to test to see if the pixel is green rather than pink (i.e., grass/leaves rather than red-ish flowers). (Hm, how might you test this using the values r, g, and/or b?) If it is, then we want to "copy" it over to the new image, since we're trying to omit the flowers and keep the grass pixels. So you should set the corresponding pixel in the new image `noFlowers` to the same color. To do this, you can use the `.setPixel(x,y,color_rgb(r,g,b))` function. (For each pixel that doesn't pass your "greenness" test, you should do nothing to the corresponding pixel in the modified image, leaving it transparent.) Note: if you want to watch as this unfolds, you should add a `win.update()` call after each `.setPixel()` call - [adding this will make your code very slow and take longer, especially on Macs]. Otherwise, your computer will just seem to freeze for a while as its working to finish "painting" all 362 x 547=198,014 pixels. Save your noFlowers image into a new file, nopink.gif.

☺Get check 8☺

Bonuses.

a. **Monte Carlo visualization**: Give the above Monte Carlo simulation (check 3 & 4) a graphical display by drawing a small dart board and displaying each of the points where the darts land (use one color to draw points that land inside the circle and another for points that land outside the circle). Display the approximate area of the circle underneath the dartboard after the simulation completes. Make sure your output mentions the name of the special number being approximated.

b. **Touching up noFlowers**: Instead of leaving transparent pixels where the flowers were removed, "camouflage" them in an attempt to "cover up" the fact that flowers were

removed from the image. You are not expected to come up with a perfect solution for this problem. Bonus credit will be given liberally here, but on a case-by-case basis.

c. **Vision recognition:** In order to do some form of vision recognition, we need to recognize the boundaries/outlines of figures in an image. These boundaries or outlines are the pixels with high contrast. Modify your double loop from checkpoint 8 to check pixels for high contrast. You could consider a pixel to have high contrast if one of the pixels next to it has very different r,g,b values. Decide what the test will be and then test pixels: those with high contrast should get put into the new image; others should not.

☺Get bonus checks☺