Question 1 has three parts wherein you are needing to debug and make small fixes to already existing code. Questions 2 is playing with the random library and loops, while Question 3 introduces you to mouse-driven events in PGL. Do not forget to adjust the README to indicate you have completed the assignment before your final commit!

Get Assignment link: https://classroom.github.com/a/ftJAxr9R

1. You get lots of practice writing your own code in this class. But an equally important, and sometimes harder, skill is to be able to interpret and troubleshoot code that you might not have created from scratch. To that end, this problem poses you with 3 different programs that attempt to solve 3 different goals. Each has several (< 5 or so) mistakes that are causing it to not perform as desired. Find and fix the broken bits, and upload the fixed versions back to Github. *There are not huge numbers of mistakes here. Most of the code is good. If you find yourself rewriting large pieces of code, you've probably missed something easier.*

   (a) Compound interest is a financial concept that generally requires differential equations to fully model. We can get similar results by simulating the process in Python though, no differential equations needed (though they are pretty cool...)! Suppose you have a certain amount of money in the bank. Each month, you will earn a certain amount of interest on that money, depending on the interest rate. Mathematically it looks like

   $$\text{Interest earned} = \text{Interest rate} \times \text{Money in bank}$$

   This earned interest is then added to the amount of money you had in the bank. Score! Suppose though that you are also being a responsible adult and setting aside of bit of your salary to savings each month as well, which we could describe with

   $$\text{Added savings} = \text{Savings rate} \times \text{Month's salary}$$

   So each month, both of those effects (interest earned and new savings added) are contributing to increasing the total amount in savings. This program is supposed to compute, for a given savings rate and salary, how many months of saving it would take to hit a desired amount of money in savings. It has a few problems though at the moment, so your task is to fix it. Doing so will also force you to ask yourself how you could test it to ensure it is working properly. I will give you 2 test cases below, but you can come up with other easily testable arguments as well to ensure everything is working.

   ```
   calc_months_to_savings(1000, 0.01)  →  325
   calc_months_to_savings(1000, 0.10)  →  48
   ```

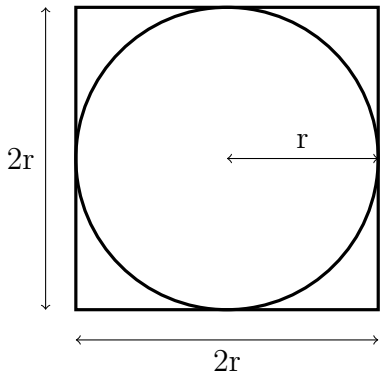
   (b) Last week in your small sections you looked at the Collatz sequence, wherein from some starting point you calculate the next number in the sequence according to:

   If the number is even, divide by two
   If the number is odd, multiply by 3 and add 1

And you saw that it always eventually finds its way to be 1. This program focuses on determining the length of those chains, or how many steps it takes until you get to 1. In particular, its goal is to find the starting number in a specific range that results in the longest chain (greatest number of steps to reach 1). Several things are a bit off though, so again, your goal is to fix it. Similar to the last problem, there are some very simple tests you can make to help check, but I'll give you two other checks below.

```
find_longest_chain(100)  →  118
find_longest_chain(1000)  →  178
```

(c) On Problem Set 2, you estimated the value of π using Leibniz's sequence. Another, more visual method of approximating the value of π is to throw darts randomly at a circular target which is enclosed in a square. All the darts must land within the square, but only a fraction of the darts will also land within the circle. It turns out that the ratio of the number of darts that strike the circle over the total number of thrown darts ends up approximating $\frac{\pi}{4}$! You can actually see this if you just look at the ratio of the areas of a circle enscribed in a square:

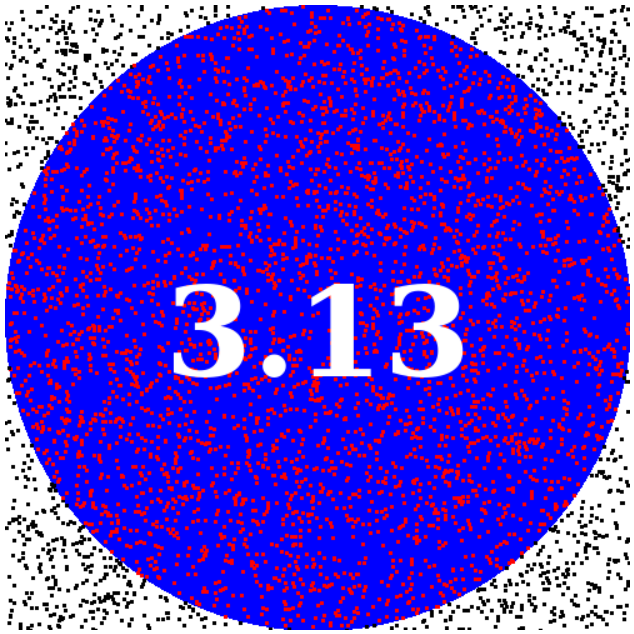


$$\frac{A_{circ}}{A_{rect}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

This program attempts to approximate π by throwing random darts within the square and then checking to see if they landed within the radius of the circle. This can be checked by using the distance formula to find the distance between the center of the circle and where the dart was placed. If your math is a bit rusty, the distance formula is:

$$(\text{distance from center})^2 = (x_{dart} - x_{cent})^2 + (y_{dart} - y_{cent})^2$$

The goal of this program was to not only simulate dart throws to determine the ratio of hits and thus the value of pi, but also to visualize all the dart locations. To this end, it endeavors to use PGL to draw the target and each location of a randomly placed dart. And at the end, it prints the approximate calculated value of π (rounded to 2 decimals) centered on the window. So the final window upon running the program might look something like this:



where red circles are darts that landed within the target circle and black circle are darts that landed outside. Note that, due to the random nature of this approximation, your values will certainly vary from what is shown here! In fact, they should vary from run to run. But if you run it for large enough numbers of darts, the approximation to π will get better and better. At 5000 darts, things usually end up within a few hundredths of 3.14.

2. Write a function called `consecutive_heads` that simulates tossing a coin repeatedly until the specified number of heads appears in a row. The function should take one input, which would be the desired number of consecutive heads. Once the number of specified heads is met, the program should print a line to the console indicating how many coin tosses were needed, and then return that value (the number of needed coin tosses). The output might look something like this:

```
Heads
Heads
Tails
Heads
Heads
Heads
It took 6 tosses to get 3 consecutive heads.
```

You don't need to print out each toss, but I think it helps for debugging purposes. Make sure you include comments and docstrings explaining your code and that you return the final needed number of tosses.

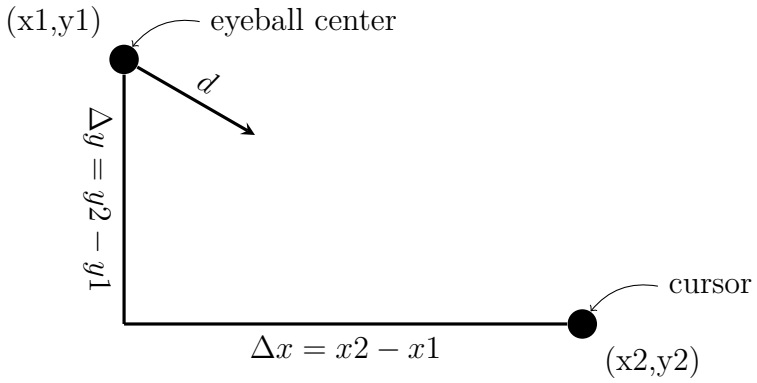
3. In the `Prob3.py`, add code to recreate a cartoon drawing of a face that looks like Figure 1a. It does not need to be exact, but it should be close.

(a) (b)

Figure 1: Examples of the face when the cursor is at different locations. (a) here shows the original face before any interactivity has been added, while (b) shows the face once the callback function has been added and the cursor is to the lower right.

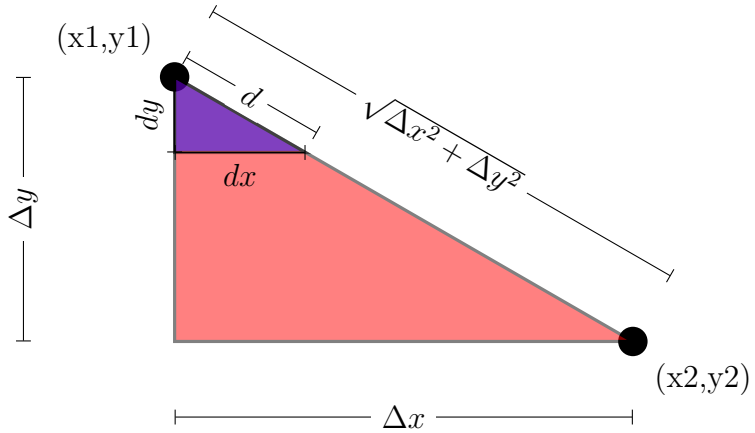
Once you have this, add a callback function for the `"mousemove"` event so that the pupils in the eyes follow the cursor position. For example, if you move the cursor to the lower right side of the screen, the pupils should shift so that they appear to be looking at that point, as seen in Figure 1b.

The tricky bit here can be figuring out the geometry to ensure that you move the pupils a small distance toward the cursor. There are a variety of ways to achieve this, and if you are feeling good about the math, have at it! If your geometry skills are feeling a bit rusty though, I'll provide some images and guidance that will hopefully help. For a given eye, you will know the position of the eyeball and of the mouse cursor, as I am showing below.



From that you can determine what the difference is in the x and y positions, which are denoted Δx and Δy in the image. If you just wanted to move the pupil to the location

of the mouse, you could just move it by those distances in both the x and y and it would work. But that is not how most people's eyeballs work! Instead, you want to move the pupil just a short fixed distance in that direction, which I'm labeling d in the image above. One way to figure out the x and y pieces of d would be to use trig, but alternatively, we can do it with ratios and similar triangles! So lets redraw what we have, but making the triangles more clear.



The fundamental idea behind similar triangles is that the ratios of their side lengths are the same. Thus, if we are comparing the vertical sides to the hypotenuses, we'd get something like

$$\frac{dy}{d} = \frac{\Delta y}{\sqrt{\Delta x^2 + \Delta y^2}}$$

and by comparing the horizontal sides to the hypotenuses:

$$\frac{dx}{d} = \frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}$$

Remember that you can calculate Δx and Δy and that d is a value you are setting, which is however far you want the pupil to move towards the mouse. Thus you can compute what dx and dy are, which are the distances you'll need to move the pupil away from the center of the eye. Note that you'll need to do these calculations within your callback function, as they need to be recomputed every single time the mouse is moved. And remember that you'll need to do the above twice, once for each eye.

Although the difference might appear slight when the cursor is outside the face, it is important to compute the position of each pupil independently for each eye. If you move the mouse in between the eyes, for instance, the pupils should point in opposite directions and make the face appear cross-eyed.