Name _____

CPADS Assignment V – Due 11/18

"Punkin' π and Let It Snow!"

So far we have covered the basics of creating functions and implementing (fixed) iteration. The next important programming concept is decisions, i.e. *conditional* execution. The most common structure for decisions is the `if-else` construct. In Python the syntax is:

```
if condition:
      true statements
elif condition2:
      true2 statements
      ...
else:
      false statements
```

Just as with functions and iteration, the body of each branch of the decision logic is indicated by *indentation*, otherwise any valid Python code may be used within the branch.

We can also combine iteration and decision structures to create *conditional* iteration (a `while` loop). For conditional iteration, the loop will execute *until* the condition statement becomes false. In Python, the syntax is:

```
while condition:
      statements to execute while condition is true
```

1. Estimating π

An approximation of π can be computed by generating random points in a square of some arbitrary size centered at the origin of the *x*-*y* plane. Each time one of the random points falls within a circle centered at the origin (where the circle completely fills the square), increase a counter. After some number of random points has been generated, divide the counter (how many random points were within the circle) by the total number of random points. This number will approximate the area of a circle of radius 0.5. We know that for a circle, the area *a* is computed by the equation

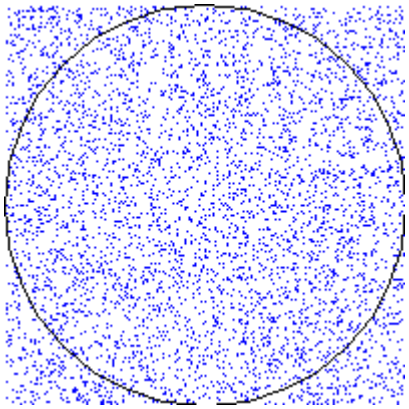
$$a = \pi r^2$$

Letting *r* equal 0.5, and re-arranging to solve for π,

$$\pi = a / 0.25 \quad \text{or} \quad \pi = 4a$$

Name _____

For example, here is a picture showing 5000 random points within a 200 by 200 square centered at the origin, along with an outline of a circle within the square:



Based on these 5000 points, we estimate π to be 3.0936. This is obviously not very accurate: computing more random points would lead to a more accurate estimate.

Your task is to complete the **while** loop in the **comp_pi** function such that:

1) The **while** loop runs while the difference between π and the approximate value of π being calculated in the **while** loop is greater than some threshold supplied by the user.
2) Keep a count of the total number of random points generated (***num_points***).
3) Calculate the distance of the random point (*x*,*y*) from the origin using the equation from the Pythagorean Theorem:

$$dist = \sqrt{x^2 + y^2}$$

4) Keep a separate count (***num_in***) of the points that fall within the circle of radius 1 (***dist*** ≤ 1).
5) Calculate a new approximation for π for each iteration of the loop:
   approx_pi = 4 x (points in circle / total points)
6) In **main()**, embed the input line that requests the number of digits within a **while** loop so that the user is required to enter a value for **digits** that falls within the specified prompt range.
7) In **main()**, embed the input line that requests the number of runs within a **while** loop so that the user is required to enter a value for **runs** that falls within the specified prompt range.

Name _____

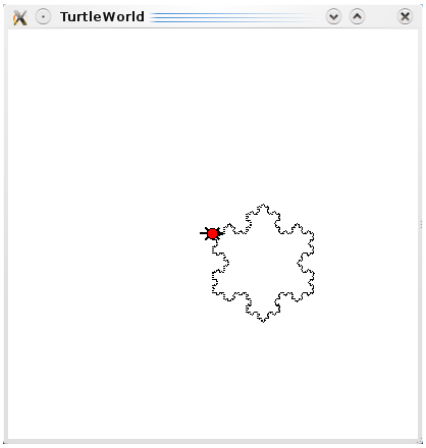
2. Snowflake

An interesting geometric figure known as a *Koch* curve can be drawn using a *recursive* function, i.e. a function that calls itself. One important consideration that must be observed when dealing with recursive functions is to ensure that they eventually reach a *termination* point, otherwise they will recurse indefinitely and eventually crash your program. To determine where to terminate, a recursive function always has some type of decision logic within the function. The recursive algorithm for drawing a Koch curve of length *L* is show below. Note the different drawing done in each branch of the conditional statement.

```
Koch(len):
      if len > 2:
            Draw a Koch of length L/3
            Turn left 60 degrees
            Draw a Koch of length L/3
            Turn right 120 degrees
            Draw a Koch of length L/3
            Turn left 60 degrees
            Draw a Koch of length L/3
      else:
            Draw a line of length L
```

- Complete the function named **`draw_Koch()`** that takes two parameters – **`t`** for the drawing turtle and **`length`** for the length of the curve.

- Extend your program by adding code to the **`draw_snowflake()`** function to draw a *snowflake* using 3 Koch curves, see the sample output below. Hint: Consider each Koch curve as the side of a triangle and use a *loop*.

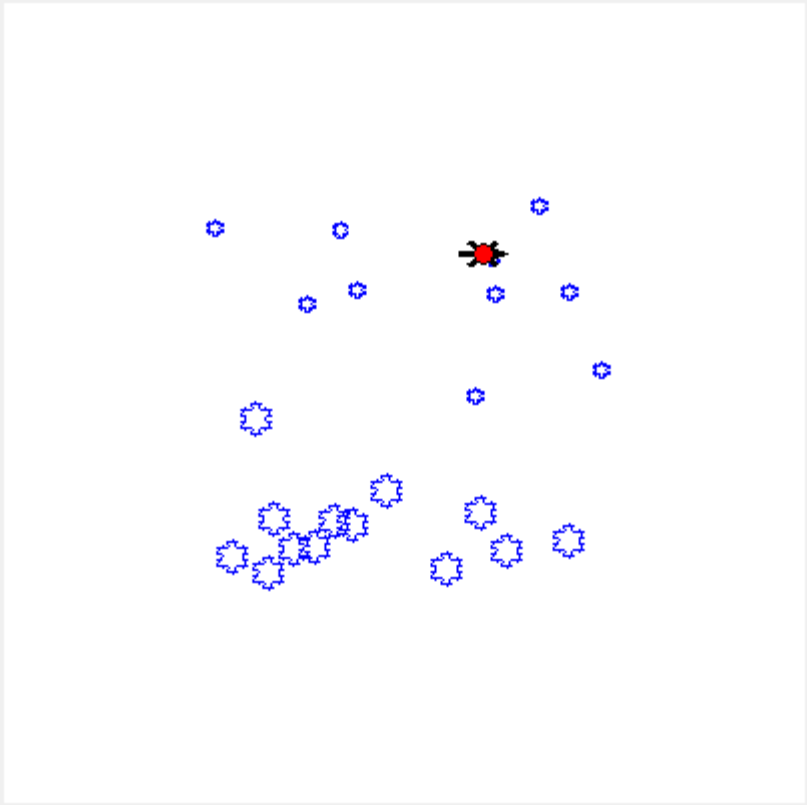


- Test your program.

Name _____

3. Blizzard

- Complete the `draw_blizzard()` function to use the parameters `num_top` for the number of snowflakes to draw in the top half of the screen, i.e. positive y values, and `num_bottom` for the number of snowflakes to draw in the bottom half of the screen, i.e. negative y values. The function should continue to generate randomly placed snowflakes **as long as** there are not at least `num_top` and `num_bottom` snowflakes in each half of the screen.
- Draw the snowflakes in the **top** half of the screen at half size
- Add code in `main()` to ensure that
  - The snowflake size is greater than 10
  - The number of snowflakes in the top half is between 0-10
  - The number of snowflakes in the bottom half is between 0-10



- When done, print out and attach a copy of your `blizzard.py` program to this activity. Additionally, submit your source file through Marmoset as **program04** (https://cs.ycp.edu/marmoset).