Assignment 7: Due Wednesday May 11 at 6pm

You must work either on your own or with one partner. If you work with a partner, you and your partner must first register as a group in CMS (this requires an invitation issued by one of you on CMS and the other of you accepting it on CMS) and then submit your work as a group. As mentioned in class, we strongly recommend against a “divide-and-conquer”/Henry-Ford-assembly-line approach: each member of a group should work on each part of each problem to get the full educational value out of the assignment.

You may discuss background issues and general solution strategies with others outside your CMS group, but the program(s) you submit must be the work of just you (and your partner). We assume that you are thoroughly familiar with the discussion of academic integrity that is on the course website. Any doubts that you have about “crossing the line” should be discussed with a member of the teaching staff before the deadline, and you should also document such situations as comments in the header of your submission file(s).

Topics. Recursion, Inheritance.

1 Recursive Flakes

In this problem you will implement a recursive graphics procedure that can draw what we will call an \( n \)-flake. Here are two 3-flake examples:

![3-flake examples]

1.1 Factorial

Review the April 19 lecture on recursion paying particular attention to the discussion about the factorial computation. From the “recursive” point of view here is how we think about the computation of \( 5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \):

- To compute \( 5! \) we compute \( 4! \) and multiply the answer by 5.
- To compute \( 4! \) we compute \( 3! \) and multiply the answer by 4.
- To compute \( 3! \) we compute \( 2! \) and multiply the answer by 3.
- To compute \( 2! \) we compute \( 1! \) and multiply the answer by 2.
- To compute \( 1! \) we compute \( 0! \) and multiply the answer by 1.
- To compute \( 0! \) we use the definition \( 0! = 1 \)

Extrapolating from this example we are led to the following recursive implementation of the factorial function \( f(n) = 1 \cdot 2 \cdot 3 \cdots n \):
def f(n):
    """ Returns n!
    PreC: n is a nonnegative int
    """
    if n==0:
        return 1
    else:
        return n*f(n-1)

The idea of defining $f(n)$ in terms of $f(n-1)$, shown above with the factorial function, is the key to the $n$-flake computation that you are required to develop.

1.2 $n$-Flakes

Here is a 3-flake:

To compute a 3-flake you compute a 2-flake and replace each side with a blipped side:
To compute a 2-flake you compute a 1-flake

and replace each side with a blipped side. To compute a 1-flake you compute a 0-flake

and replace each side with a blipped side. How can we carry this out in Python? And how do we “blip” a side?

1.3 Representing a Flake

A Flake is a polygon and polygons in the plane can be represented a number of ways:

**Option 1.** A list of floats \(x\) and a list of floats \(y\) can be used to house the vertex information. In this representation, the \(k\)-th vertex is \((x[k], y[k])\).

**Option 2.** A list of \(\text{Point}\) objects \(P\) can be used to house the vertex information. In this representation, the \(k\)-th point is \(P[k]\).

**Option 3.** A list of \(\text{LineSeg}\) objects \(L\) can be used to house the side information. In this representation, the \(k\)-th side is \(L[k]\).

We will go with Option 3 because it simplifies the discussion.

In \(A7.zip\) we provide you with modules that define a class \(\text{Point}\) and a class \(\text{LineSeg}\). (See \(\text{ThePointClass.py}\) and \(\text{TheLineSegClass.py}\)). Let’s take a look at what is in the \(\text{LineSeg}\) class:
class LineSeg(object):
    
    Attribute:
    
    P1: endpoint [Point]
    P2: endpoint [Point]
    
def __init__(self,P1,P2):
        self.P1 = P1
        self.P2 = P2

def Blipped(self,rightProb=0):
    
        """ Returns a length-4 list K of line segments that represent the blipped version of self. The items in K satisfy these properties:
        
        K[0].P1 and self.P1 represent the same point
        K[0].P2 and K[1].P1 represent the same point
        K[1].P2 and K[2].P1 represent the same point
        K[2].P2 and K[3].P1 represent the same point
        K[3].P2 and self.P2 represent the same point
        """

        PreC: rightProb is the probability that the "blip" is on the right as you "walk" from self.P1 to self.P2.

    
With this class we can represent a polygon as a list of LineSeg objects. For example, if Z0, Z1, Z2, and Z3 are Point objects, then

    L = [ LineSeg(Z0,Z1), LineSeg(Z1,Z2), LineSeg(Z2,Z3), LineSeg(Z3,Z0) ]

represents the quadrilateral obtained by connecting Z0 to Z1 to Z2 to Z3 to Z0. In general, a length-n list of LineSeg objects L represents a polygon if L[k].P2 represents the same point as L[(k+1)%n].P1 for k = 0,1,...,n - 1.

Let’s discuss the method Blipped. You do not have to understand the math behind the implementation. However, it is a good to have some idea of how it works. Suppose you have a line segment that connects two points P1 and P2, i.e., LineSeg(P1,P2). Think of the line segment as a road from P1 to P2 and that we now have to construct a “detour” around the middle third of the route. Instead of traveling directly from P1 to P2 directly, we travel from P1 to Q1 to T to Q2 to P2:

This four-leg route has the property that Q1, T and Q2 define an equilateral triangle. As drawn, the “blip” is on the left as you journey from P1 to P2. The code

    L = LineSeg(P1,P2)
    K = L.Blipped()

produces a length-4 list of line segments K. If we display the line segments that it encodes we would get the above figure. On the other hand,
\[ L = \text{LineSeg}(P_1, P_2) \]
\[ K = L.\text{Blipped}(\text{rightProb}=1) \]

puts the blip on the right side:

\[ L = \text{LineSeg}(P_1, P_2) \]
\[ K = L.\text{Blipped}(\text{rightProb}=0.3) \]

This puts the blip on the right side with probability 0.3.

1.4 MakeFlake

The module Flakes.py is all set to illustrate these ideas once you complete the implementation of

```
def MakeFlake(n, p=0):
    """ Returns a list of LineSeg objects L that represents
    an n-Flake with rightSide probability p. L has the
    property that L[k].P2 and L[(k+1)\%n].P1 represent the
    same point for k = 0, 1, ..., n-1 where n = len(L).\n    """

    "PreC: n is a nonnegative int, p is a float that satisfies 0<=p<=1.\n    ""
```

Your implementation must be recursive. The \( n = 0 \) base case should return a representation of the triangle with vertices

\[
P_0 = (0, 1) \\
P_1 = (\sqrt{3}/2, -1/2) \\
P_2 = (-\sqrt{3}/2, -1/2)
\]

Submit your finished version of Flakes.py to CMS.

2 Super Long Integer Arithmetic

COMING SOON