CS1110 Spring 2016

Assignment 7: Due Wednesday May 11 at 6pm
UPDATES on Monday May 9

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).*

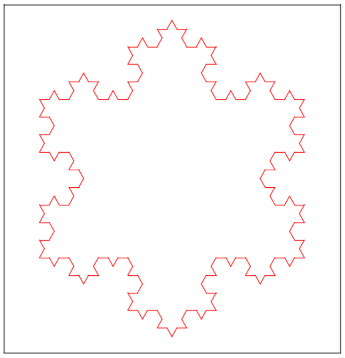
Warning: submit an early version before the regular deadline, and be clear on the slip day policies. Recent experiences suggest that we:

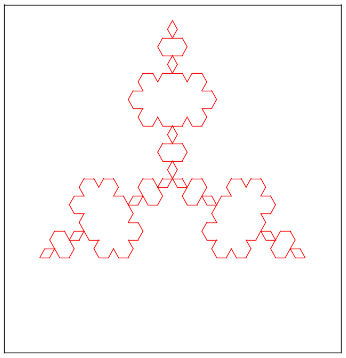
1. ...remind you that the specific slip day policies are posted on the Assignments page of the course website. *You can spend at most two slip days on an assignment; slip days are counted on the 24-hour marks.* See `http://www.cs.cornell.edu/courses/cs1110/2016sp/assignments/index.php` for more details.
2. ...advise you to *submit something to CMS well in advance of the deadline*, since you can always overwrite earlier submissions. For instance, even if you are planning to use two slip days, it would be wise to still submit what you have *before* the May 11th Wednesday 6pm deadline or on Thursday, to avoid the possibility that you somehow accidentally miss the (slip-day) deadline and, with nothing on CMS, get no credit for the assignment.

Topics. Recursion, method overloading, type-based displatch using `isinstance`.

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:





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 $0! = 1$

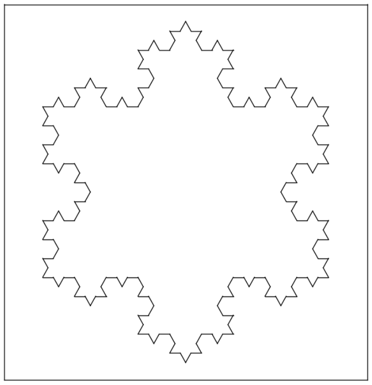
Extrapolating from this example we are led to the following recursive implementation of the factorial function $f(n) = 1 \cdot 2 \cdot 3 \cdots n$:

```
def f(n):
    """ Returns n!
    PreC: n is a nonnegative int
    """
    if n==0:
        return 1
    else:
        return n*f(n-1)
```

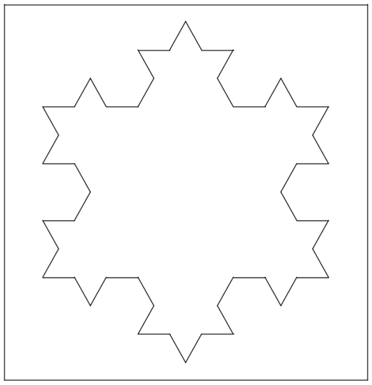
The idea of defining $f(n)$ in terms of $f(n-1)$, shown above with the factorial function, is the key to the n-flake computation that you are required to develop.

1.2 n-Flakes

Here is a 3-flake:

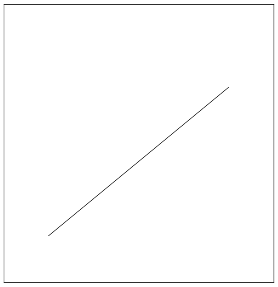


To compute a 3-flake you compute a 2-flake

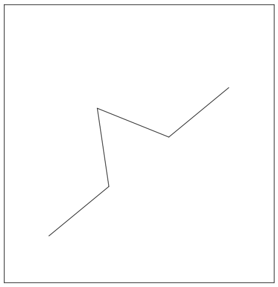


and replace each side with a blipped side:

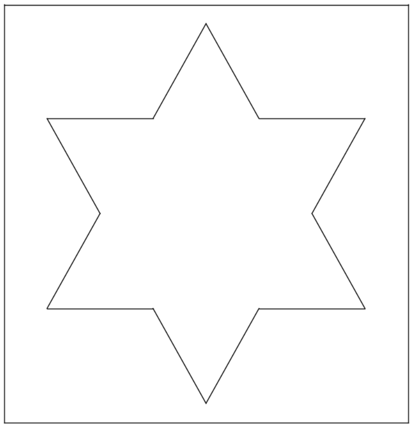
Side:



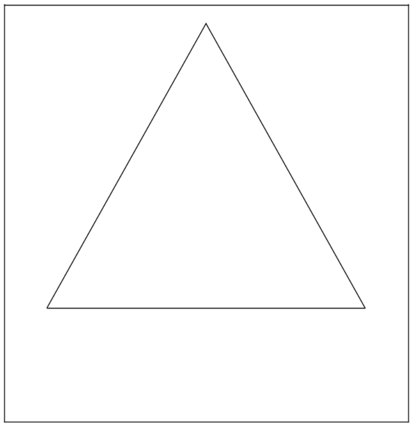
Blipped Side:



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 `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 `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 `Point` and a class `LineSeg`. (See `ThePointClass.py` and `TheLineSegClass.py`). Let's take a look at what is in the `LineSeg` class:

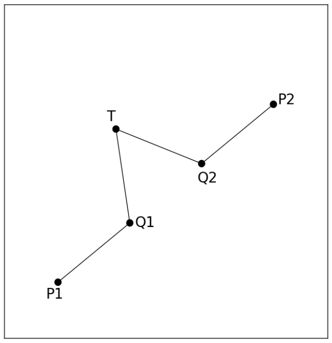
```
class  LineSeg(object):
    """
    Attribute:
         P1: endpoint [Point]
         P2: endpoint [Point]
    """
    def __init__(self,P1,P2):
        self.P1 = P1
        self.P2 = P2

    def Blipped(self,rightProb=0):
        """ Returns a length-4 list K of line segments that represents
        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 $k = 0, 1, \ldots, n-1$.

Let's discuss the method `Blipped`. You do not have to understand the math behind the implementation. However, it is 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 from `P1` to `P2` directly, we travel from `P1` to `Q1` to `T` to `Q2` to `P2`:



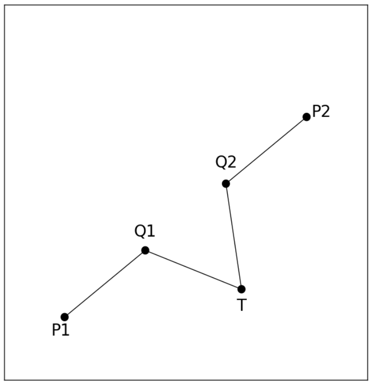
This four-leg route has the property that `Q1`, `T` and `Q2` define an equilateral triangle. As drawn, the "blip" is on the left as you journey from `P1` to `P2`. The code

```
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 = LineSeg(P1,P2)
K = L.Blipped(rightProb=1)
```

puts the blip on the right side:



It is also possible to randomly locate the blip:

```
L = LineSeg(P1,P2)
K = L.Blipped(rightProb=.3)
```

This puts the blip on the right side with probability .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).
```

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

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

$$P_0 \;=\; (0,1) \qquad P_1 \;=\; (\sqrt{3}/2, -1/2) \qquad P_2 \;=\; (-\sqrt{3}/2, -1/2)$$

Submit your finished version of `Flakes.py` to CMS.

2 Long Decimal Arithemtic

Download the module `TheLongIntClass.py` and the skeleton module `TheLongDecimalClass.py`. You will use the former as you develop the latter.

2.1 The Type `long`

Let's do some computations with big integers:

```
>>> 2**29
536870912
>>> 2**30
1073741824
>>> 2**31
2147483648L
```

Hey, what's with the "L" that shows up when we compute 2^{31}? The reason has to do with the fact that most computers are set up to do integer arithmetic with hardware that uses 32 bits to represent integers. One bit is used for the sign and that leaves 31 bits for the "number part." The largest possible `int` value is $2^{31} - 1 = 2147483647$. When an integer computation generates an `int` that is too big to store, Python converts the value to a type called `long`. If this happens in interactive mode, then Python uses the "L" to tell you about it:

```
>>> x = int(2**31 -1)
>>> x
2147483647
>>> y = x+1
>>> y
2147483648L
```

This is a handy feature that permits the computation of very big integers:

```
>>> 2**1000
107150860718626732094842504906000181056140481170553360744375038837
518714528569231404359845775746985748039345677748242309854210746050
4607706291457119647768654216766042983165262438683720566806937 6L
```

To implement `long` arithmetic, the Python designers had to write software–they could not delegate things like integer addition and multiplication to the hardware. This takes us back to the world of place value and carries.

2.2 The Class `LongInt`

To get an idea of what "hand coded" integer arithmetic looks like we have implemented a class `LongInt` that can be used to perform addition and multiplication on arbitrarily long nonnegative integers. (The secret is to use strings to hold digits, since strings can be really, really long.) The docstrings for its attributes and methods are below.

```
class LongInt(object):
    """
    Attributes:
        string: a string composed of the digits for this LongInt
    """
    def __init__(self,x):
        """
        x is a string of digits, or a nonnegative integer, or
        a reference to a LongInt. (The last possibility is useful for
        making copies.)
        """

    def getString(self):
        """Returns the String underlying this LongInt."""

    def __str__(self):
        """ For pretty printing. To display a LongInt object x
        just write  print x.
        """

    def __add__(self,other):
        """ Returns the sum of self and other as a LongInt.
        PreC: other is a reference to a LongInt object.
        """

    def DigitMult(self,d):
        """ Returns a LongInt whose value is int(d) times the
        value of self.
        PreC: d is a length-1 string that is a digit.
        """

    def times10(self,k):
        """Modifies self so that it represents an integer
        that is 10**k times bigger.
        PreC: k is a nonnegative int
        """

    def __mul__(self,other):
        """ Returns the product of self and other as a LongInt.
        PreC: other is a reference to a LongInt object.
        """
```

For details, browse through (and run and play with) the given module `TheLongIntClass.py`.

2.3 The Class `LongDecimal`

In this problem you are to develop a class `LongDecimal` that supports addition and multiplication on *long decimals*. These are numbers like `392898492098408208404.33787493`.

Below are some examples that a successful implementation can handle. (For the sake of brevity, we've chosen "little" numbers that don't really require using our new class to represent. You should imagine that the real use of these classes would be for numbers with, say, a 100 digits.) The first shows how to multiply one `LongDecimal` by another:

```
>>> x = LongDecimal(1234.56789)
>>> y = LongDecimal('398948937.3389493374')
>>> z = x*y
>>> print z
492529547788.288898290816086
```

It also shows that the constructor can accept either a float or a string. The next example shows how we can add one `LongDecimal` to another:

```
>>> x = LongDecimal(111.1111)
>>> y = LongDecimal(22.22)
>>> z = x+y
>>> print z
133.3311
```

We can also add a `LongDecimal` to a `LongInt`:

```
>>> x = LongDecimal(12345.56789)
>>> y = LongInt(1000)
>>> z = x+y
>>> print z
13345.56789
```

(The `LongInt` must be to the right of the “+”.) We can also multiply a `LongDecimal` by a `LongInt`:

```
>>> x = LongDecimal('123456789.123456789')
>>> y = LongInt(1000)
>>> z = x*y
>>> print z
123456789123.456789
```

Update May 9: corrected '123456789123.4567890000' to 123456789123.456789

(The `LongInt` must be to the right of the “*”.)

The given module `TheLongDecimal.py` contains a skeleton of the class `LongDecimal` that you are to implement. Here are some of its essential features: Note the UPDATE portions in the skeleton below; the rationale for these changes are given in section 2.3.1.

```
class LongDecimal(object):
    """
    Attributes:
        Whole: the whole number part  [LongInt]
        Fract: the fraction part      [LongInt]

        UPDATE (Mon May 9): Fract's String should always have length at least 1,
        and should contain no extra trailing zeroes.  For example, '0' is allowed;
        '', None, '00', and '04000' are not allowed.
    """
    def __init__(self,x):
        """
        PreC: x can be either
           a string that is composed of digits and at most one decimal point.
        or
           a valid nonnegative int or float literal, i.e., 37 , .37 , 37.
        or
           a reference to a LongDecimal object.

        UPDATE (Mon May 9): for the purposes of this assignment, assume input is not
        in scientific notation.

        """
    def __str__(self):
        """For pretty printing."""

    def __add__(self,other):
        """ Returns the sum of self and other as a LongDecimal
        PreC: other is a reference to a LongDecimal or a LongInt
        """

    def __mul__(self,other):
        """ Returns the product of self and other as a LongDecimal
        PreC: other is a reference to a LongDecimal or a LongInt
        """

    # UPDATE (Mon May 9) added this helper function for students to use.
    # helper method.
    def fix_trailing(self):
        """Remove trailing zeroes from the (String of) Fract of LongDecimal ld.
        If removing all the trailing zeroes would leave an empty string,
        set the Frac String to "0".
        Returns self (as a convenience to the caller).
        PreC: self.Fract.String is a string. """
```

2.3.1 UPDATES on representation: zeroes and scientific notation

We would like to have a unique representation for each number, and thus impose the following policy for zeroes in the `String` for the `Fract` of a `LongDecimal`:

> A Fract’s String should always have length at least 1, and should contain no extra trailing zeroes.
> For example, ’0’ is allowed; ”, None, ’00’ and ’04000’ are not allowed.

To make this update easier for you to manage, we’ve added a helper method for you in the `TheLongDecimalClass.py` skeleton on the web:

```
# helper method
def fix_trailing(self):
    """Remove trailing zeroes from Fract of LongDecimal ld.
    If removing all the trailing zeroes would leave an empty string,
    set the Frac String to "0".
    Returns self (as a convenience to the caller).
    PreC: self.Fract.String is a string."""
    self.Fract.String = self.Fract.String.rstrip(’0’)
    if len(self.Fract.String) == 0:
        self.Fract.String == "0"
    return self
```

Also, to save you time on this assignment, you are not responsible for handling inputs to the LongDecimal constructor that are in scientific notation.

2.3.2 Hints and grading

In grading your implementation of the class `LongDecimal` we will look at (a) correctness, (b) how well you took advantage of the class `LongInt`, and (c) code cleanliness: redundant or long portions within methods should be replaced by (specified) helper methods. Submit your finished version of `TheLongDecimal.py` to CMS. You will lose points if the submitted module contains anything other than the class `LongDecimal`.

Here are two hints that suggest how you might exploit the addition and multiplication methods in `LongInt`:

```
1234.56 + 29.2983        is related to       12345600 + 292983

123.456 * 7362.78033     is related to       123456 * 736278033
```

We are leaving you to decide how to test your code, but here’s a hint There are (at least) four classes of things around that you’re going to need to deal with: `LongDecimals`, `LongInts`, strings, and floats. Be aware of what you should be working with at any given time; we expect using print statements of the form `print type(x)` should be helpful.

3 Reminders about submission and CMS:

1. If you meant to be grouped, verify ahead of time that CMS group invitations were issued and accepted before submitting.
2. Make sure your submitted `.py` files begin with header comments listing:
    (a) The name of the file
    (b) The name(s) and netid(s) of the person or group submitting the file
    (c) The date the file was finished

If there were other people who contributed to your thought processes in developing your code, mention them by name in header comments. (You don't have to mention course-staff members, although it makes sense to do so.)

3. Make sure all your functions have appropriate docstrings. These should include explanations of what the parameter variables "mean" and what preconditions (constraints) are assumed on their values, and what the user can expect as a result of calling your functions.
4. If you submit earlier than two hours before the submission deadline, CMS allows you to download your files to check that they are what you think you submitted. We thus strongly suggest that you submit a version early and do this check. (If you want to make changes, no problem: CMS allows you to overwrite an older version by uploading a new version.)
5. (We have heard that the following issue has been fixed, but better to be safe than sorry...) Do not "reload"/"refresh" the CMS assignment submission page after the submission deadline passes: this will trigger another upload of your files, which can result in your submission being counted as late. The safest policy is to close your browser tab/window after upload.