# COM110: Lab 10
# Recursive drawings, fractals

1) Recursion can be used to make interesting drawings. In order to set the stage for drawing recursively, we will make use of something called **Turtle graphics**. Picture a "turtle" who only knows three things:
   - how to move forward a certain distance while leaving a trail (i.e. "*draw*ing"),
   - how to change direction (i.e. "*turn*ing"), and
   - how to *move to* a specified point (as though someone picked the turtle up and placed it down at the specified point without changing the direction it's facing).

   a) Open up turtle.py to see these commands realized as a Python class called `Turtle`. (Don't worry about the code in the body of the methods, but read all the method signatures and their documentation and comments.) In the main program, try out a few turtle commands to see how the "turtle" draws and moves. A `Drawing` window with a `Turtle` object has already been created for you. Notice the `Turtle` object has been moved to the left side of the graphical window, halfway up. Also note that when you `turn` the turtle, the degrees of rotation must be given in *radians*. So to `turn` the turtle **90 degrees to the *left***, you must `turn` by **`pi/2`** radians (since `pi` is 180 degrees). And to `turn` **60 degrees to the *right***, you must `turn` by **`-pi/3`** radians. (Positive radians means turn the turtle to the *left* (counterclockwise) and negative radians means turn the turtle to the *right* (clockwise).)

   b) Practice using the turtle commands (`turn()`, `moveTo()`, and `draw()`) by drawing your initials in the graphical window.

**☺ Get check 1 ☺**

2) Now let's create a **drawing using recursion**. Recall that any recursive function always needs a *base case* under which the recursion stops. One way to implement a base case is to imagine "levels" of recursion. Each time we make a recursive call, we reduce the "level" by 1, and we stop making recursive calls when the "level" reaches 0, which will be our base case. Let's create a square spiral by drawing a certain **length** in one direction, turning left 90 degrees and then calling the function recursively with a *smaller* **length** and with *"*level" reduced by one. Complete the turtle.py `spiral()` method by filling in the code for the recursive case. In the recursive case:

   a) `draw` for the given `length`,
   b) then `turn` 90 degrees to the left,
   c) then recursively call a "smaller" instance of `spiral()`: call it with *reduced* parameter values... specifically, call spiral with a *reduced* `length` (scaled down by, say, 14/15 from the original length) and a *reduced* `level` (level - 1).

   Test your `spiral()` function by calling it from the `main` function.

**Answer:**

```
if level==0:
        turtle.draw(length)

else:

        turtle.draw(length)
        turtle.turn(pi/2)
        spiral(turtle,14*length/15,level-1)
```

**☺ Get check 2 ☺**

3) **Fractals** are geometric objects that exhibit *self-similarity* and "infinite complexity" as you "zoom-in" on them closer and closer. It was not possible to generate even partial pictures of fractals until we could harness the power and speed of computers. The Mandelbrot set is one example of a fractal: http://en.wikipedia.org/wiki/Fractal.
4) The above link shows a series of images of the same picture at progressively higher zoom levels. You can see from this sequence of images that as you zoom in to the center point of the picture further and further, you eventually *again* find *the original Mandelbrot set*!
5) Just as we've learned that a recursive function keeps invoking smaller and smaller instances of itself, fractals also exhibit this kind of repeated and self-similar behavior. Indeed, using recursion is a common way to generate images of fractals. Take a look further down the wikipedia page on fractals to see other examples of famous fractals. Look especially at the animation/description of the Koch snowflake. Here is another page about the Koch snowflake: http://mathworld.wolfram.com/KochSnowflake.html.
6) You can also learn about the Koch snowflake by reading exercise 8 on page 464 of Zelle. Creating this snowflake will be our next task.
   a) At the upper-most level of recursion, as you can see from the above websites, the Koch snowflake is just a triangle. In the `main` function, create an equilateral triangle. (Hint: at each corner you'll need to turn 120 degrees, which is the exterior angle of each angle in the equilateral triangle. In radians, 120 degrees is 2*pi/3. *The below instructions for check 4 assume that you draw this triangle in the clockwise direction*. E.g., start at (200,250), `draw` straight across a certain distance, `turn` 120 degrees to the right, `draw` again the same distance, `turn` 120 degrees to the right again, and `draw` the same distance again.)

      ```
      turtle.moveTo(Point(200,250))
      turtle.draw(200)
      turtle.turn(-2*pi/3)
      turtle.draw(200)
      turtle.turn(-2*pi/3)
      turtle.draw(200)
      ```

   b) Now, for each side of the triangle, rather than calling the `draw()` function, call the `kcurve()` function with a recursion level of 0. (Notice this makes use of the base case that has already been coded for you, which simply is a call to the `draw()` function. It

does not require that you complete the rest of kcurve() yet, and it just creates the same triangle as in part a.)

**Answer:**

```
turtle.moveTo(Point(200,250))
kcurve(turtle,200,0)
turtle.turn(-2*pi/3)
kcurve(turtle, 200, 0)
turtle.turn(-2*pi/3)
kcurve(turtle, 200, 0)
```

**☺ Get check 3 ☺**

c) Finally, we must complete the recursive case of the `kcurve()` function. Think of the turtle standing on one corner of the triangle, facing the next corner in the clockwise direction. To draw a Koch curve between these two corners, take the following steps.

i) You need to reduce `length` to 1/3 of the old length and reduce the `level` by 1.
ii) Call `kcurve()` with this new length and reduced level.
iii) By doing this you've reached 1/3 of the way across the side of the triangle so it's time to form that "equilateral bump" that wikipedia and our book talked about.
iv) How many degrees are in each inner angle of an equilateral triangle? Turn to the left that many degrees and then draw your new spur (by calling `kcurve()` again).
v) You are now out on the tip of your "equilateral bump" so you must make your way back to the side of the triangle. Turn right 120 degrees (the complement of the inner angle of the triangle), and create a spur going back (by calling `kcurve()` again).
vi) Finally, `turn` left to finish the final third of the side of the triangle, again calling `kcurve()` so that you arrive at the destination corner!

Once you've done all this, when you increase the `level` of recursion in your calls to `kcurve()` in the main function, you should see the Koch snowflake appear! (Try recursion levels of 2 and 3.)

```
if level==0:
    turtle.draw(length)
else:
    kcurve(turtle,length/3,level-1)
    turtle.turn(pi/3)
    kcurve(turtle, length/3, level-1)
    turtle.turn(-2*pi/3)
    kcurve(turtle,length/3,level-1)
    turtle.turn(pi/3)
    kcurve(turtle,length/3,level-1)
```

**☺ Get check 4 ☺**

7) Koch Snowflake follow up:
   a) Try setting each side of the triangle to a different level of recursion.

b) Try changing all right turns to left turns and vice versa. What happens to your snowflake? Can you explain what's going on?

**☺ Get check 5 ☺**

**Bonuses**

A. Complete programming exercise 4 on page 497 (463 in second edition) of Zelle. (Write the max() function recursively.)

B. Complete programming exercise 9 on page 500 (466 in second edition) of Zelle. (More fractal fun!)

C. **Only if you've already completed bonus A and B above, you may also now take a shot at any bonuses from previous labs.**

D. **You may also work on current assignment (assignment 5)**

**☺ Get bonus checks ☺**