CS 1 Honor Roll: Introduction To Computer Programming, Fall 2013

Assignment 2h: Recursion: See "Recursion"

Due: *Friday, December 6, 02:00:00*

What to hand in

Put all of your files for this assignment into a single zip file, and submit that as `lab2h.zip`. This will include separate files for each of the problems, plus whatever test scripts or other files you want to include.

Submit your files to csman, to the CS 1 Honor Roll "course".

This lab will deal with recursion. You'll some non-graphical and then some graphical functions to help you understand how recursion works.

General advice: solving recursive problems

A recursive problem is defined in terms of itself. For this to make sense, it must be defined in terms of a smaller version of itself. For example, a function which computes the Nth fibonacci number (from the fibonacci series 0, 1, 1, 2, 3, 5, 8, 13, 21, ...), can be described recursively as:

```
fib(0) = 0
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```

This is a recursive definition. The function `fib` is defined in terms of itself. Notice, though, that two of the three cases are non-recursive; cases that can be solved without recursion are referred to as **base cases**. When writing a definition for a recursive function, it is nearly always a good idea to solve the base cases first (in an `if/else` statement, for instance) before solving the recursive case(s). Also, the recursive case calls the `fib` function, but with *smaller values for the arguments*. This is essential, or the recursion will never terminate. After the recursive calls return, their results are combined to give the final result.

Writing recursive functions can be very tricky even for otherwise quite competent programmers. It's a bit like riding a bike; once you've done it, it's no big deal, but until then it may seem extremely hard. If you find these problems frustratingly difficult, don't spend insane amounts of time on them and don't feel bad; instead, you should take CS 4 in the Winter term, which covers recursive problem solving in great detail. After that, you'll be an expert.

Basic exercises involving recursion

Write the following functions using recursion instead of *e.g.* a loop. Make sure you identify what the base case or cases are and handle them first (a base case is one that can be solved without using recursion). Therefore, your functions will all contain an `if`/`else` statement at the beginning.

As usual, don't forget to write docstrings for all of your functions.

1. Write a function called `countdown` which takes one argument (a positive integer `n`), and prints each number from `n` down to (but not including) zero, with one number per line. For instance:

   ```
   >>> countdown(5)
   5
   4
   3
   2
   1
   ```

   (This is really easy to do with a `for` or `while` loop, but don't do it that way! The solution using recursion is also very short.)

2. Write the `fib` function discussed above. Note that this function is easy to write imperatively; we want you to write it in the most naive recursive way possible (just translate the definition into Python).

   Examples:

   ```
   >>> fib(0)
   0
   >>> fib(1)
   1
   >>> fib(2)
   1
   >>> fib(3)
   2
   >>> fib(4)
   3
   >>> fib(5)
   5
   # etc.
   ```

   Try some larger numbers (say, `fib(40)`). Does it take a long time to execute? It turns out that the most straightforward way to write a recursive function to compute fibonacci numbers is extremely inefficient (that's OK here; we want you to do it that way). There is a whole field of computer science called *computational complexity* that deals with how to compute the efficiency of functions. This topic will be covered (at a basic level) in CS 4, and then in much more detail in CS 21 and CS 38.

Recursion and memoization

The `fib` example shown above is extremely inefficient because the same quantities are recomputed over and over again. For instance, consider `fib(6)`. This would give rise to these computations:

```
fib(6)
```

```
fib(5), fib(4)
  fib(4), fib(3), fib(3), fib(2)
    fib(3), fib(2), fib(2), fib(1), fib(2), fib(1), fib(1), fib(0)
      fib(2), fib(1), fib(1), fib(0), fib(1), fib(0), fib(1), fib(0)
        fib(1), fib(0)
```

So in order to compute `fib(6)`, we must compute:

```
fib(5)  --> 1 time
fib(4)  --> 2 times
fib(3)  --> 3 times
fib(2)  --> 5 times
fib(1)  --> 8 times
fib(0)  --> 5 times
```

This is a tremendous amount of recomputation, and it only gets worse for larger inputs to the `fib` function. This may lead you to think that recursion is for the birds, and in fact for this problem it's easy to solve it non-recursively and efficiently (note that solving it non-recursively doesn't guarantee efficiency!). For many problems, though, non-recursive solutions are much more complicated to write than recursive ones. It would be nice if we had a way to make the recursive function more efficient.

We can do this using a technique called **memoization**. This is an incredibly useful programming trick which is also quite simple. The idea is this: you want to compute a function for some input value N. You have been using this function for a while, and have kept track of which values of N the function has been called on, along with the results for those values of N. The values are stored in a table. So if (for instance) `fib` was called with a value of 4, and it had previously been called with that same value, instead of recomputing `fib(4)` from scratch, just look it up in the table and return it! Only if the number is not present in the table do we compute it, and after we've computed it, we store it into the table so we don't have to compute it again.

Write a memoized version of the `fib` function called `fib_memo`. Use a global dictionary called `fib_memo_values` to store the previously-computed values of `fib_memo`. When `fib_memo` is called, first look to see if the number is in the dictionary, and if it is, return the corresponding value. Otherwise, compute it and store it back into the dictionary. Then check the performance of `fib_memo` compared to `fib` on numbers between 40 and 50. You can also use much larger numbers with `fib_memo` (*e.g.* `fib_memo(1000)`) but please don't try this with regular `fib`, or your computer may grind to a halt and major sadness will ensue!

Harder problems involving recursion

Here are a couple of harder problems that can be solved using recursion.

NOTE: Although it may be very tempting, we don't want you looking up solutions on the internet for this section! Honor code rules apply. Write your own code. Hints are included below.

1. Write a function called `perms` which takes a single argument, which is a list, and returns a list of all the permutations of the original list. You may assume that the list elements are unique. (If you're really good, make it work even for lists that have repeated elements!)

   Examples:

   ```
   >>> perms([])
   [[]]
   >>> perms([1])
   [[1]]
   >>> perms([1, 2])
   [[1,2],[2,1]]
   >>> perms([1,2,3])
   [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
   ```

   Note that the order of the permutations in the output list is not important (any order will do).

   Hints: You know how to handle the base case. For the recursive case, first recursively compute the permutations of a slice of the original list which consists of all of the original list except for one element (say, the first element). Then figure out how to combine these sub-permutations with the missing element into the final list of permutations. A helper function may be very useful.

2. Write a function called `n_queens` which will solve the N-queens problem. This problem is to place N queens on an NxN chessboard so that none of the queens can capture the other queens on the next move. In other words, place the queens so that they are not on the same row, column or diagonal with any other queen.

   There are a number of ways you might attempt to solve this problem. You could go brute-force and generate all possible locations of N queens on the chessboard and then filter out the ones you don't want, but that would be very inefficient. Also, the way you represent a solution is important. We want you to represent solutions in a particular way and solve the problem recursively as we'll describe.

   To represent an NxN chessboard with queens on it, use a list of integers between 1 and N of length exactly N. Each element of the list represents the row index of a queen. The column index is the list index of the element plus one. So if you have (for a 4x4 chessboard) `[2, 4, 1, 3]`, that means that in column 1, the queen is at row 2; in column 2, it's at row 4; in column 3, it's at row 1, and in column 4, it's at row 3. This representation is nice in that it's compact and it automatically enforces one of the invariants: that the different queens aren't in the same column. You have to generate lists of row indices where the rows aren't duplicated and the queens also don't share the same diagonals.

   Examples:

   ```
   >>> n_queens(4)
   [[2, 4, 1, 3], [3, 1, 4, 2]]
   >>> n_queens(5)
   [[1, 3, 5, 2, 4], [1, 4, 2, 5, 3], [2, 4, 1, 3, 5], [2, 5, 3, 1, 4], [3, 1, 4, 2, 5], [3, 5, 2, 4, 1], [4, 1, 3, 5, 2], [4, 2, 5, 3, 1], [5, 2, 4, 1, 3], [5, 3, 1, 4, 2]]
   >>> n_queens(8)[:4]
   [[1, 5, 8, 6, 3, 7, 2, 4], [1, 6, 8, 3, 7, 4, 2, 5], [1, 7, 4, 6, 8, 2, 5, 3], [1, 7, 5, 8, 2, 4, 6, 3]]
   >>> len(n_queens(8))
   92
   >>> len(n_queens(12))
   14200
   ```

   Here are two ways to solve this problem. One uses recursion directly, the other indirectly.

   1. Define a function called `n_queens_columns_to` which generates a list of solutions to the N-queens problem, but only up to a particular column. This is a recursive function. There is a base case for either the first column or the zeroth column (the latter is a bit cleaner). For later columns you use a recursive call to the function to get the previous columns, then you figure out which indices you could append to the end of each partial solution to get a partial solution which includes one more column. Once you've done all the columns, you have all your (complete) solutions. Call this function from `n_queens`.

2. The previous solution is pretty efficient if you write it correctly. But if you're really lazy and don't care about efficiency, you can use the `perms` function you just defined to give you a first cut on solutions. If you have all permutations of integers between 1 and N, you have all solutions to the N-queens problems, but also many non-solutions. The row and column constraints are automatically enforced, but you have to filter out pseudo-solutions where queens share the same diagonal. Once you do that, you have all solutions. This approach doesn't involve any recursion except indirectly through the `perms` function, which you've already defined.

For full credit, write functions which solve the N-queens problems using both of these approaches (obviously, different functions for each approach). You can write as many helper functions as you like; in fact, this is good design.

Recursion and graphics

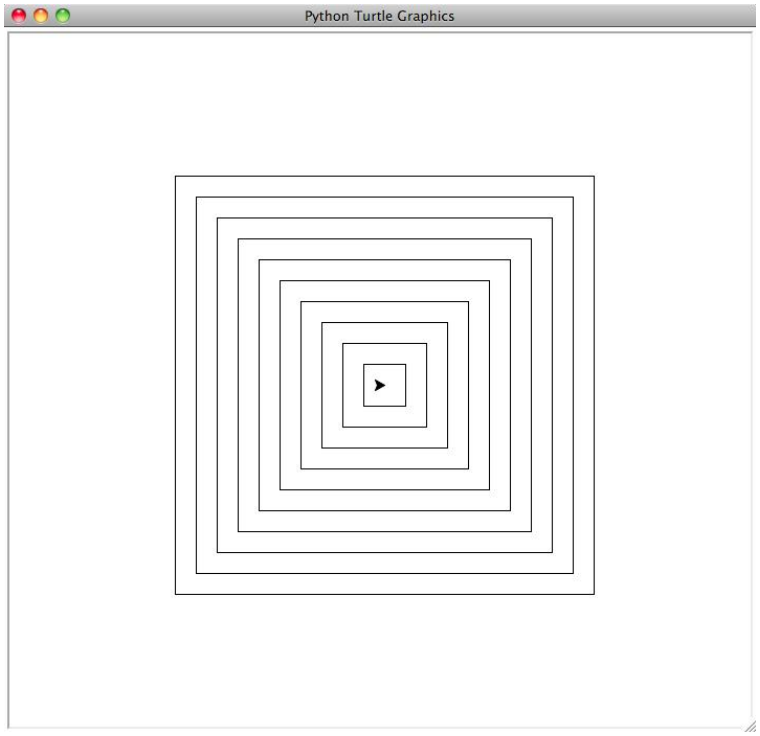
Now we're going to combine what you know about recursion with turtle graphics to create interesing pictures.

Before you begin, please look at the Python turtle graphics module documentation for information about how turtle graphics works in Python (in case you've forgotten).

As usual, don't forget to write docstrings for all of your functions.

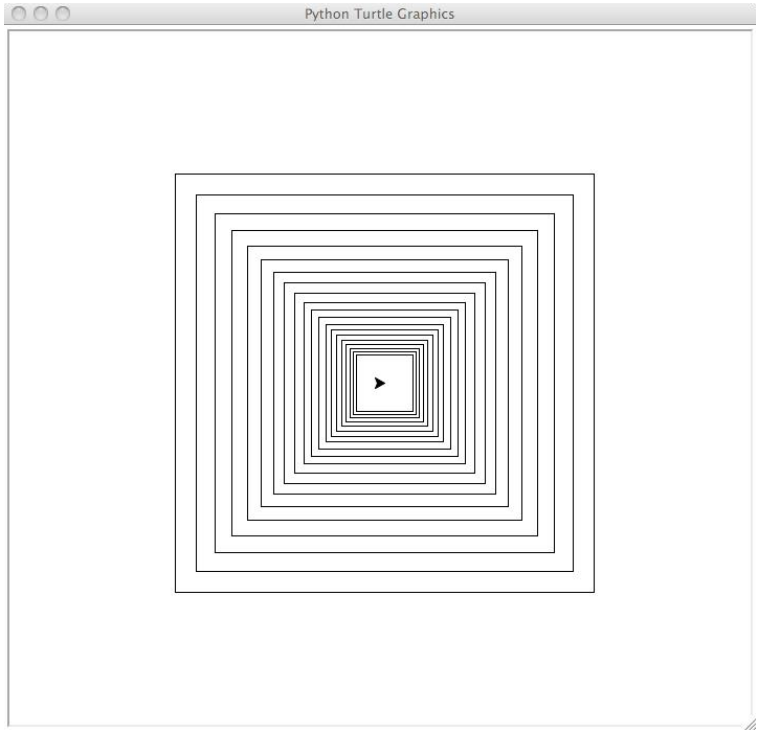
1. Write a function called `draw_nested_squares_1` which takes three arguments: a positive integer `n` which represents the number of nested squares to draw, a number called `size` which represents the size of the sides of the first square drawn (in pixels), and a number called `d` (for "decrement") which represents how much to decrease the size of the squares each time you draw one. The squares should all be centered on the initial location of the turtle, and the turtle should return to that location after drawing each square. Do not use a `for` or a `while` loop in this problem; instead, use recursion as follows. If `n` is zero or the size is less than zero, just return without drawing anything. Otherwise, draw a square and recursively call the function with smaller `n` and `size` values.

   For instance, the function call `draw_nested_squares_1(10, 400, 40)` will give an image that looks like this:



2. Write a function called `draw_nested_squares_2` which takes three arguments: a positive integer `n` which represents the number of nested squares to draw, a number called `size` which represents the size of the sides of the first square drawn (in pixels), and a number called `m` (for "multiplier") which represents how much to multiply the size of the squares each time you draw one. `m` should be in the range (0.0, 1.0) so that the squares will shrink each time. Use an `assert` statement to check this. The squares should all be centered on the initial location of the turtle, and the turtle should return to that location after drawing each square. Again, do not use a `for` or a `while` loop in this problem; instead, use recursion as in the previous problem.

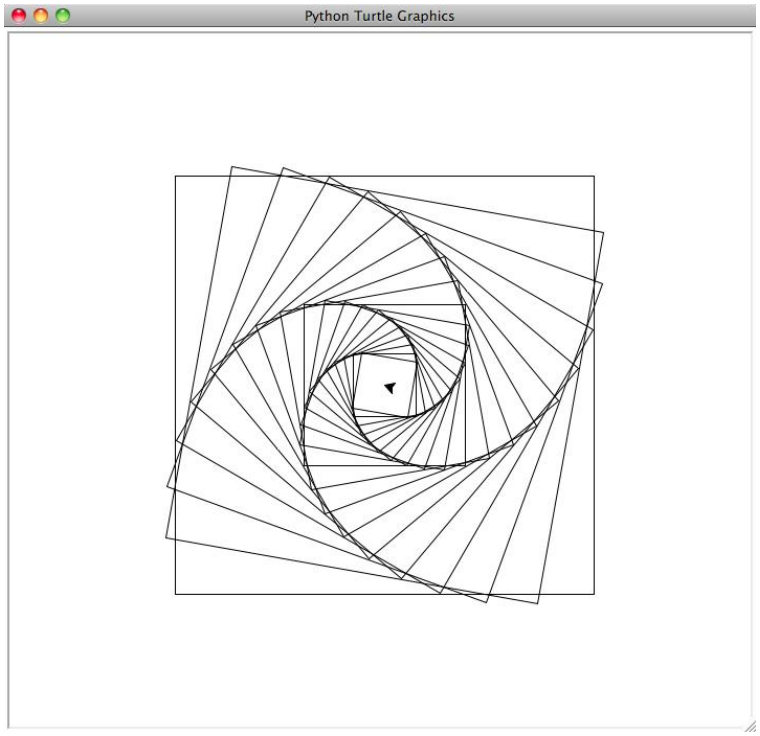
   For instance, the function call `draw_nested_squares_2(20, 400, 0.9)` will give an image that looks like this:



Notice the difference relative to the previous picture; this one looks like you're looking down a tunnel.

3. Now for our final nested squares example -- the best of all! Write a function called `draw_nested_squares_3` which takes *four* arguments: the number of squares and size as before, the multiplier as in the last problem, and a positive integer which represents the number of degrees to rotate each square relative to the previous square drawn. Again, use recursion has you have been doing previously. Again, use `assert` statements to check that the multiplier is in the right range and that the rotation is between 0 and 360 degrees.

   For instance, the function call `draw_nested_squares_3(20, 400, 0.9, 10)` will give an image that looks like this:



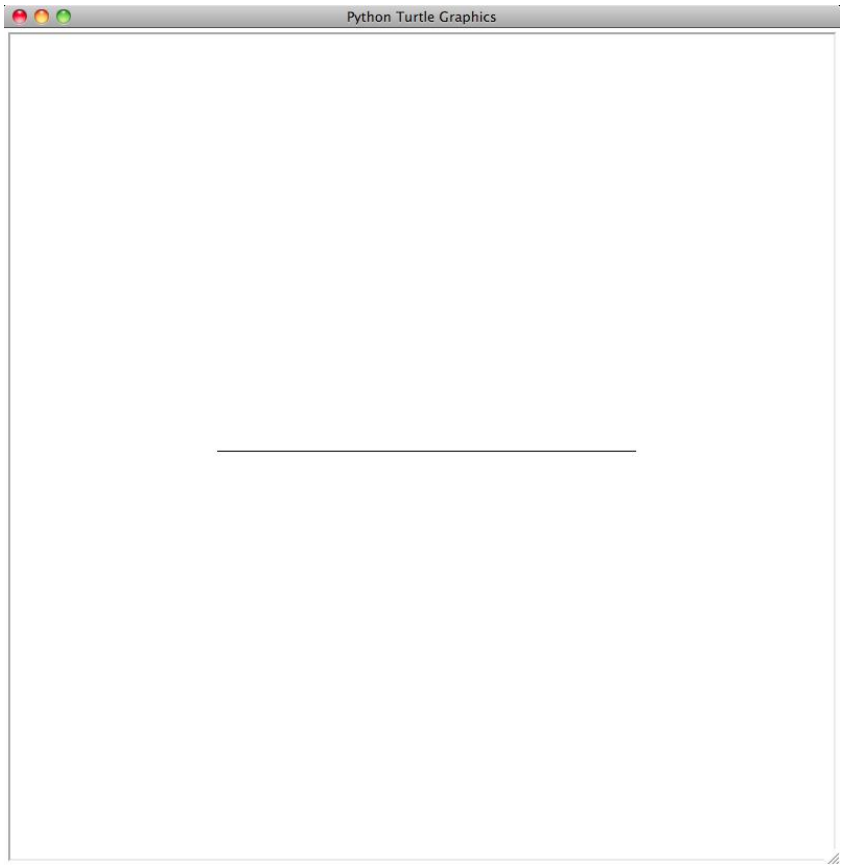
Cool, huh? Experiment with smaller and larger multipliers, and different numbers of degree rotations.

More complex graphics

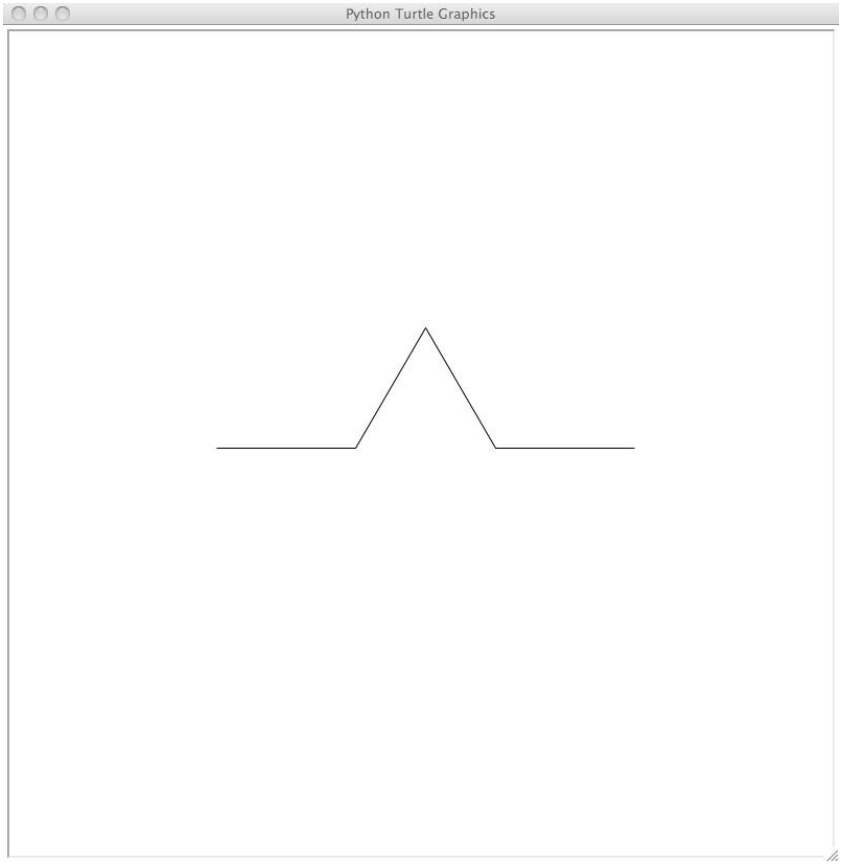
In lab 3, you used L-systems to create images of the Hilbert curve, the Koch snowflake and the Sierpinski triangle. You can also do this directly, using turtle graphics and recursive functions. That's what you'll do in this section.

The Koch snowflake

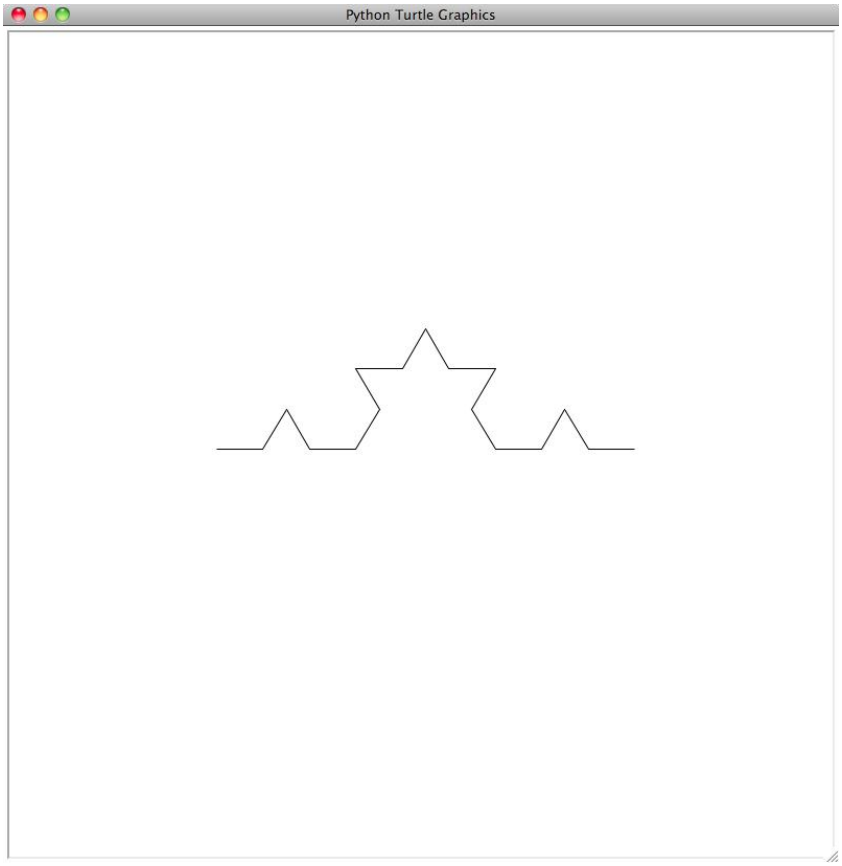
First, consider a straight line:



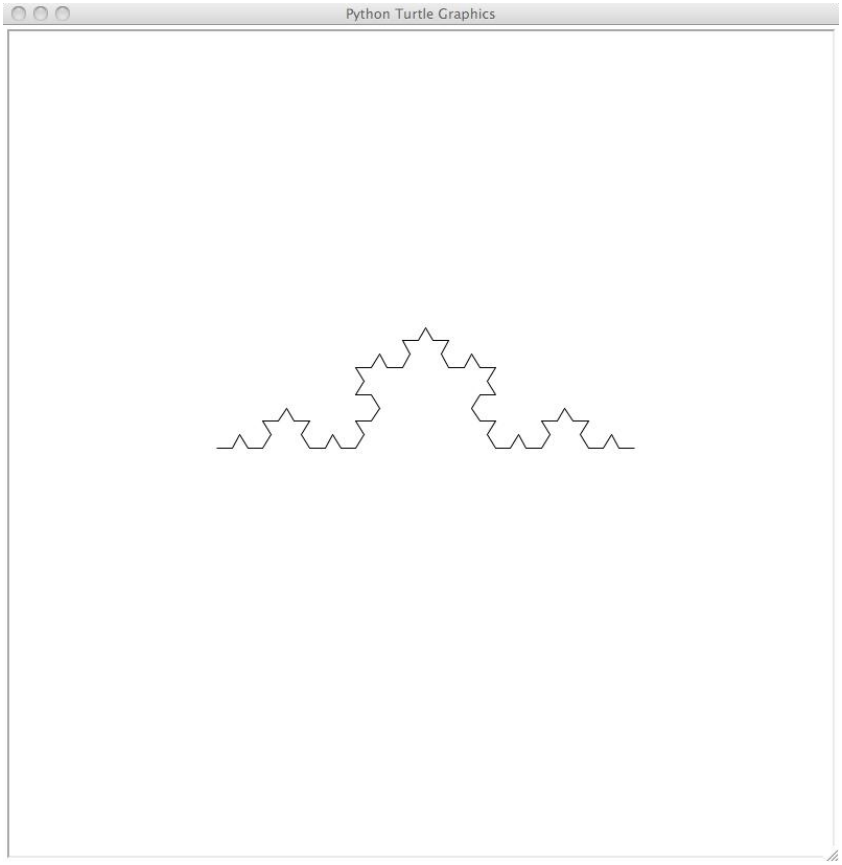
Imagine dividing that line up into three equal parts. Remove the middle part, and replace it by two more parts of the same size, connected to the two original parts that remain. The two new parts are at a 60 degree angle to the original line. The result looks like this:



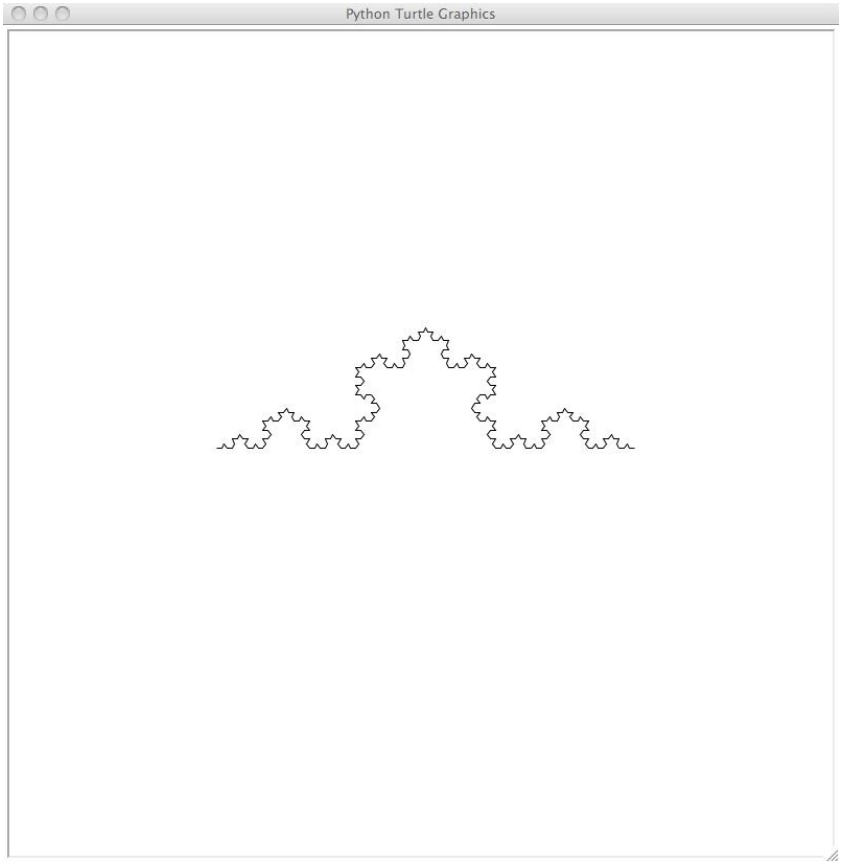
Here is where it becomes fun. The figure we just showed you has four lines in it. Take each of those four lines, and transform it recursively in the same way you transformed the first original line. The result is this:



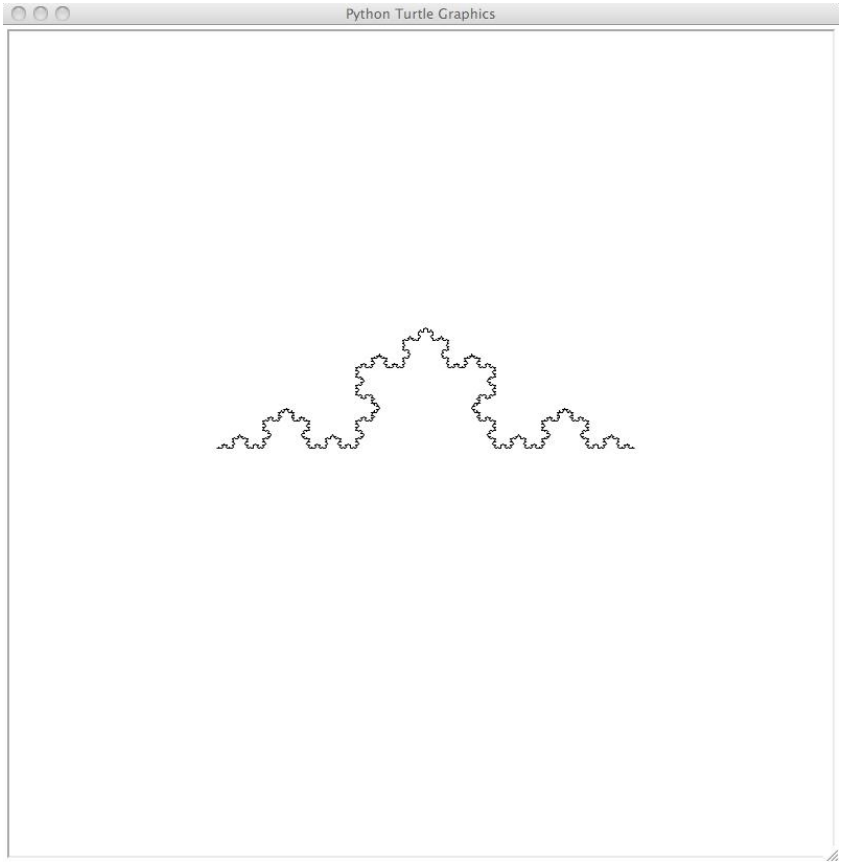
Note how the original line after two transformations of this kind is becoming more and more bumpy. We can keep repeating this transformation for every line in the new figure, leading to this:



One more transformation of this kind gives us this figure:

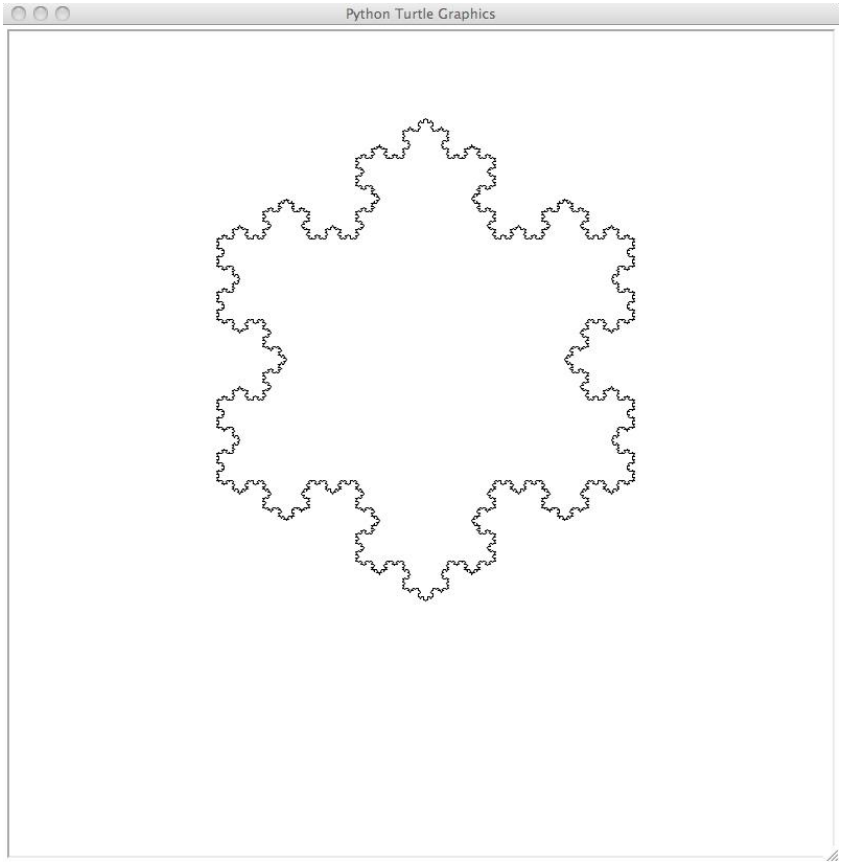


And one more transformation gives this:



We could continue like this, but you get the point. As the transformation is performed over and over, the original straight line gets bumpier and bumpier. Also, every time you do a transformation, the total length of all the lines in the figure increases by a factor of 4/3.

Now for one last trick. Instead of starting with a single line, what if we started with an equilateral triangle (with angles of 60 degrees)? Then we would generate this shape after a few transformations:



And guess what? You just made a Koch snowflake.

OK, now here's how you draw it. First, write a function called `koch_side` which will draw one side of a Koch snowflake (like the images above, except for the last one). This function takes two arguments: the length of the original side and the number of times to divide it up according to the algorithm described above. The images you've seen above (except for the last one), were generated from these calls to `koch_side` on a window of size 800x800 pixels:

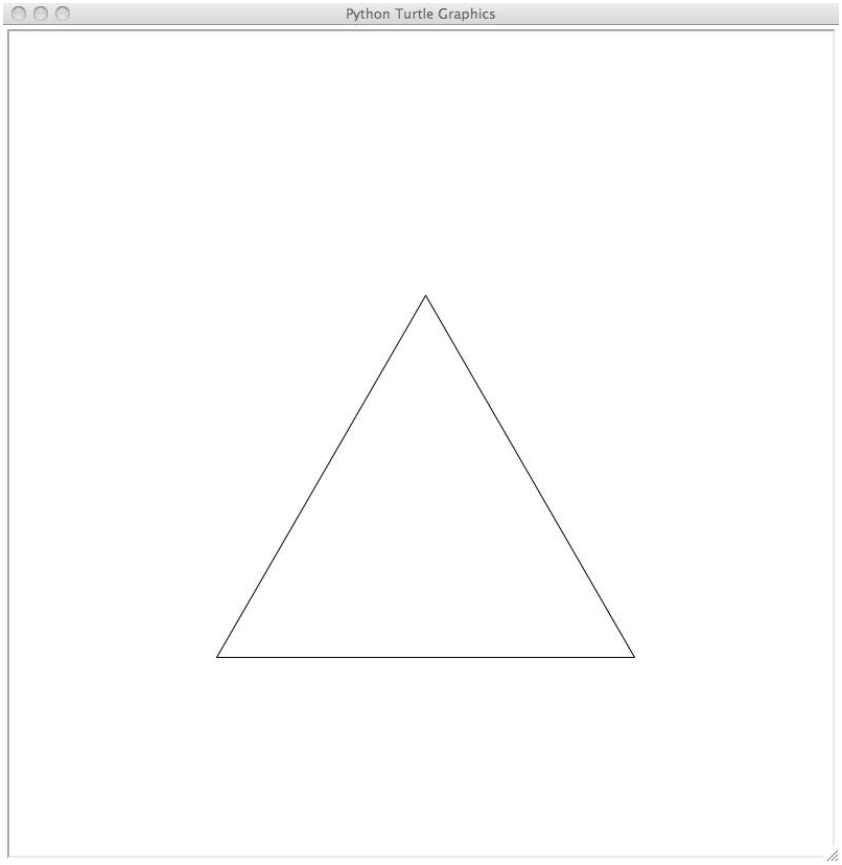
```
koch_side(400.0, 0)  # straight line
koch_side(400.0, 1)
koch_side(400.0, 2)
koch_side(400.0, 3)
koch_side(400.0, 4)
koch_side(400.0, 5)
```

You may find this function to be a bit tricky to write, even though it's only a few lines long. Use recursion, and use the case where the second argument is 0 as your base case (in that case, just draw a straight line). Before implementing the recursion, you should make sure you can draw the first two images shown above. Then replace the straight line drawing parts with recursive calls to get the full function. What part of the function call has to decrease with each recursive call? By how much?

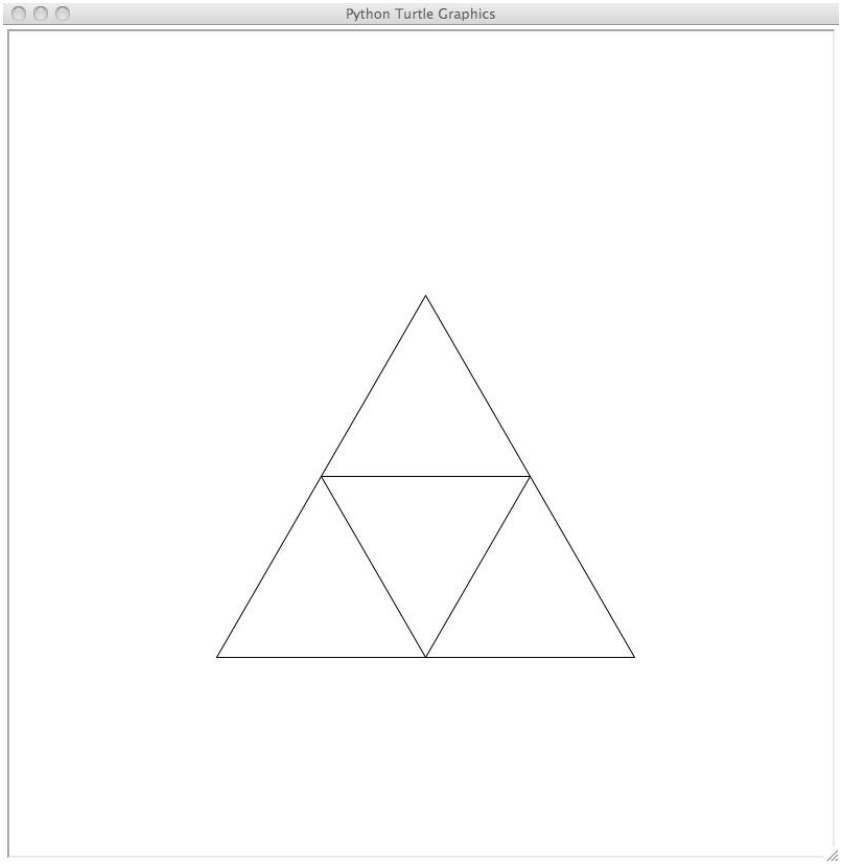
Once you've figured out how to draw the sides of a Koch snowflake, the entire image is easy to draw: just draw three sides, each rotated 120 degrees relative to the previous side. If you do it right, you now have a drawing of a Koch snowflake!

The Sierpinski triangle

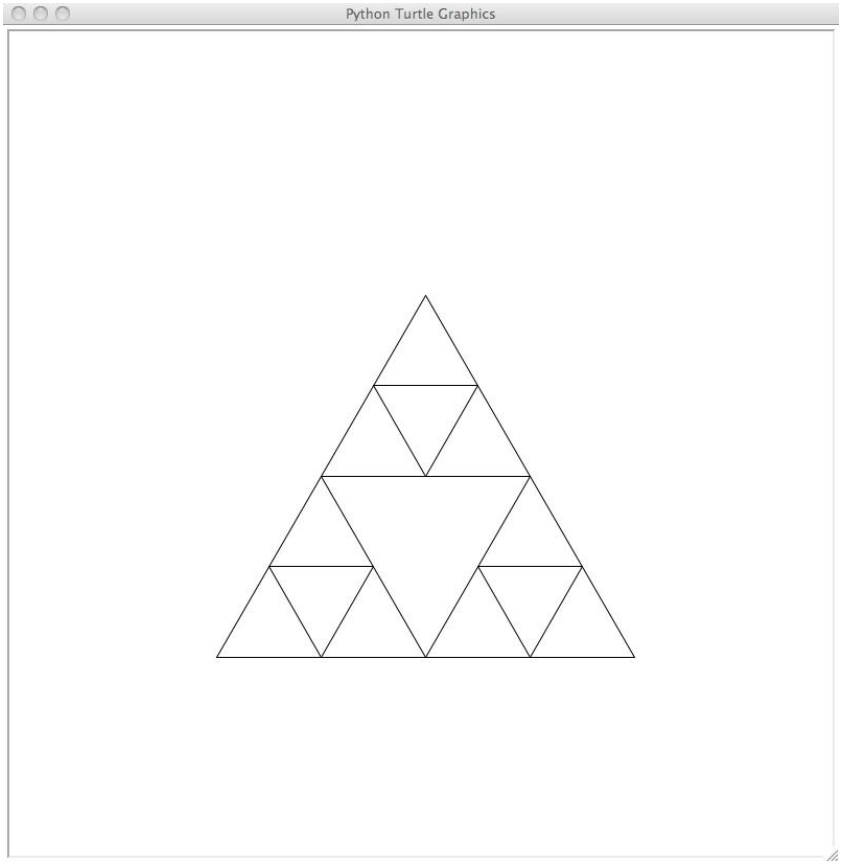
Now that we've drawn one fractal shape, let's draw another. The Sierpinski triangle is a fractal shape made as follows. First, draw a triangle oriented with the base at the bottom of the triangle, like this:



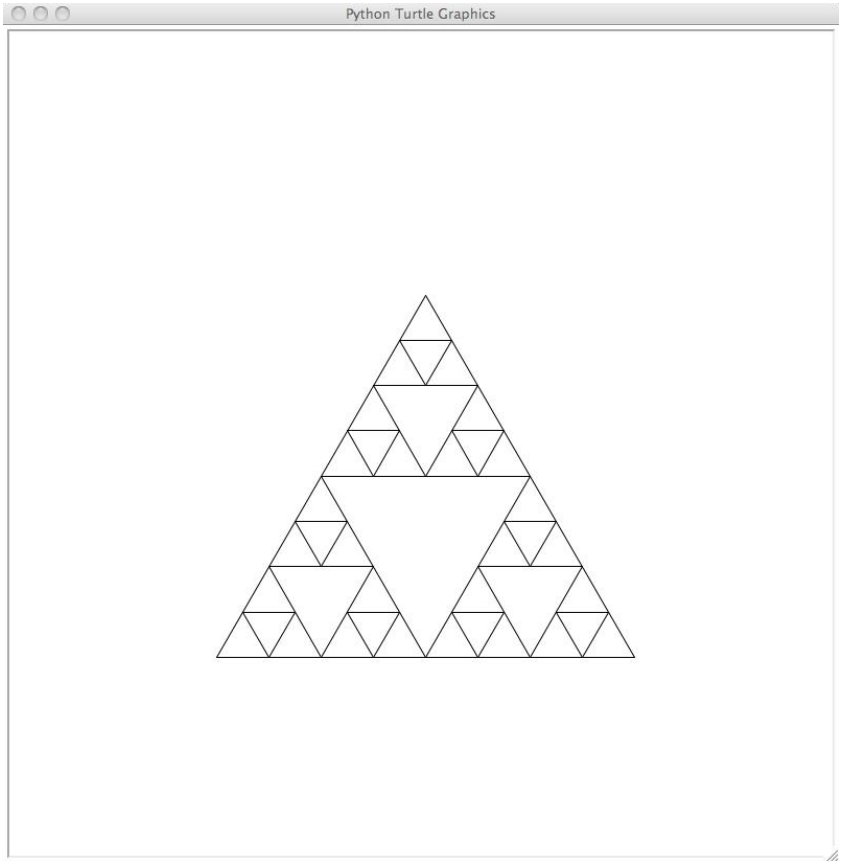
We'll call this orientation of a triangle an "upward-facing triangle", and the opposite orientation a "downward-facing triangle". Now draw a downward-facing triangle of half the size, starting in the middle of one of the edges of the original triangle. This gives this image:



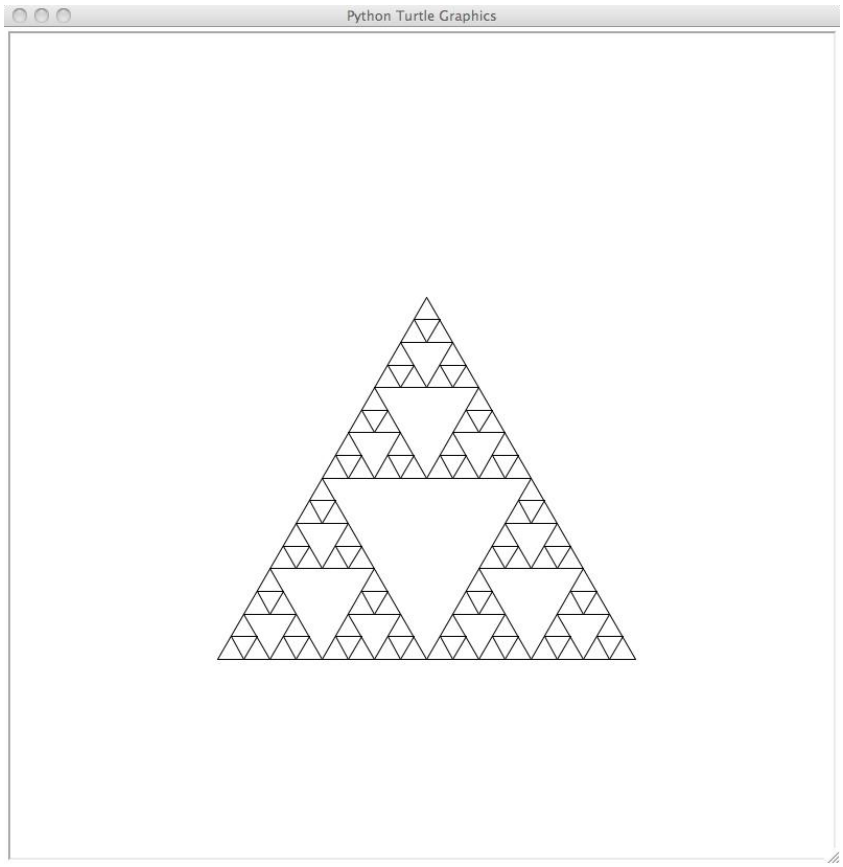
Notice that this image has three small upward-facing triangles. What if we drew downward-facing triangles in the middle of each of them? The result would be this:

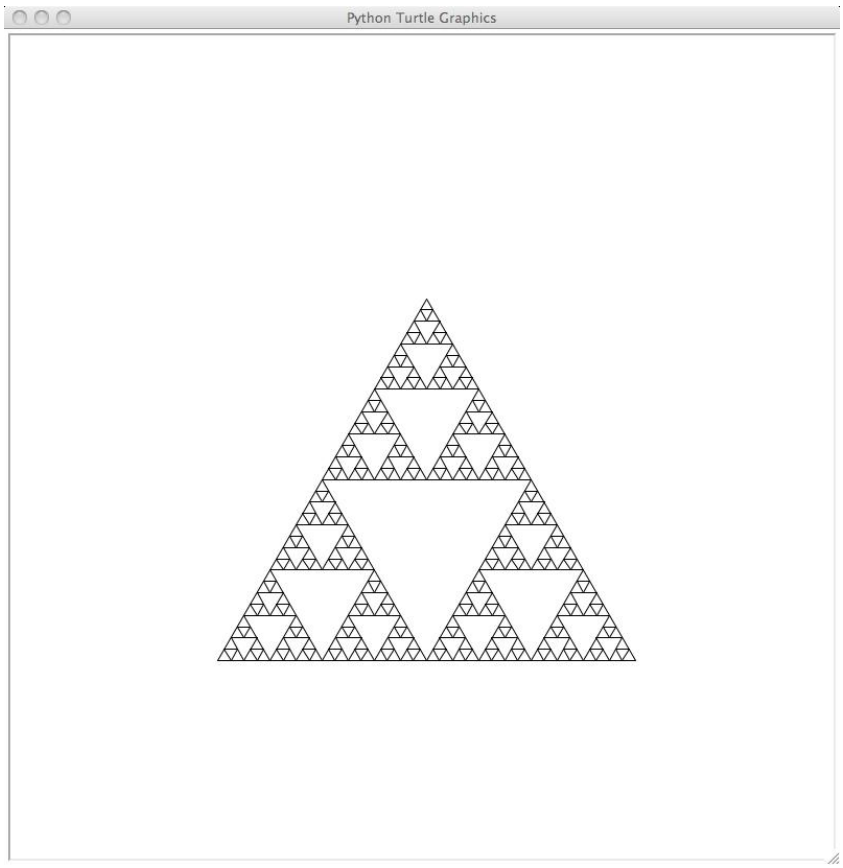


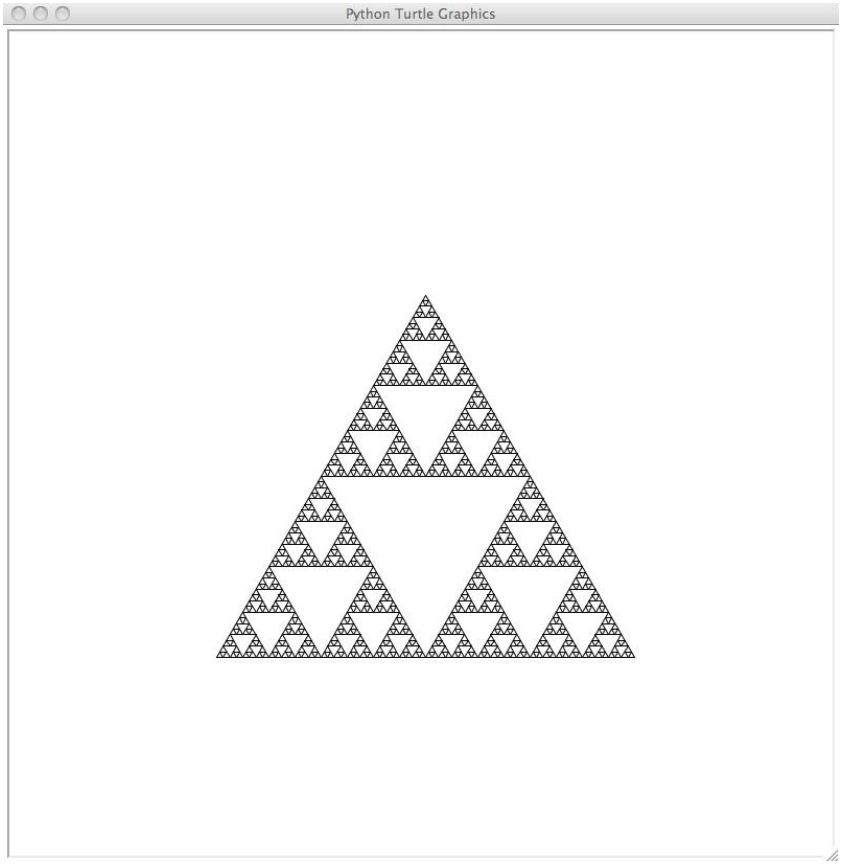
Now we have nine small upward-facing triangles. Again, draw downward-facing triangles in the middle of each of them, to get this:



Repeat this process a few more times, to get these images:


Python Turtle Graphics


Python Turtle Graphics



This fractal shape is the Sierpinski triangle. Notice that apart from the first upward-facing triangle, all you're actually drawing are downward-facing triangles.

Here's how to draw the Sierpinski triangle:

1. First, write functions to draw upward-facing triangles of a particular side length (equilateral triangles) and downward-facing triangles of a particular side length.
2. Write a function called `sierpinski` which takes two arguments: the size of a side of the Sierpinski triangle (in pixels) and the number of levels of iteration (where level 0 is just an upward-facing triangle, level 1 includes one downward-facing triangle, level 2 contains smaller downward-facing triangles, and so on like in the pictures above). This will draw the outermost upward-facing triangle of a Sierpinski triangle, and if the level is greater than 0, it will call the function `sp_helper` to complete the drawing.
3. Write a function called `sp_helper` which takes the same two arguments as `sierpinski`. The level argument should be at least 1 (you can use an `assert` to check for this). This will draw the largest downward-facing triangle for the Sierpinski triangle, and if the level is greater than 1, it will call itself recursively with the size halved and the level decreased by 1. It will have to call itself recursively a total of three times to completely fill up the Sierpinski triangle. Each time the turtle will need to be at a different location, so make sure you move it after each recursive call.

Hint: All the real work of this drawing happens in the `sp_helper` function. It's very easy to make mistakes here, but one way to keep things simple is to make sure that every time you draw a downward-facing triangle inside an upward-facing triangle, you return to the lower-left corner of the upward-facing triangle.

The Hilbert curve

For this last drawing, we'd like you to write functions to draw the Hilbert curve at any desired level of detail. We won't give you any other hints; use what you've learned from the previous two drawings, and have fun!

Speeding up your drawings

When drawing fractals, your program often has to draw a very large number of line segments. The normal drawing speed of the turtle graphics module is much too slow to do this in a reasonable amount of time. You can set the speed to the highest value by setting `speed(0)`, but even that isn't enough. The way to get the absolute maximum drawing speed is to turn tracing of the image by the turtle completely off. Before you start a drawing, you can include this code:

```
tracer(0)
```

This will turn tracing off. Then do the drawing, and at the end, include this code:

```
update()
tracer()
```

The first line causes the drawing to appear, and the second line turns tracing back on.

You only really need to do this for the drawings in the second part of the lab. You can do it on a function-by-function basis.

You probably also want to hide the turtle using `hideturtle()` because the turtle is more of a nuisance than a help when drawing fractals (at least when there are a large number of lines to draw).
