159.172 Computational Thinking

Tutorial 3: Recursive Drawings

In this tutorial you will develop some recursive functions to draw simple pictures and fractals. Since recursion is such an important concept we will come back to it later on the course, for now the idea is to get some practice with writing simple recursive functions. We need to keep a couple of things in mind:

- What is the base case for our recursive solution?
- What is the recursive case? What are we recursing on?
- What parameters will we need to give our functions to make the recursion work?
- What is the terminating condition for our recursive process?

Go to Stream and download the files

```
blocks.py
snowman.py
sierpinski.py
```

Set up a new project and add these files to it.

Task 1

blocks.py is a program that is meant to draw a stack of blocks, of a certain height, at a particular location on the screen. In the main loop you will see the function call

```
draw_stack(screen, 200, 300, 10)
```

This function call should display a stack of 10 blocks on the screen, "anchored" at the location x = 200, y = 300. At the moment this function call results in only one block being displayed, as the *draw_stack* function simply calls the provided *draw_block* function which draws a single block at a particular location. Your first job is to update the function definition

```
def draw_stack(screen, x, y, height):
```

so that it is a recursive function that does display a stack of blocks of the correct height, anchored at the location (x,y). Note that I haven't told you exactly what is meant by "anchored", this will depend on how you choose to define your function and to develop the recursive process. Think about the relative locations of your "base case" single block and of your recursively defined stack of blocks of height one smaller than the final required. Note that the horizontal co-ordinates used by Pygame increase from left to right, the vertical co-ordinates increase from top to bottom.

Task 2

Task 2

snowman.py is a program that draws a snowman at a particular location on the screen. In the main loop you will see the function call

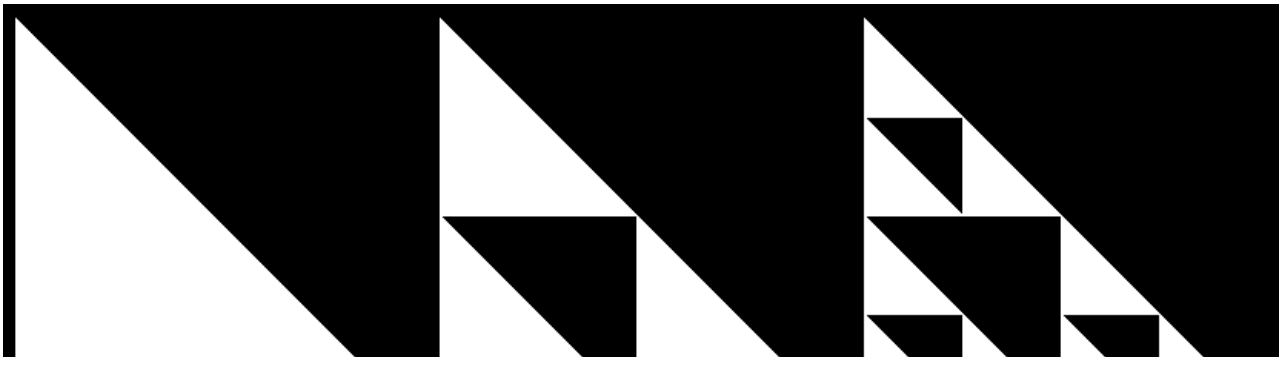
```
draw_snowman(screen, 10, 200)
```

This function call draws a single snowman anchored at the location (10, 200) on the screen. Your second job is to create a new recursive function definition *draw_snowmen(screen, x,y)* that displays a row of snowmen anchored at location (x, y) which will fill the whole screen horizontally. Think about what your recursive process will need to do. What is the terminating condition for the recursion? (When you run out of room on the screen? Note the screen size parameters.) How will you use the function that draws a single snowman, with the parameters provided, to make the recursion work? Replace the *draw_snowman(screen, 10, 200)* function call with your new *draw_snowmen(screen, 10, 200)* version to test it out.

Your third job is a bit harder. Update your *draw_snowmen(screen, x, y)* function definition to a new function *draw_snowmen2* that draws a row of snowmen which get progressively smaller. Now you need to think about the parameters that your recursive function will need. Each recursive call will need both location and size information to make progress. You will need to add a "size" parameter to the function that draws a single snowman and redefine the drawing of the ellipses in terms of the given size, rather than as absolute numbers. Note that the ellipse drawing function uses four parameters, the first two give the x and y co-ordinates of the top left location of the "bounding box" for the ellipse, the third and fourth parameters give the horizontal and vertical magnitude for the ellipse. You will need to define your recursive *draw_snowmen2* function, using the single snowman drawing function as a subroutine, but now taking both location and size into account. What might be a sensible terminating condition for the recursion, when the snowman gets too small? when you run out of room on the screen? a combination of both of these?

Task 3

sierpinksi.py is a program that is meant to draw a fractal image called Sierpinski's Triangle. A *fractal* is a mathematical structure that repeats itself infinitely often in successively finer detail. Sierpinski's Triangle can be described via successive drawings of a triangle. The first drawing is a single triangle, the second drawing subdivides the first triangle by drawing three triangles, each one half the original in both length and height. The third drawing subdivides each of the triangles in the second drawing in the same way, and so on, as shown below. Imagine doing this ad infinitum, and there you have Sierpinski's Triangle.



We can't actually show this infinitely dense triangle in a graphics window, because we are limited by pixel size. So to draw Sierpinski's Triangle we stop subdividing the triangles at some predetermined point, and then just draw each tiny triangle completely at that point.

In the main loop of *sierpinski.py* you will see the function call

```
sierpinski(screen, 0, 512, 512)
```

At the moment this function call draws a single right-angled triangle, with horizontal and vertical sides of length 512, and with its right angle anchored at the location (0, 512) on the screen. The *sierpinski* function simply calls the provided *draw_triangle* function which draws a single triangle of a given size at a given location. Your final job is to update the function definition

```
def sierpinski(screen, x, y, size):
```

so that it is a recursive function that displays an image of Sierpinski's triangle, up to a predetermined resolution, anchored at the point (x,y). You should experiment to work out a sensible terminating triangle size for the recursion. To code your recursive function, follow these steps:

1. Set a minimum size for the triangles that you will draw.
2. Set up the base case for your recursive solution. If the current function call uses a size parameter less than or equal to the minimum size, then you should just go ahead and draw a single triangle.
3. Set up the recursive case. If the current function call uses a size parameter greater than the minimum size, you need to make *three* recursive calls to your function, to recursively draw the three Sierpinski's Triangles that make up your current drawing. It is not always the case that one recursive call will do!

   Each of these recursive calls will use a size parameter exactly one half that of your current function call. Note that the location parameters define the position of the right angle, bottom left of the drawing in this case. You need to figure out how to set the location parameters for each of your three recursive calls so that the first is anchored in the same spot as your current drawing, the second is anchored half way up, and the third is anchored half way across.

Submit a copy of your *blocks.py*, *snowman.py* and *sierpinski.py* files, with your completed functions, via Stream. Include a brief report of any experimentation that you have done. Submit your work via Stream by 12 midnight, Sunday August 9th.

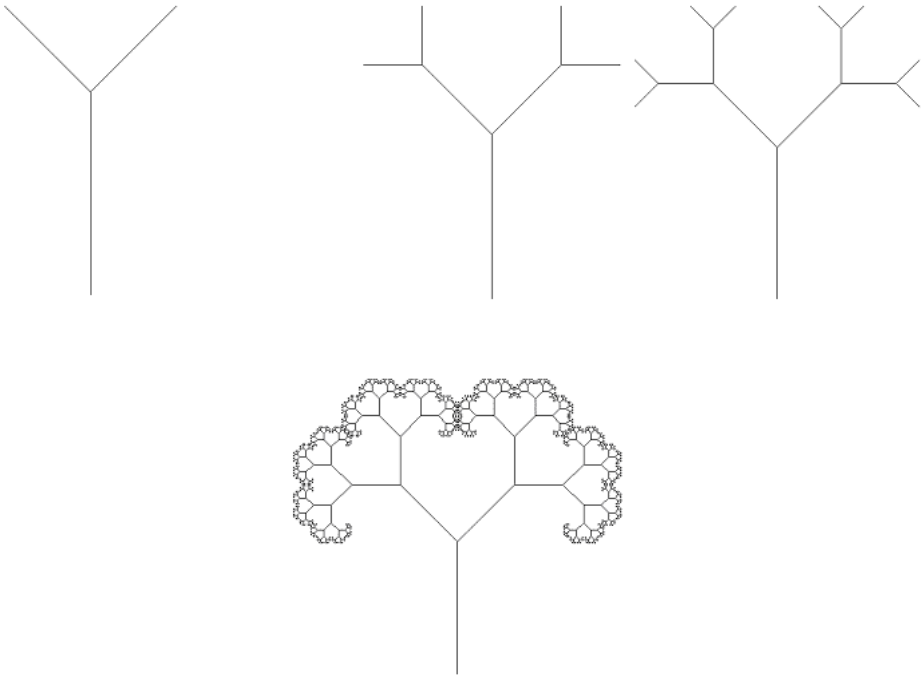
Optional

If you enjoy this sort of thing, try writing Python programs to draw the following

If you enjoy this sort of thing, try writing Python programs to draw the following fractals.You can find documentation for the required Pygame drawing commands at www.pygame.org/docs/ref/draw.html.

1. The *Snowflake* fractal. Draw a Star of David (two opposing equilateral triangles, one superimposed on the other). Then repeat the process for each of the six points, but with drawings one third of the size.

2. The Lindenmayer *Tree* fractal. Draw a tree, which is a trunk of length **`t`** with two length **`t/2`** branches coming out of the top of it at a right angle to each other. Each branch is itself a tree, so now repeat the process for each of the branches.



To see some nice examples of fractal art, using more complex versions of these sorts of *self-similar transforms*, go to

http://www.smashingmagazine.com/2008/10/17/50-phenomenal-fractal-art-pictures