Drawing Fractals

A fractal shape is an ideal way to explore recursion, because each step 'into' a self-similar fractal produces a copy of the original shape. Since each step is relatively simple, we can code the whole thing recursively.

A perfectly self-similar fractal is one which:

- ***Is the same as a subset of itself.***
- ***Contains an exact replica of itself.***

This definition is a little bit mind-bending, and of course can only be genuinely true for a shape with infinite complexity and infinite detail. But we can generate the first few stages of a fractal shape simply by repeating our instructions within themselves.

Example:
Consider the following infinite continued fraction:

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cdots}}}}}$$

Notice that the fraction continues forever, but the pattern is always the same. If we focus on the term which forms the first denominator, we find it exactly matches the entire thing:

$$1+\cfrac{1}{\color{red}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cdots}}}}}}$$

Which means that, if we label our fraction x, we could actually write the following:

$$x = 1 + \frac{1}{x}$$

Solving this equation (which turns into a quadratic) will yield not only one but two possible values for our infinite fraction! You can even verify them directly. Try the same approach to find the value of:

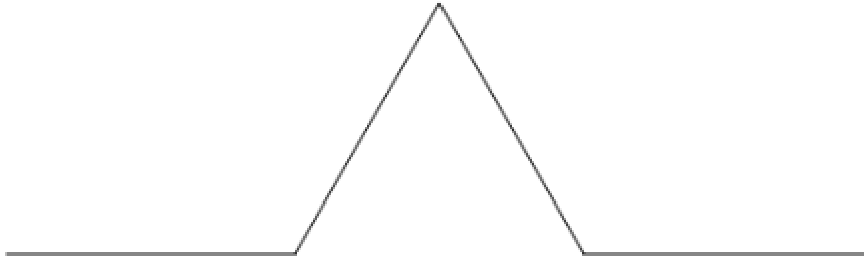
$$2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+\cdots}}}}$$

The Koch Snowflake:

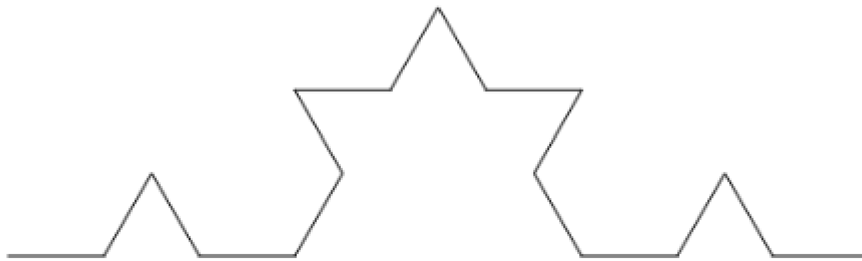
This is a classic fractal pattern that is derived from a very simple recursive relationship:

Step 1: Begin with a single line:

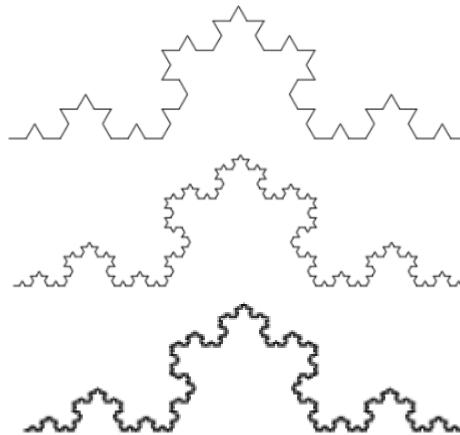
Step 2: Divide the line into three equal sections, and replace the middle third with an equilateral triangle:



Step 3: For each of the four lines in your shape, repeat step 2:

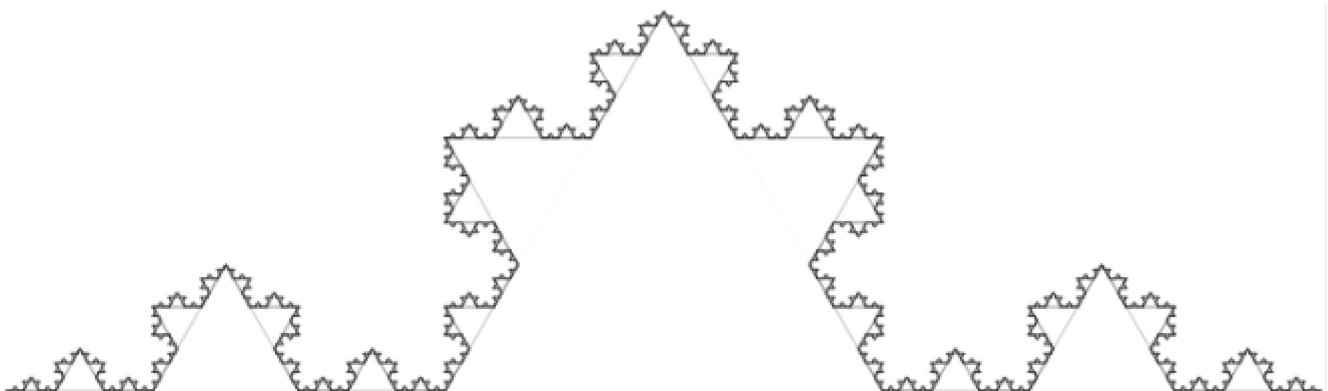


Step 4: Repeat step 3 forever (or until the length of line is less than 5 pixels long):



…

And drawing each new iteration with a darker grey shows the progression quite nicely:



Turn over to see how to generate this fractal, and others, using Python.

Drawing Fractals *with Python*

For this project, if you're using repl.it, you'll need to choose:

The recursive capacity of functions in Python is what lets us create a shape which contains itself. A function can call itself, which means I can do things like this:

```
1  import turtle
2
3  t = turtle.Turtle()
4
5  def spiral(length):
6    if length > 1:
7      t.forward(length)
8      t.right(90)
9      spiral(length - 4)
10
11 spiral(100)
```

The first line loads the code within Python's '**turtle**' module. Next, I create a **turtle** object called **t**.

The function here takes a **length** as an input, and does nothing if it is called with a length which is 1 or less.
If **length** is larger than 1, it moves forward that far, turns right 90 degrees, and then… runs that same code all over again, but with a smaller length.

Running **spiral(100)** moves us forward 100, turn right 90 degrees, then run **spiral(96)**, which passes control down to this new function, waiting for it to finish before continuing:

Run **spiral(100)**:

fd 100, rt 90, run **spiral(96)**:

………

fd 4, rt 90, run **spiral(0):**

do nothing.

When the final function call resolves (**spiral(0)** does nothing and then terminates), control is passed back to the function above, which, since there is no more to do, itself terminates, and so on all the way back up the chain. To see exactly when each bit happens, you can add: `print(f"Starting spiral with length {length}")` at the very start of the function, and: `print(f"Completed spiral with length = {length}")` at the end.

Fractal Y-tree:

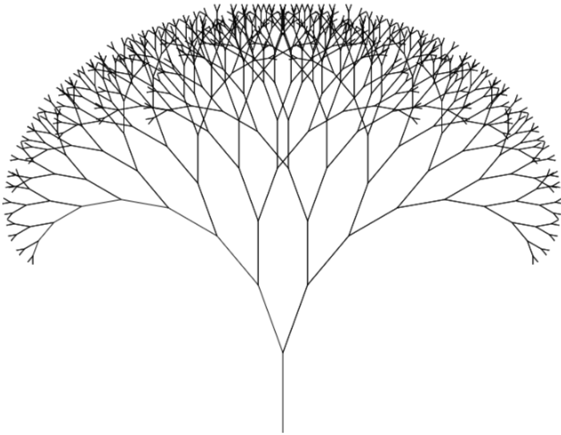
```
1  import turtle
2
3  t = turtle.Turtle()
4
5  def branch(length):
6    if length > 5:
7      t.fd(length)
8      t.rt(20)
9      branch(length - 10)
10     t.lt(40)
11     branch(length - 10)
12     t.rt(20)
13     t.bk(length)
14
15 t.penup()
16 t.goto(0, -200)
17 t.setheading(90)
18 t.pendown()
19 branch(100)
```

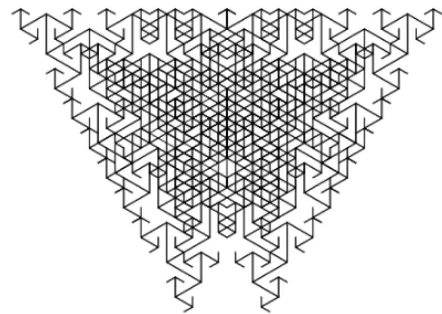
This is a lovely fractal to create, partly because the final result is pleasing visually, but also because we can watch the process in action, seeing each level of recursion.
The **branch** function is the key recursive element in this program: it moves forward, turns right, then *does **branch** again*, with a smaller length. After that, it turns left, *does **branch** again* (with the same smaller length), and finally turns back and reverses, to finish where it started.

Tip: to speed up the animation, add the line ***`t.speed(0)`***
The speed can be set from 1 to 10, but 0 means 'as fast as the processor allows'. You can improve speed still further using ***`t.hideturtle()`****, and* ***`turtle.tracer(20)`*** *instructs the program to only update the screen every 20ms (massively speeding up the animation process).*

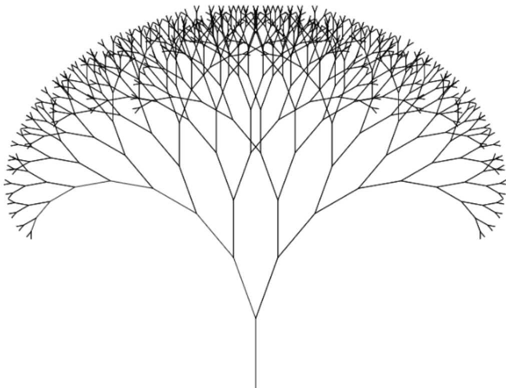
Extending the code

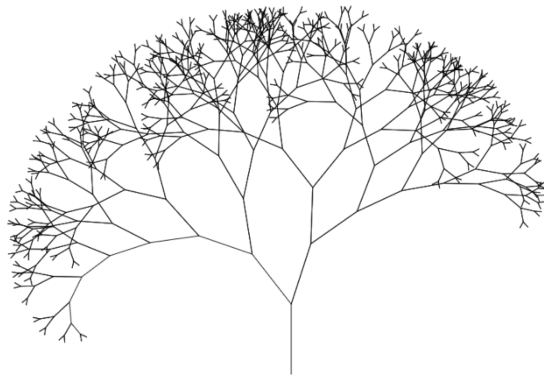
Customisation:
The functions we've constructed had a few 'magic numbers' in them: that is, values that we use throughout the function, but have not been assigned to a variable. The main ones in the Y-tree fractal are the angle we turn (we chose 20°), the level of precision (we had 5) and the amount by which the branch length reduces at each stage (we used 10).
By defining some or all of those numbers as variables, we can more easily change them. You could even make them parameters for the function along with **length**, so you could quickly investigate the effect of changing them:





Randomization:
By including the line `import random` at the top, you can make use of functions like `random.randint(10, 40)` to produce random numbers between limits, allowing you to encode some 'natural' variation into your trees:





Beautification:
Look up the documentation for the Python Turtle module by googling "python turtle documentation", and you can find details of how to do all sorts of extra things such as change the colour or width of the lines drawn, paving the way for all sorts of pretty trees: