In [1]:
```
using PyPlot
```

Floating Point Precision

Floating point numbers in Julia (and all modern computing practically) follows something called the IEEE floating point standard, where the number is split into:

1. Sign bit (1 bit)
2. Exponent bits (8/11 bits for single/double precision)
3. Mantissa, the fraction part (all other bits)

The idea is that any floating point number can be written as 1.(some digits) * 2^exponent

This is why cutoff happens, you can imagine that there is a smallest number that can be represented using this with a finite number of bits, this is called machine epsilon, in Julia this can be seen using the `eps` command.

In [2]:
```
eps(Float16)
```

Out[2]:
```
Float16(0.000977)
```

In [3]:
```
eps(Float32)
```

Out[3]:
```
1.1920929f-7
```

In [4]:
```
eps(Float64)
```

Out[4]:
```
2.220446049250313e-16
```

These precisions are called half, single, double respectively (naming for historical purposes). I am sweeping a some details under the rug (subnormal numbers, exceptions, ...) but come ask me if you're interested.

You may ask why we want less precision? Speed! The smaller the number the faster operations can be performed and sometimes you don't need your answer to be too accurate. There is actually development for 8 bit floating point numbers - driven largely by ML communities (e.g. Google).

Recursion

Recursion is where a function calls itself on a smaller subproblem in order to compute something. This is basically coding up numerical induction.

In numerical induction, you say that some hypothesis is true for all $k < n$ and that for the $k = n$ case, you can use the previous $k < n$ cases to solve it. This is how recursion works.

As an example, let's consider the Fibonacci numbers:

$$a_1 = 1,$$
$$a_2 = 1,$$
$$a_n = a_{n-1} - a_{n-2}$$

Notice how for the nth number we need the n-1 and n-2 Fibonacci numbers. This thus suggests a recursive type structure on a problem with "smaller n".

In [5]:
```
function fibonacci(n)

    if n < 1
        return 0
    end
    if n == 1 || n == 2
        return 1
    end

    return fibonacci(n-1) + fibonacci(n-2)

end
```

Out[5]:
```
fibonacci (generic function with 1 method)
```

In [6]:
```
fibonacci(7)
```

Out[6]:
```
13
```

It works! It might seem a little bit mystical but the idea really is that if we get the base cases right (where you don't need to use some recurrence relation) and can combine smaller cases correctly then by induction it should just work.

WARNING While recursion is often easy to code up, it actually tends to have very poor performance compared to iteration unless you are extremely careful. This is because everytime you reinstance the functionyou allocate new memory for that without releasing the memory for the original function call which can grow exponentially.

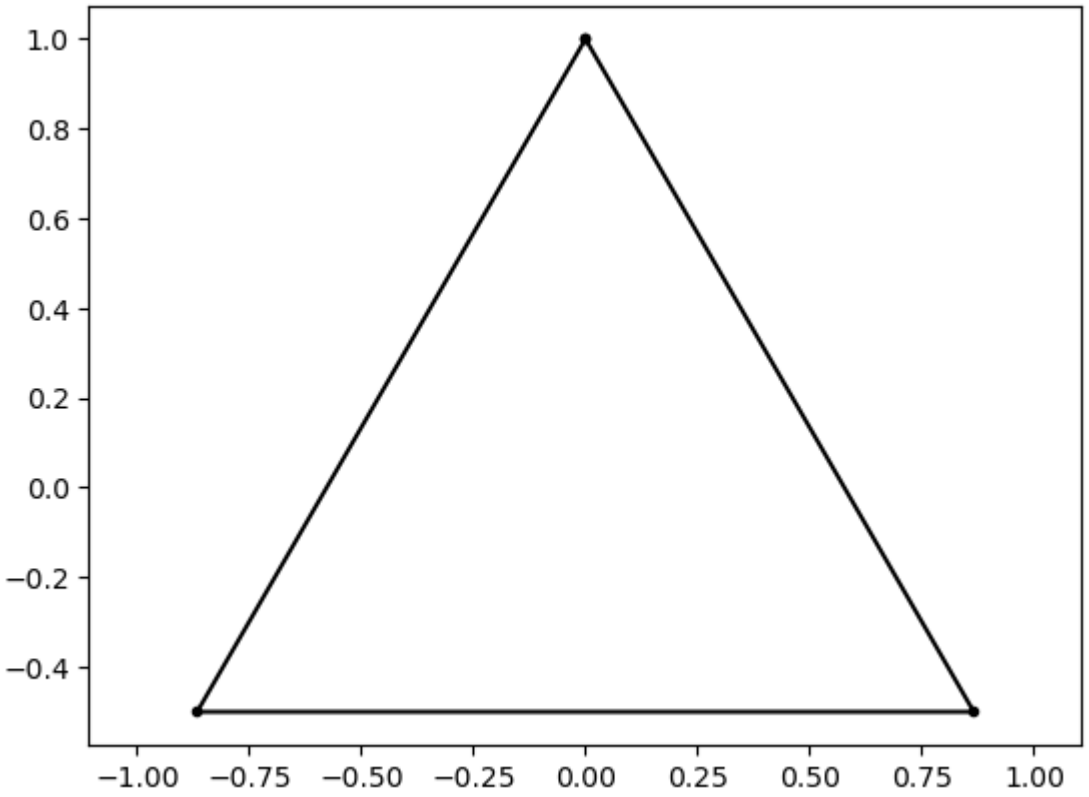
In fact anything you can do with recursion you can do with iteration and in practice people just use iteration. However for things like recurrence relations it is often conceptually easier to use recursion for prototyping etc.

Let's do another example that recursion is famous for - drawing fractals. A fractal is a shape is a pattern that contains itself infinitely many times. Famoous examples are the Mandelbrot set (in you hw) and the Sierpinski triangle which we will do here.

In [7]:
```
function drawTriangle1( x,y,radius )

    ptsx = [ x+radius*cos(pi*0.5), x+radius*cos(7*pi/6), x+radius*cos(11
    ptsy = [ y+radius*sin(pi*0.5), y+radius*sin(7*pi/6), y+radius*sin(11

    plot(ptsx, ptsy, "k.-")
    axis("equal")

end
```

Out[7]:
```
drawTriangle1 (generic function with 1 method)
```

In [8]:
```
drawTriangle1( 0,0,1 );
```

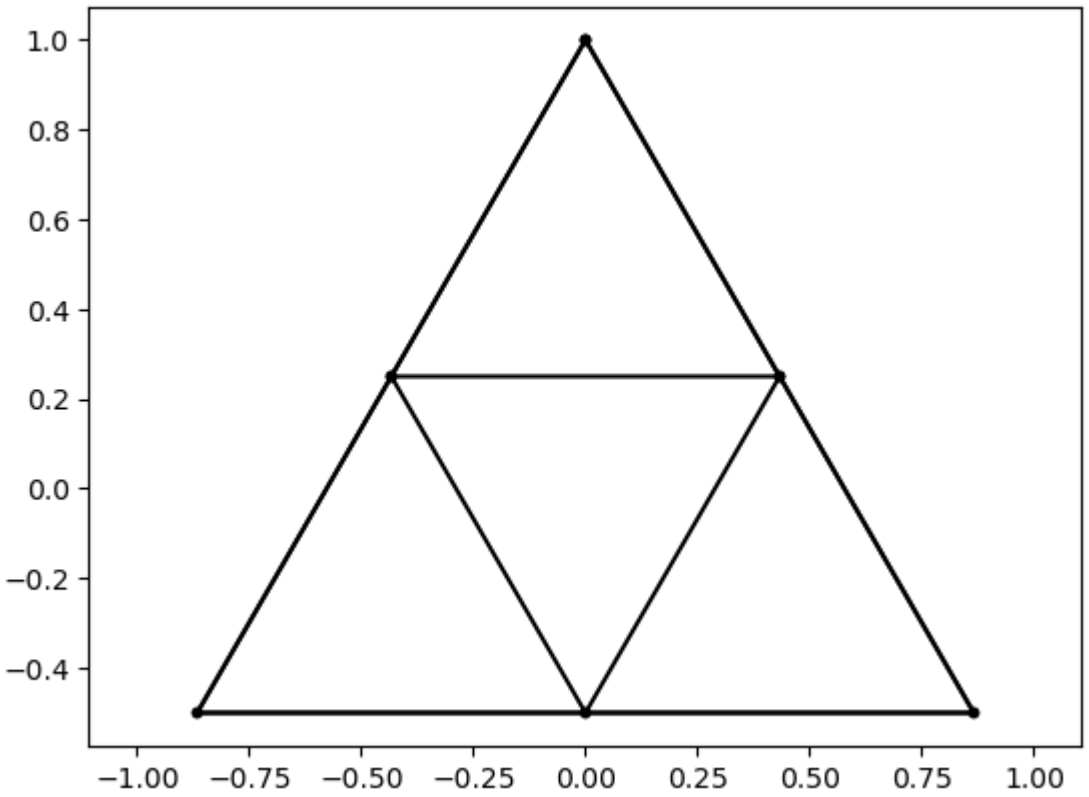


So this is the base case - or the base shape. Let's now look at the n=2 case.

In [9]:
```
function drawTriangle2( x,y,radius )

    ptsx = [ x+radius*cos(pi*0.5), x+radius*cos(7*pi/6), x+radius*cos(11
    ptsy = [ y+radius*sin(pi*0.5), y+radius*sin(7*pi/6), y+radius*sin(11

    plot(ptsx, ptsy, "k.-")
    axis("equal")

    drawTriangle1( x+radius*cos(pi*0.5)/2, y+radius*sin(pi*0.5)/2, radiu
    drawTriangle1( x+radius*cos(7*pi/6)/2, y+radius*sin(7*pi/6)/2, radiu
    drawTriangle1( x+radius*cos(11*pi/6)/2, y+radius*sin(11*pi/6)/2, rad

end
```

Out[9]:
```
drawTriangle2 (generic function with 1 method)
```

In [10]:
```
drawTriangle2( 0,0,1 );
```

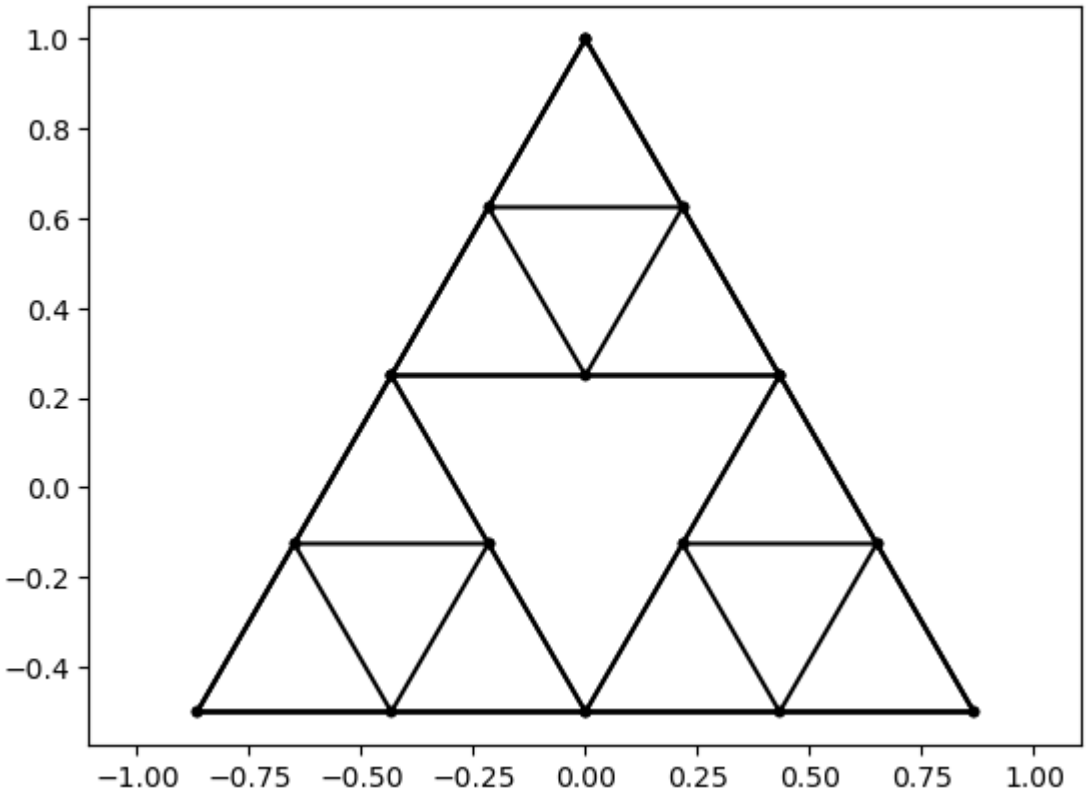


So we are just calling the function inside but with smaller parameters! To generalise this for arbitrary n > 0 then

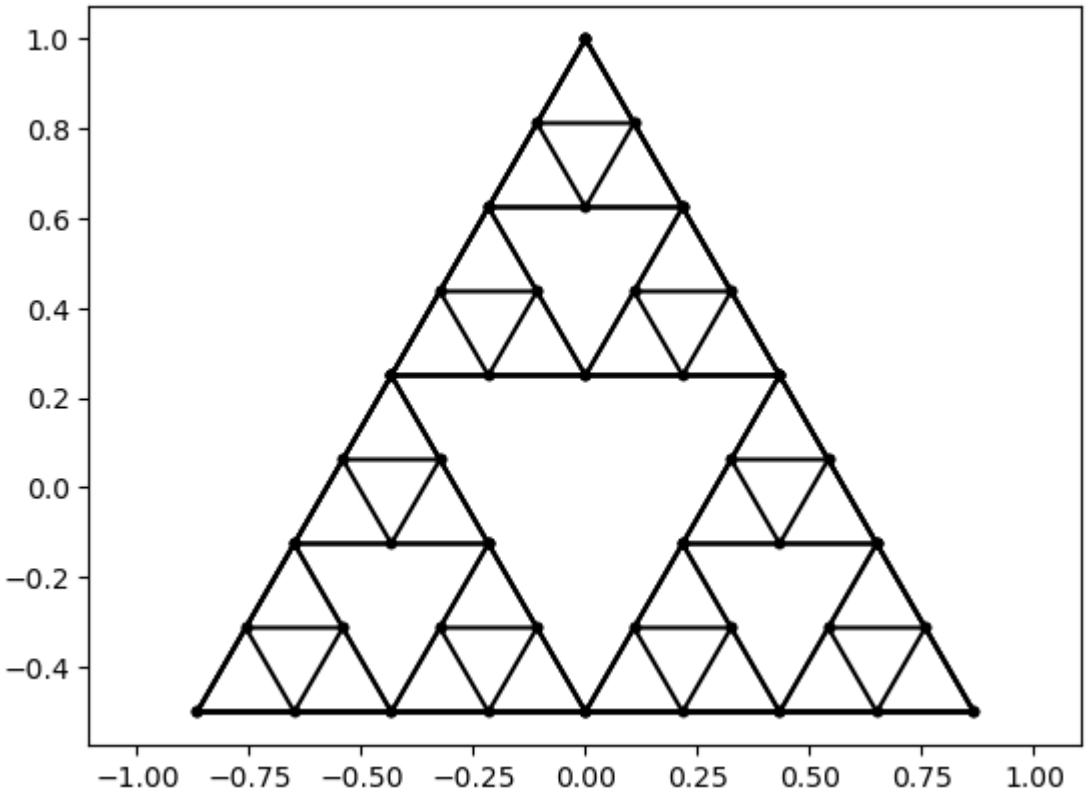
In [11]:
```
function drawTriangle( x,y,radius,count )

    ptsx = [ x+radius*cos(pi*0.5), x+radius*cos(7*pi/6), x+radius*cos(11
    ptsy = [ y+radius*sin(pi*0.5), y+radius*sin(7*pi/6), y+radius*sin(11

    plot(ptsx, ptsy, "k.-")
    axis("equal")

    if count > 1
        drawTriangle( x+radius*cos(pi*0.5)/2, y+radius*sin(pi*0.5)/2, ra
        drawTriangle( x+radius*cos(7*pi/6)/2, y+radius*sin(7*pi/6)/2, ra
        drawTriangle( x+radius*cos(11*pi/6)/2, y+radius*sin(11*pi/6)/2,
    end

end
```

Out[11]:
```
drawTriangle (generic function with 1 method)
```

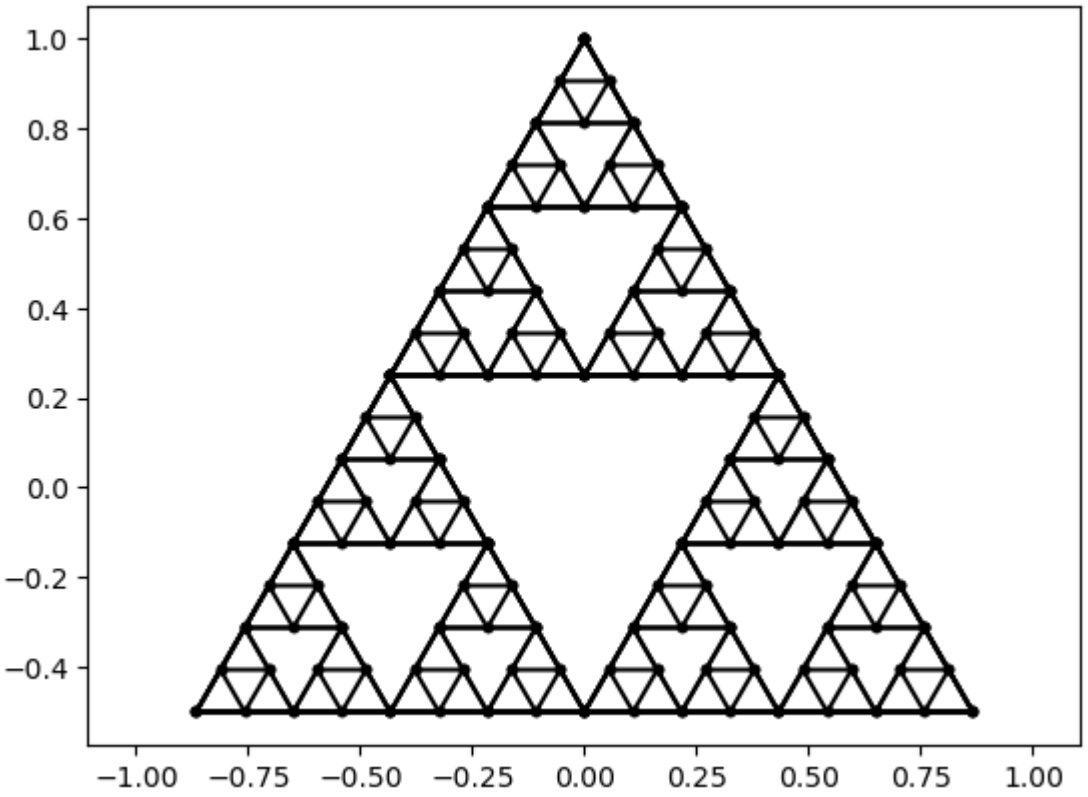
In [12]:
```
drawTriangle( 0,0,1,3 );
```



In [13]:
```
drawTriangle( 0,0,1,4 );
```



In [14]:
```
drawTriangle( 0,0,1,5 );
```



In [ ]:
```

```