

17. Recursion

Recursion means “defining something in terms of itself” usually at some smaller scale, perhaps multiple times, to achieve your objective. For example, we might say “A human being is someone whose mother is a human being”, or “a directory is a structure that holds files and (smaller) directories”, or “a family tree starts with a couple who have children, each with their own family sub-trees”.

Programming languages generally support **recursion**, which means that, in order to solve a problem, functions can *call themselves* to solve smaller subproblems.

17.1. The Factorial Function

A simple example of a problem that is recursive in nature is the factorial of a natural number n , denoted $n!$. Recall that $n!$ is the product of the natural numbers up to and including n , i.e. $n! = 1 \cdot 2 \cdot 3 \cdots n$. For example, $6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$.

A recursive way to think of this is to observe that $n! = n \cdot (n - 1)!$. In other words, computing $n!$ involves computing $(n - 1)!$. Taking advantage of the fact that a Python function may call itself, we have:

```

1 def factorial(n):
2     if n == 0 or n==1:
3         return 1
4     return n*factorial(n-1)

```

Notice that this function gives the correct value, 1, for both $0!$ and $1!$.

17.2. The Knapsack Problem

The knapsack problem is a more complex example of a problem that is amenable to a recursive solution; a non-recursive solution would be much more difficult to organize. A simple version of the problem goes like this:

The Knapsack Problem

Given n objects with weights w_1, w_2, \dots, w_n , determine whether there exists a subset of the objects with a total weight equal to some specified target weight W .

A solution is simple to state: try all combinations until the target weight is made or until we run out of combinations. On a small set of weights this is easy to do by inspection, but the programming of such a task for a large number of weights is not trivial. (Unfortunately, this is an example of a problem for which the best known solution is of the “try all possibilities” variety.)

Let’s state the proposed solution with a bit more precision and see how this restatement leads almost directly to the recursive procedure to solve the problem. Given a target weight W , we first guess that we can use the first weight w_1 . In that case we have a new version of the knapsack problem to deal with: the target weight is now $(W - w_1)$ and the list of available weights is w_2, w_3, \dots, w_n . We solve this smaller problem; if the answer is *yes* we’re in business, because then the answer to the original problem is also *yes*. If the answer to the smaller problem is *no*, then we decide that the weight w_1 is of no use to us, and instead ask whether we can make the target W out of w_2, w_3, \dots, w_n .

We proceed in this manner until we have tried all possible combinations. The only thing left is to examine a couple of special cases in which we can give an immediate answer. If at any point we are trying to solve any version of the knapsack problem with target weight 0, then we can answer *yes* immediately, since we can always make a target of 0 by choosing no weights at all. If we have a target

weight which is less than 0, the answer is clearly *no*, since a negative weight makes no sense. Finally, if we have a non-zero target weight but no more weights to consider, we return the answer *no*. This has a nearly direct translation into a recursive procedure, shown here:

```

1  def knapsack(target, candidate, weight):
2      if target == 0:
3          return True
4      elif target < 0 or candidate >= len(weight):
5          return False
6      elif knapsack(target - weight[candidate], candidate+1, weight):
7          print("Weight number: ", candidate)
8          print("Value: ", weight[candidate])
9          return True
10     else:
11         return knapsack(target, candidate+1, weight)

```

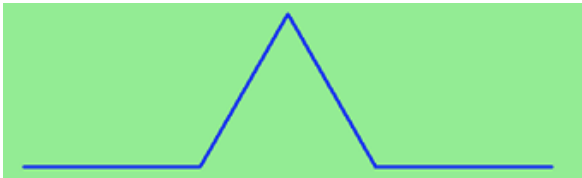
17.3. Drawing Fractals

For our purposes, a **fractal** is a drawing which also has *self-similar* structure, where it can be defined in terms of itself.

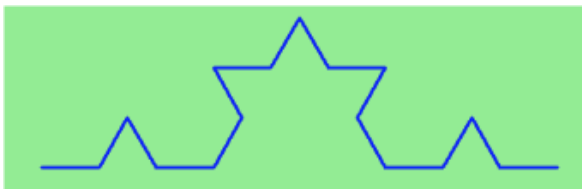
Let us start by looking at the famous Koch fractal. An order 0 Koch fractal is simply a straight line of a given size.



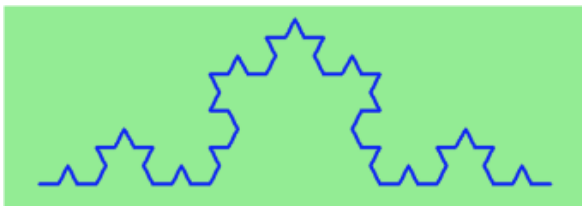
An order 1 Koch fractal is obtained like this: instead of drawing just one line, draw instead four smaller segments, in the pattern shown here:



Now what would happen if we repeated this Koch pattern again on each of the order 1 segments? We'd get this order 2 Koch fractal:



Repeating our pattern again gets us an order 3 Koch fractal:



Now let us think about it the other way around. To draw a Koch fractal of order 3, we can simply draw four order 2 Koch fractals. But each of these in turn needs four order 1 Koch fractals, and each of those in turn needs four order 0 fractals. Ultimately, the only drawing that will take place is at order 0. This is very simple to code up in Python:

```

1  def koch(t, order, size):
2      """
3      Make turtle t draw a Koch fractal of 'order' and 'size'.

```

```

4      """ Leave the turtle facing the same direction.
5
6
7      if order == 0:          # The base case is just a straight line
8          t.forward(size)
9      else:
10         koch(t, order-1, size/3) # Go 1/3 of the way
11         t.left(60)
12         koch(t, order-1, size/3)
13         t.right(120)
14         koch(t, order-1, size/3)
15         t.left(60)
16         koch(t, order-1, size/3)

```

The key thing that is new here is that if order is not zero, `koch` calls itself recursively to get its job done.

Let's make a simple observation and tighten up this code. Remember that turning right by 120 is the same as turning left by -120. So with a bit of clever rearrangement, we can use a loop instead of lines 10–16:

```

1  def koch(t, order, size):
2      if order == 0:
3          t.forward(size)
4      else:
5          for angle in [60, -120, 60, 0]:
6              koch(t, order-1, size/3)
7              t.left(angle)

```

The final turn is 0 degrees — so it has no effect. But it has allowed us to find a pattern and reduce seven lines of code to three, which will make things easier for our next observations.

Recursion, the high-level view

One way to think about this is to convince yourself that the function works correctly when you call it for an order 0 fractal. Then do a mental *leap of faith*, saying “the fairy godmother (or Python, if you can think of Python as your fairy godmother) *knows how to handle the recursive level 0 calls for me on lines 11, 13, 15, and 17, so I don't need to think about that detail!*” All I need to focus on is how to draw an order 1 fractal *if I can assume the order 0 one is already working*.

You're practicing *mental abstraction* — ignoring the subproblem while you solve the big problem.

If this mode of thinking works (and you should practice it!), then take it to the next level. Aha! now can I see that it will work when called for order 2 *under the assumption that it is already working for level 1*.

And, in general, if I can assume the order $n-1$ case works, can I just solve the level n problem?

Students of mathematics who have played with proofs of induction should see some very strong similarities here.

Recursion, the low-level operational view

Another way of trying to understand recursion is to get rid of it! If we had separate functions to draw a level 3 fractal, a level 2 fractal, a level 1 fractal and a level 0 fractal, we could simplify the above code, quite mechanically, to a situation where there was no longer any recursion, like this:

```

1  def koch_0(t, size):
2      t.forward(size)
3
4  def koch_1(t, size):
5      for angle in [60, -120, 60, 0]:
6          koch_0(t, size/3)
7          t.left(angle)
8
9  def koch_2(t, size):
10     for angle in [60, -120, 60, 0]:
11         koch_1(t, size/3)
12         t.left(angle)
13

```

```

14     def koch_3(t, size):
15         for angle in [60, -120, 60, 0]:
16             koch_2(t, size/3)
17             t.left(angle)

```

This trick of “unrolling” the recursion gives us an operational view of what happens. You can trace the program into `koch_3`, and from there, into `koch_2`, and then into `koch_1`, etc., all the way down the different layers of the recursion.

This might be a useful hint to build your understanding. The mental goal is, however, to be able to do the abstraction!

17.4. Recursive data structures

All of the Python data types we have seen can be grouped inside lists and tuples in a variety of ways. Lists and tuples can also be nested, providing many possibilities for organizing data. The organization of data for the purpose of making it easier to use is called a **data structure**.

It's election time and we are helping to compute the votes as they come in. Votes arriving from individual wards, precincts, municipalities, counties, and states are sometimes reported as a sum total of votes and sometimes as a list of subtotals of votes. After considering how best to store the tallies, we decide to use a *nested number list*, which we define as follows:

A *nested number list* is a list whose elements are either:

- a. numbers
- b. nested number lists

Notice that the term, *nested number list* is used in its own definition. **Recursive definitions** like this are quite common in mathematics and computer science. They provide a concise and powerful way to describe **recursive data structures** that are partially composed of smaller and simpler instances of themselves. The definition is not circular, since at some point we will reach a list that does not have any lists as elements.

Now suppose our job is to write a function that will sum all of the values in a nested number list. Python has a built-in function which finds the sum of a sequence of numbers:

```

>>> sum([1, 2, 8])
11

```

For our *nested number list*, however, `sum` will not work:

```

>>> sum([1, 2, [11, 13], 8])
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>>

```

The problem is that the third element of this list, `[11, 13]`, is itself a list, so it cannot just be added to `1`, `2`, and `8`.

17.5. Processing recursive number lists

To sum all the numbers in our recursive nested number list we need to traverse the list, visiting each of the elements within its nested structure, adding any numeric elements to our sum, and *recursively repeating the summing process* with any elements which are themselves sub-lists.

Thanks to recursion, the Python code needed to sum the values of a nested number list is surprisingly short:

```

1 def r_sum(nested_num_list):

```

```

2     tot = 0
3     for element in nested_num_list:
4         if type(element) == type([]):
5             tot += r_sum(element)
6         else:
7             tot += element
8     return tot

```

The body of `r_sum` consists mainly of a `for` loop that traverses `nested_num_list`. If `element` is a numerical value (the `else` branch), it is simply added to `tot`. If `element` is a list, then `r_sum` is called again, with the element as an argument. The statement inside the function definition in which the function calls itself is known as the **recursive call**.

The example above has a **base case** (on line 13) which does not lead to a recursive call: the case where the element is not a (sub-) list. Without a base case, you'll have **infinite recursion**, and your program will not work.

Recursion is truly one of the most beautiful and elegant tools in computer science.

A slightly more complicated problem is finding the largest value in our nested number list:

```

1  def r_max(nxs):
2      """
3      Find the maximum in a recursive structure of lists
4      within other lists.
5      Precondition: No lists or sublists are empty.
6      """
7      largest = None
8      first_time = True
9      for e in nxs:
10         if type(e) == type([]):
11             val = r_max(e)
12         else:
13             val = e
14
15         if first_time or val > largest:
16             largest = val
17             first_time = False
18
19     return largest
20
21 test(r_max([2, 9, [1, 13], 8, 6]) == 13)
22 test(r_max([2, [[100, 7], 90], [1, 13], 8, 6]) == 100)
23 test(r_max([[13, 7], 90], 2, [1, 100], 8, 6]) == 100)
24 test(r_max(["joe", ["sam", "ben"]]) == "sam")

```

Tests are included to provide examples of `r_max` at work.

The added twist to this problem is finding a value for initializing `largest`. We can't just use `nxs[0]`, since that could be either a element or a list. To solve this problem (at every recursive call) we initialize a Boolean flag (at line 8). When we've found the value of interest, (at line 15) we check to see whether this is the initializing (first) value for `largest`, or a value that could potentially change `largest`.

Again here we have a base case at line 13. If we don't supply a base case, Python stops after reaching a maximum recursion depth and returns a runtime error. See how this happens, by running this little script which we will call `infinite_recursion.py`:

```

1  def recursion_depth(number):
2      print("{0}, ".format(number), end="")
3      recursion_depth(number + 1)
4
5  recursion_depth(0)

```

After watching the messages flash by, you will be presented with the end of a long traceback that ends

with a message like the following:

```
RuntimeError: maximum recursion depth exceeded ...
```

We would certainly never want something like this to happen to a user of one of our programs, so in the next chapter we'll see how errors, any kinds of errors, are handled in Python.

17.6. Case study: Fibonacci numbers

The famous **Fibonacci sequence** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, ... was devised by Fibonacci (1170–1250), who used this to model the breeding of (pairs) of rabbits. If, in generation 7 you had 21 pairs in total, of which 13 were adults, then next generation the adults will all have bred new children, and the previous children will have grown up to become adults. So in generation 8 you'll have 13+21=34, of which 21 are adults.

This *model* to explain rabbit breeding made the simplifying assumption that rabbits never died. Scientists often make (unrealistic) simplifying assumptions and restrictions to make some headway with the problem.

If we number the terms of the sequence from 0, we can describe each term recursively as the sum of the previous two terms:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2) for n >= 2
```

This translates very directly into some Python:

```
1 def fib(n):
2     if n <= 1:
3         return n
4     t = fib(n-1) + fib(n-2)
5     return t
```

This is a particularly inefficient algorithm, and we'll show one way of fixing it when we learn about dictionaries:

```
1 import time
2 t0 = time.clock()
3 n = 35
4 result = fib(n)
5 t1 = time.clock()
6
7 print("fib({0}) = {1}, ({2:.2f} secs)".format(n, result, t1-t0))
```

We get the correct result, but an exploding amount of work!

```
fib(35) = 9227465, (10.54 secs)
```

17.7. Example with recursive directories and files

The following program lists the contents of a directory and all its subdirectories.

```
1 import os
2
3 def get_dirlist(path):
4     """
5     Return a sorted list of all entries in path.
6     This returns just the names, not the full path to the names.
7     """
```

```

8     dirlist = os.listdir(path)
9     dirlist.sort()
10    return dirlist
11
12    def print_files(path, prefix = ""):
13        """ Print recursive listing of contents of path """
14        if prefix == "": # Detect outermost call, print a heading
15            print("Folder listing for", path)
16            prefix = "| "
17
18        dirlist = get_dirlist(path)
19        for f in dirlist:
20            print(prefix+f)           # Print the line
21            fullname = os.path.join(path, f) # Turn name into full pathname
22            if os.path.isdir(fullname):     # If a directory, recurse.
23                print_files(fullname, prefix + "| ")

```

Calling the function `print_files` with some folder name will produce output similar to this:

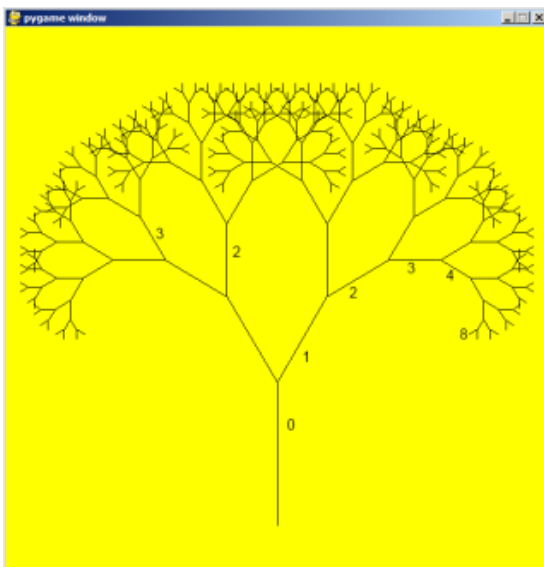
```

Folder listing for c:\python31\Lib\site-packages\pygame\examples
| __init__.py
| aacircle.py
| aliens.py
| arraydemo.py
| blend_fill.py
| blit_blends.py
| camera.py
| chimp.py
| cursors.py
| data
| | alien1.png
| | alien2.png
| | alien3.png
...

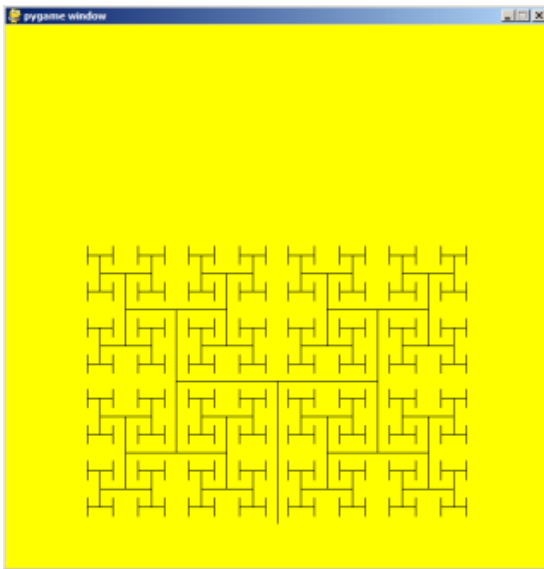
```

17.8. An animated fractal, using PyGame

Here we have a tree fractal pattern of order 8. We've labelled some of the edges, showing the depth of the recursion at which each edge was drawn.



In the tree above, the angle of deviation from the trunk is 30 degrees. Varying that angle gives other interesting shapes, for example, with the angle at 90 degrees we get this:



An interesting animation occurs if we generate and draw trees very rapidly, each time varying the angle a little. Although the Turtle module can draw trees like this quite elegantly, we could struggle for good frame rates. So we'll use PyGame instead, with a few embellishments and observations. (Once again, we suggest you cut and paste this code into your Python environment.)

```

1  import pygame, math
2  pygame.init()           # prepare the pygame module for use
3
4  # Create a new surface and window.
5  surface_size = 1024
6  main_surface = pygame.display.set_mode((surface_size,surface_size))
7  my_clock = pygame.time.Clock()
8
9
10 def draw_tree(order, theta, sz, posn, heading, color=(0,0,0), depth=0):
11
12     trunk_ratio = 0.29    # How big is the trunk relative to whole tree?
13     trunk = sz * trunk_ratio # length of trunk
14     delta_x = trunk * math.cos(heading)
15     delta_y = trunk * math.sin(heading)
16     (u, v) = posn
17     newpos = (u + delta_x, v + delta_y)
18     pygame.draw.line(main_surface, color, posn, newpos)
19
20     if order > 0:         # Draw another layer of subtrees
21
22         # These next six lines are a simple hack to make the two major halves
23         # of the recursion different colors. Fiddle here to change colors
24         # at other depths, or when depth is even, or odd, etc.
25         if depth == 0:
26             color1 = (255, 0, 0)
27             color2 = (0, 0, 255)
28         else:
29             color1 = color
30             color2 = color
31
32         # make the recursive calls to draw the two subtrees
33         newsz = sz*(1 - trunk_ratio)
34         draw_tree(order-1, theta, newsz, newpos, heading-theta, color1, depth+1)
35         draw_tree(order-1, theta, newsz, newpos, heading+theta, color2, depth+1)
36
37
38 def gameloop():
39
40     theta = 0
41     while True:
42
43         # Handle events from keyboard, mouse, etc.
44         ev = pygame.event.poll()
45         if ev.type == pygame.QUIT:
46             break;

```



```

47
48     # Updates - change the angle
49     theta += 0.01
50
51     # Draw everything
52     main_surface.fill((255, 255, 0))
53     draw_tree(9, theta, surface_size*0.9, (surface_size//2, surface_size-50), -math.pi/2)
54
55     pygame.display.flip()
56     my_clock.tick(120)
57
58
59 gameloop()
60 pygame.quit()

```

- The `math` library works with angles in radians rather than degrees.
- Lines 14 and 15 uses some high school trigonometry. From the length of the desired line (`trunk`), and its desired angle, `cos` and `sin` help us to calculate the `x` and `y` distances we need to move.
- Lines 22–30 are unnecessary, except if we want a colorful tree.
- In the main game loop at line 49 we change the angle on every frame, and redraw the new tree.
- Line 18 shows that PyGame can also draw lines, and plenty more. Check out the documentation. For example, drawing a small circle at each branch point of the tree can be accomplished by adding this line directly below line 18:

```

1  pygame.draw.circle(main_surface, color, (int(posn[0]), int(posn[1])), 3)

```

Another interesting effect — instructive too, if you wish to reinforce the idea of different instances of the function being called at different depths of recursion — is to create a list of colors, and let each recursive depth use a different color for drawing. (Use the depth of the recursion to index the list of colors.)

17.9. Glossary

base case

A branch of the conditional statement in a recursive function that does not give rise to further recursive calls.

infinite recursion

A function that calls itself recursively without ever reaching any base case. Eventually, infinite recursion causes a runtime error.

recursion

The process of calling a function that is already executing.

recursive call

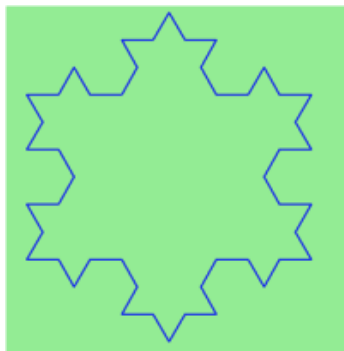
The statement that calls an already executing function. Recursion can also be indirect — function *f* can call *g* which calls *h*, and *h* could make a call back to *f*.

recursive definition

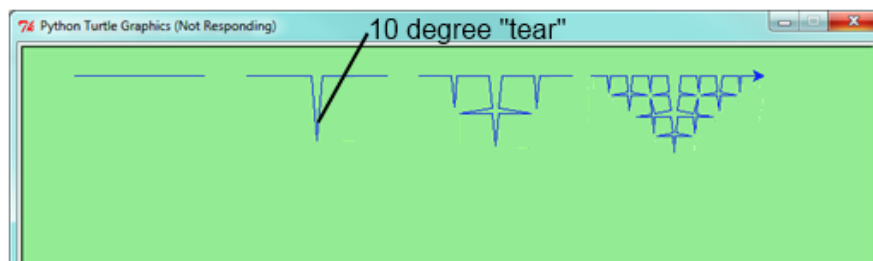
A definition which defines something in terms of itself. To be useful it must include *base cases* which are not recursive. In this way it differs from a *circular definition*. Recursive definitions often provide an elegant way to express complex data structures, like a directory that can contain other directories, or a menu that can contain other menus.

17.10. Exercises

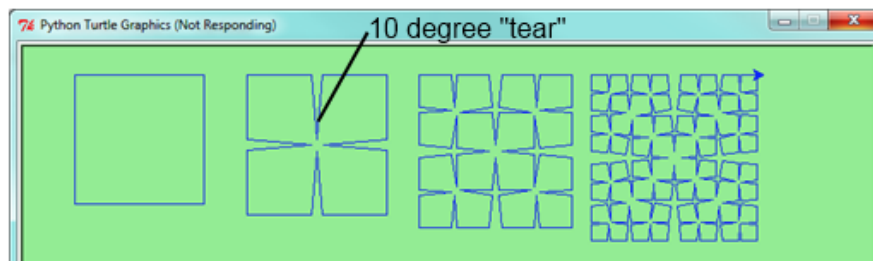
1. Modify the Koch fractal program so that it draws a Koch snowflake, like this:



2. a. Draw a Cesaro torn line fractal, of the order given by the user. We show four different lines of orders 0,1,2,3. In this example, the angle of the tear is 10 degrees.



- b. Four lines make a square. Use the code in part a) to draw cesaro squares. Varying the angle gives interesting effects — experiment a bit, or perhaps let the user input the angle of the tear.



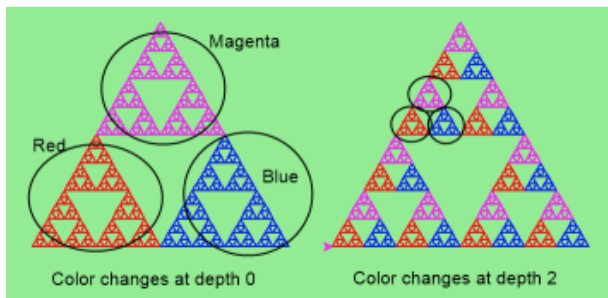
- c. (For the mathematically inclined). In the squares shown here, the higher-order drawings become a little larger. (Look at the bottom lines of each square – they're not aligned.) This is because we just halved the drawn part of the line for each recursive subproblem. So we've "grown" the overall square by the width of the tear(s). Can you solve the geometry problem so that the total size of the subproblem case (including the tear) remains exactly the same size as the original?

3. A Sierpinski triangle of order 0 is an equilateral triangle. An order 1 triangle can be drawn by drawing 3 smaller triangles (shown slightly disconnected here, just to help our understanding). Higher order 2 and 3 triangles are also shown. Draw Sierpinski triangles of any order input by the user.



4. Adapt the above program to change the color of its three sub-triangles at some depth of recursion. The illustration below shows two cases: on the left, the color is changed at depth 0 (the outmost level of recursion), on the right, at depth 2. If the user supplies a negative depth, the

color never changes. (Hint: add a new optional parameter `colorChangeDepth` (which defaults to `-1`), and make this one smaller on each recursive subcall. Then, in the section of code before you recurse, test whether the parameter is zero, and change color.)



5. Write a function, `recursive_min`, that returns the smallest value in a nested number list. Assume there are no empty lists or sublists:

```
test(recursive_min([2, 9, [1, 13], 8, 6]) == 1)
test(recursive_min([2, [[100, 1], 90], [10, 13], 8, 6]) == 1)
test(recursive_min([2, [[13, -7], 90], [1, 100], 8, 6]) == -7)
test(recursive_min([[[-13, 7], 90], 2, [1, 100], 8, 6]) == -13)
```

6. Write a function `count` that returns the number of occurrences of `target` in a nested list:

```
test(count(2, []), 0)
test(count(2, [2, 9, [2, 1, 13, 2], 8, [2, 6]]) == 4)
test(count(7, [[9, [7, 1, 13, 2], 8], [7, 6]]) == 2)
test(count(15, [[9, [7, 1, 13, 2], 8], [2, 6]]) == 0)
test(count(5, [[5, [5, [1, 5], 5], 5], [5, 6]]) == 6)
test(count("a",
    ["this", "a", ["thing", "a"], "a"], "is", ["a", "easy"]]) == 4)
```

7. Write a function `flatten` that returns a simple list containing all the values in a nested list:

```
test(flatten([2, 9, [2, 1, 13, 2], 8, [2, 6]]) == [2, 9, 2, 1, 13, 2, 8, 2, 6])
test(flatten([[9, [7, 1, 13, 2], 8], [7, 6]]) == [9, 7, 1, 13, 2, 8, 7, 6])
test(flatten([9, [7, 1, 13, 2], 8], [2, 6]) == [9, 7, 1, 13, 2, 8, 2, 6])
test(flatten(["this", ["a", ["thing"], "a"], "is"], ["a", "easy"]]) ==
    ["this", "a", "thing", "a", "is", "a", "easy"])
test(flatten([]) == [])
```

8. Rewrite the fibonacci algorithm without using recursion. Can you find bigger terms of the sequence? Can you find `fib(200)`?
9. Use help to find out what `sys.getrecursionlimit()` and `sys.setrecursionlimit(n)` do. Create several experiments similar to what was done in `infinite_recursion.py` to test your understanding of how these module functions work.
10. Write a program that walks a directory structure (as in the last section of this chapter), but instead of printing filenames, it returns a list of all the full paths of files in the directory or the subdirectories. (Don't include directories in this list — just files.) For example, the output list might have elements like this:

```
["C:\\Python31\\Lib\\site-packages\\pygame\\docs\\ref\\mask.html",
 "C:\\Python31\\Lib\\site-packages\\pygame\\docs\\ref\\midi.html",
 ...
 "C:\\Python31\\Lib\\site-packages\\pygame\\examples\\aliens.py",
 ...
 "C:\\Python31\\Lib\\site-packages\\pygame\\examples\\data\\boom.wav",
 ... ]
```

11. Write a program named `litter.py` that creates an empty file named `trash.txt` in each subdirectory of a directory tree given the root of the tree as an argument (or the current directory as a default). Now write a program named `cleanup.py` that removes all these files.

Hint #1: Use the program from the example in the last section of this chapter as a basis for these two recursive programs. Because you're going to destroy files on your disks, you better get this right, or you risk losing files you care about. So excellent advice is that initially you should fake the deletion of the files — just print the full path names of each file that you intend to delete. Once you're happy that your logic is correct, and you can see that you're not deleting the wrong things, you can replace the print statement with the real thing.

Hint #2: Look in the `os` module for a function that removes files.