# Demystifying Recursion:

Michelle Berry Follow
Mar 31, 2017 · 7 min read

I recently graduated from Hackbright Academy's Software Engineering Fellowship for women. In 10 weeks, the bootcamp teaches you how to build a functioning web application (Check out the apps built by my amazing colleagues!) and it also teaches you fundamental CS topics like data structures, algorithms, and recursion. This was a lot of information to cover in a short amount of time, and the topic that I and others seemed to struggle with the most was recursion.

> *Nothing strikes greater fear in a recent bootcamp grad than a request to code a recursive problem at the whiteboard.*

I wanted to write this blog post to share the insights I've had while learning recursion and hopefully demystify its seemingly magical properties. This blog will be most helpful for learners who have tried a few recursive problems on their own, but don't feel comfortable with it yet.

First, what is recursion? **It's a function that calls itself**. That's it. Don't fall into the trap of thinking recursion necessarily eliminates for/while loops. Although this is frequently true, there are plenty of recursive functions that still implement loops (more on this later). In fact, because recursion has such a broad definition, there are frequently many different recursive strategies for solving the same problem.

Because a recursive function calls itself, it needs a **base case**. Otherwise, the function would call itself for ever. Eventually you will run out of memory on your stack (the part of your computer's memory that's allocated for function calls), and this is called **stack overflow.** Many recursive functions start with an ***if*** statement that checks for the base case. However, not every recursive function has such an obvious or explicit base case.

## My thesis:

Most of us cannot keep track of the full call-stack in our head. Therefore, recursion, like most engineering problems, is about **pattern matching**. This means that with exposure to different recursive patterns and enough practice, *anybody* can do it well. Often, instructors and interviewers encourage you to "think about the base case and the recursive case", as if the intuition behind those things should be obvious. Well it's not! That is, *until* you've had enough practice with different recursive patterns. I will now demonstrate several common recursive patterns with examples in Python.

The first three patterns solve the same problem: counting the number of items in a list. The easiest way to solve this problem is to use the python built-in *len()* method for lists, but let's assume we don't have that at our disposal. We could solve this problem iteratively:

```
1  def count(lst):
2      count = 0
3      for item in lst:
4          count += 1
5      return count
```

Now let's think about how to solve this problem recursively. The first thing I want to emphasize is that whenever we work with iterables like lists or strings in a recursive function, we probably want to use **slicing** to make progress. Think about it, how can we break down a problem involving a long iterable into an easier problem? Break it down into successively smaller iterables until we reach a trivial case—e.g. an empty list or empty string.

More importantly, there isn't only one solution to this. I will now highlight three different recursive patterns to solve this problem.

```python
# Solution 1
def count_rec1(lst):
    if not lst:
        return 0
    return 1 + count_rec1(lst[1:])

# Solution 2
def count_rec2(lst, count=0):
    if not lst:
        return count
    return count_rec2(lst[1:], count+1)

# Solution 3
def count_rec3(lst):
```

## Pattern 1: The result is built-up in the return statement

The first solution works by adding the **count**, along with the sliced list, in the return statement. Essentially, it adds 1 every time we recurse, and we recurse n-times.

If our list was [1, 2, 3], our function calls would look like this:

```
1 + [2, 3]
1 + 1 + [3]
1 + 1 + 1 + []
1 + 1 + 1 + 0
```

The only nuance here is the **base case.** Our base case is when we reach an empty list. We need to return something from this call, otherwise we never stop recursing. But what? It *must* be an integer, because if we return something else like *None* or an empty list, we will raise an error when we try to add it to our other function calls. Therefore, we return zero.

This recursive pattern is good for solving problems that we could otherwise solve by iterating with a for-loop and appending or adding something at each step. In the same way you might think about looping over an iterable, think about returning [the result of the current item]

+ [the function called on the rest of the iterable]. Some other problems that easily lend themselves to this same pattern are **summing the numbers in a list** and **reversing a string**.

## Pattern 2: The return value is a function argument

In solution 2, we pass the thing we want to return—the count variable—as an argument to our function. This pattern is an example of something called **tail recursion**, and this argument has a special name, the **accumulator**. What distinguishes tail recursion from traditional recursion, is that the final function-call contains everything that's needed to make the final return statement. In this example, we return the value of the count variable at the last call, and we disregard values from the previous function calls. Accumulator variables are common in tail recursive solutions, but they are not strictly necessary.

In some languages, like Scheme and JavaScript, tail recursive solutions have a memory optimization advantage over traditional recursive solutions. Think about it, we can forget about all of the other functions being stored in the call stack and only retain the most recent call in memory. However, Python is a language that doesn't allow for this optimization, because its creator, Guido Van Rossum, preferred it to have proper function tracebacks for debugging purposes.

In this example, our function calls for [1, 2, 3] looks like this:

```
count_recr2([2,3], count=1)

count_recr2([3], count=2)

count_recr2([], count=3)
```

## Pattern 3: Accumulate the count in a variable, aka WTF???

This particular pattern is a real mind-bender. Doesn't the count variable get reset to 1 every time you call the function? Yes, yes it does. But herein lies the magic of recursion. Every recursive call is a separate function call, and as such, it maintains its own set of local variables. This means that for the list [1, 2, 3], there are three different frames on

the call stack, each with a variable called count, set to the value of 1. When we reach our base case, we return zero. Then, our function at the top of the stack returns 1 to the previous function and so on, until all of the counts have accumulated into our count variable from our first function call. I'm still incredulous about this one, but it helps to visualize it in the **python visualizer**.

This solution is essentially the same as **solution 1**, except the count is stored in a variable before it is returned. However, I wanted to use this example to illustrate how variable scope works within a recursive function. It also appears in the next example. . .

## Pattern 4: When recursion meets iteration

Another confusing pattern for me, personally, was recursive functions that included iterative loops. This pattern comes up when you have some sort of nesting. In the example below, we want to count the number of nodes in a tree. A node's children are stored in a list, so we need to iterate through each of the children. Then, for each child, we need to traverse and count their respective children.

```
class Node(object):
    """Node in a tree."""

    def __init__(self, name, children=None):
        self.name = name
        self.children = children or []

def count_nodes(node):

    # Handles empty node input
    if not node:
        return 0

    count = 1
```

Notice, this code very closely resembles **Solution 3** from above when we counted items in a list.

One important thing to watch out for when using a loop inside of a recursive function is the return statement. If we return inside of a for-

loop, it won't complete the loop. This is why patterns 1 and 2 don't work well with this example. Just remember, don't return inside of a for loop unless you've hit your final "win" or "loss" case. Other coding problems that use this pattern are depth-first-search and generating permutations.

## Pattern 5: inner recursive functions

This pattern seemed like cheating when I first saw it, but it's a totally valid solution. The idea is to define an inner function that's recursive *inside* of a non-recursive function. The convention is to name this inner function the same thing as the outer function, but to include an underscore at the beginning to indicate that this is a helper function that should only be used locally.

This pattern is useful when you need to traverse a data structure *and* then do something else that depends on the object in the initial call. The traversal is easy to code recursively, and the last steps can be defined in the outer, non-recursive function.

The example I'll use here is reversing a linked list in place. We traverse our linked list and swap the direction of each arrow with every recursive call. However, when we've reached the end of our list, we need to update the original list's head. Why can't we do this in our recursive function? Because of object orientation. If we were to solve this problem with one recursive function, we would be passing in a new instance of a linked list every time. When we would get to the end of our linked list, there would be no way to refer back to the initial instance and change it's head.

```python
class Node:
    """Node class for linked list"""

    def __init__(self, data, next=None):
        self.data = data
        self.next = next

class LinkedList:
    """LinkedList class"""

    def __init__(self, head=None):
        self.head = head

    def reverse(self):
        """Reverse the list in place"""

        def _reverse(curr, prev):
            if not curr:
```

## Conclusion:

Becoming familiarized with these and other recursive patterns will make solving recursion problems easier. For any given recursion problem, think about which pattern it most resembles. Keep in mind that there are often multiple patterns that could work. I've found that the best way to practice recursion is to **solve a problem iteratively, then solve it recursively, and then try to solve it again with another recursive pattern**. In fact, I've learned the most from lengthy reflection on why certain patterns fail for certain problems. This way, you can test the boundaries of each pattern. Good luck and happy recursing!