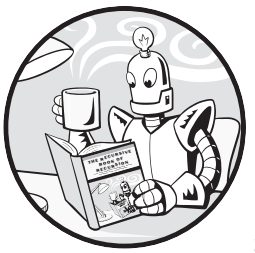
1

WHAT IS RECURSION?



Recursion has an intimidating reputation. It's considered hard to understand, but at its core, it depends on only two things: function calls and stack data structures.

Most new programmers trace through what a program does by following the execution. It's an easy way to read code: you just put your finger on the line of code at the top of the program and move down. Sometimes your finger will loop back; other times, it will jump into a function and later return. This makes it easy to visualize what a program does and in what order.

But to understand recursion, you need to become familiar with a less obvious data structure, called a *stack*, that controls the program's flow of execution. Most programming beginners don't know about stacks, because programming tutorials often don't even mention them when discussing function calls. Furthermore, the call stack that automatically manages function calls doesn't appear anywhere in the source code.

It's hard to understand something when you can't see it and don't know it exists! In this chapter, we'll pull back the curtain to dispel the overblown notion that recursion is hard, and you'll be able to appreciate the elegance underneath.

The Definition of Recursion

Before we discuss recursion, let's get the clichéd recursion jokes out of the way, starting with this: "To understand recursion, you must first understand recursion."

During the months I've spent writing this book, I can assure you that this joke gets funnier the more you hear it.

Another joke is that if you search Google for *recursion*, the results page asks if you mean *recursion*. Following the link, as shown in Figure 1-1, takes you to . . . the search results for *recursion*.

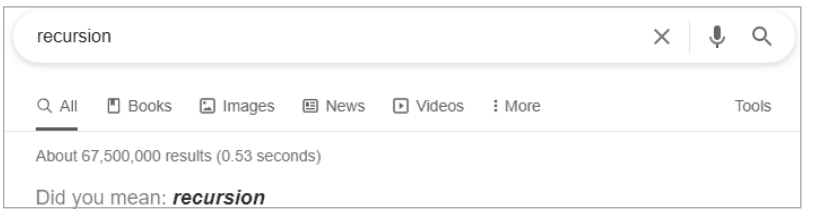


Figure 1-1: The Google search results for recursion *link to the Google search results for* recursion.

Figure 1-2 shows a recursion joke from the webcomic xkcd.

Figure 1-2: I'm So Meta, Even This Acronym (I.S. M.E.T.A.)

Most jokes about the 2010 science-fiction action movie *Inception* are recursion jokes. The film features characters having dreams within dreams within dreams.

And finally, what computer scientist could forget that monster from Greek mythology, the recursive centaur? As you can see in Figure 1-3, it is half horse, half recursive centaur.

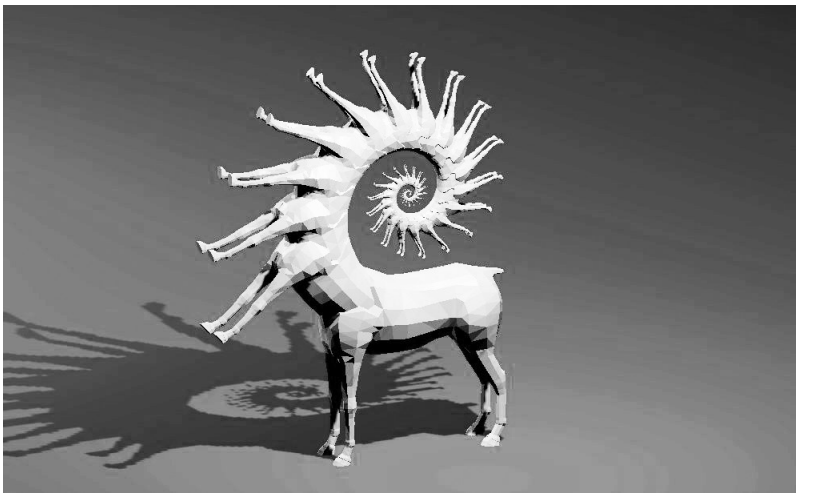


Figure 1-3: The recursive centaur. Image by Joseph Parker.

Based on these jokes, you might conclude that recursion is a sort of meta, self-referencing, dream-within-a-dream, infinite mirror-into-mirror sort of thing. Let's establish a concrete definition: a *recursive* thing is something whose definition includes itself. That is, it has a self-referential definition.

The Sierpiński triangle in Figure 1-4 is defined as an equilateral triangle with an upside-down triangle in the middle that forms three new equilateral triangles, each of which contains a Sierpiński triangle. The definition of Sierpiński triangles includes Sierpiński triangles.

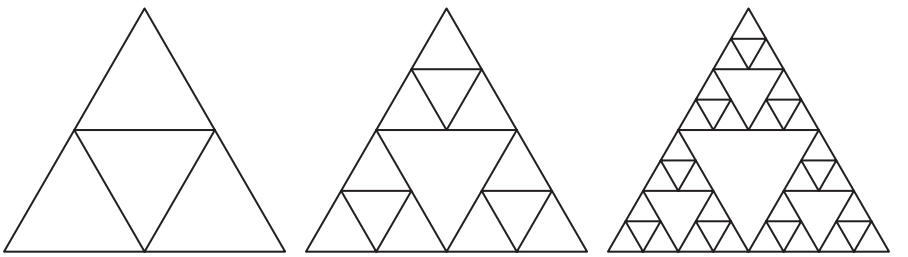


Figure 1-4: Sierpiński triangles are fractals (recursive shapes) that include Sierpiński triangles.

In a programming context, a *recursive function* is a function that calls itself. Before we explore recursive functions, let's take a step back and understand how regular functions work. Programmers tend to take function calls for granted, but even experienced programmers will find it worthwhile to review functions in the next section.

What Are Functions?

Functions can be described as mini-programs inside your program. They're a feature of nearly every programming language. If you need to run identical

instructions at three different places in a program, instead of copying and pasting the source code three times, you can write the code in a function once and call the function three times. The beneficial result is a shorter and more readable program. The program is also easier to change: If you need to fix a bug or add features, you need to change your program in only one place instead of three.

All programming languages implement four features in their functions:

1. Functions have code that is run when the function is called.
2. *Arguments* (that is, values) are passed to the function when it's called. This is the input to the function, and functions can have zero or more arguments.
3. Functions return a *return value*. This is the output of the function, though some programming languages allow functions to not return anything or return null values like `undefined` or `None`.
4. The program remembers which line of code called the function and returns to it when the function finishes its execution.

Different programming languages might have additional features, or different options for how to call functions, but they all have these four general elements. You can visually see the first three of these elements because you write them in the source code, but how does a program keep track of where the execution should return to when the function returns?

To get a better sense of the problem, create a *functionCalls.py* program that has three functions: `a()`, which calls `b()`, which calls `c()`:

Python

```
def a():
    print('a() was called.')
  ❶ b()
    print('a() is returning.')

def b():
    print('b() was called.')
  ❷ c()
    print('b() is returning.')

def c():
    print('c() was called.')
    print('c() is returning.')

a()
```

This code is equivalent to the following *functionCalls.html* program:

JavaScript

```
<script type="text/javascript">
function a() {
    document.write("a() was called.<br />");
  ❶ b();
    document.write("a() is returning.<br />");
}
```

```
function b() {
    document.write("b() was called.<br />");
  ❷ c();
    document.write("b() is returning.<br />");
}

function c() {
    document.write("c() was called.<br />");
    document.write("c() is returning.<br />");
}

a();
</script>
```

When you run this code, the output looks like this:

```
a() was called.
b() was called.
c() was called.
c() is returning.
b() is returning.
a() is returning.
```

The output shows the start of functions `a()`, `b()`, and `c()`. Then, when the functions return, the output appears in reverse order: `c()`, `b()`, and then `a()`. Notice the pattern to the text output: each time a function returns, it remembers which line of code originally called it. When the `c()` function call ends, the program returns to the `b()` function and displays `b() is returning.`. Then the `b()` function call ends, and the program returns to the `a()` function and displays `a() is returning.`. Finally, the program returns to the original `a()` function call at the end of the program. In other words, function calls don't send the execution of the program on a one-way trip.

But how does the program remember if it was `a()` ❶ or `b()` ❷ that called `c()`? This detail is handled by the program implicitly with a call stack. To understand how call stacks remember where the execution returns at the end of a function call, we need to first understand what a stack is.

What Are Stacks?

Earlier I mentioned the clichéd wisecrack, "To understand recursion, you must first understand recursion." But this is actually wrong: to really understand recursion, you must first understand stacks.

A *stack* is one of the simplest data structures in computer science. It stores multiple values like a list does—but unlike lists, it limits you to adding to or removing values from the "top" of the stack only. For lists, the "top" is the last item, at the right end of the list. Adding values is called *pushing* values onto the stack, while removing values is called *popping* values off the stack.

Imagine that you're engaged in a meandering conversation with someone. You're talking about your friend Alice, which then reminds you of a