Pre-Lecture 3: Recursion

In some sense, this week's lecture represents a slight shift of focus as we move on to a new topic. The ideas we've been talking about so far won't go away, of course, but we will have a different central focus for the next few weeks: recursion.

We expect that recursion is something that everyone has seen before in 6.0001, but recursion can feel somewhat weird and uncomfortable until you build up some experience working it and some understanding of how it works. So we're going to spend a fair amount of time over the next couple of weeks with that goal in mind: developing facility and comfort with, and a deeper understanding of, recursion. As usual, we'll come at this from a couple of different perspectives: we'll certainly look at how we can use recursion in our own programs, but we'll also focus on how recursion works behind the scenes (in terms of our environment model), as well as talking about how we can go about deciding whether a recursive solution is appropriate for a given problem (or whether some other approach might be better).

And don't worry if recursion feels strange or uncomfortable; that's a natural first reaction, and with time and practice (which we're aiming to provide over the course of the next several lectures, recitations, and labs), this idea will stop being quite so scary and start being a really powerful tool that we can use in our own programs.

Definitions

Before we can get too far into talking about using recursion, it's worth briefly talking about what recursion is. Generally speaking, **recursion** occurs when something is defined in terms of itself.

Although our focus will be on recursion from a programming perspective, recursion can also appear in other contexts. Here is an example from mathematics, a definition of the factorial operation. For nonnegative integer n,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

In general, a recursive definition will have multiple pieces:

- one or more **base cases** (terminating scenarios that do not need recursion to produce answers), and
- one or more **recursive cases** (sets of rules that reduce all other cases toward base cases)

In our example above, the case where $n = 0$ is a base case. We don't need recursion to find the answer there, since $0! = 1$ by definition.

The other case is our recursive case. We can see the recursion here in that figuring out that value depends on the definition of factorial. We can look at an example of this process in this mathematical context before we move on to programming. Let's consider determining $(4!)$ using the definition above. We can plug things into that definition above and use something like the following sequence of steps to arrive at an answer, repeatedly applying the definition of factorial until we reach a base case:

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times 3 \times 2! \\ &= 4 \times 3 \times 2 \times 1! \\ &= 4 \times 3 \times 2 \times 1 \times 0! \\ &= 4 \times 3 \times 2 \times 1 \times 1 \quad = 24 \end{aligned}$$

In this example, we can also see the important property here that the recursive case reduces us down toward a base case, in the sense that for any value n we want to compute the factorial of, the $(n-1)!$ is applying the factorial definition to a value that is closer to our base case of $n = 0$. In this way, we can rest assured that for any nonnegative integer n, this process will eventually reach a base case, at which point we can find our overall answer.

Programming

It turns out that Python lets us define functions recursively (i.e., we can define functions in terms of themselves). As an example, here is a Python implementation of the definition of factorial from above (note that it is almost an exact translation):

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

We could try to write this more concisely, but this is a nice form to demonstrate how this definition parallels the mathematical definition from above. There are some key things to notice here:

1. This is a recursive definition, since computing `factorial` depends on the definition of `factorial` (i.e., the function calls *itself* in the process of computing its result in some cases).
2. We have one base case. When `n == 0`, we can compute the result without any recursion.
3. We have one recursive case, which reduces down toward a base case.

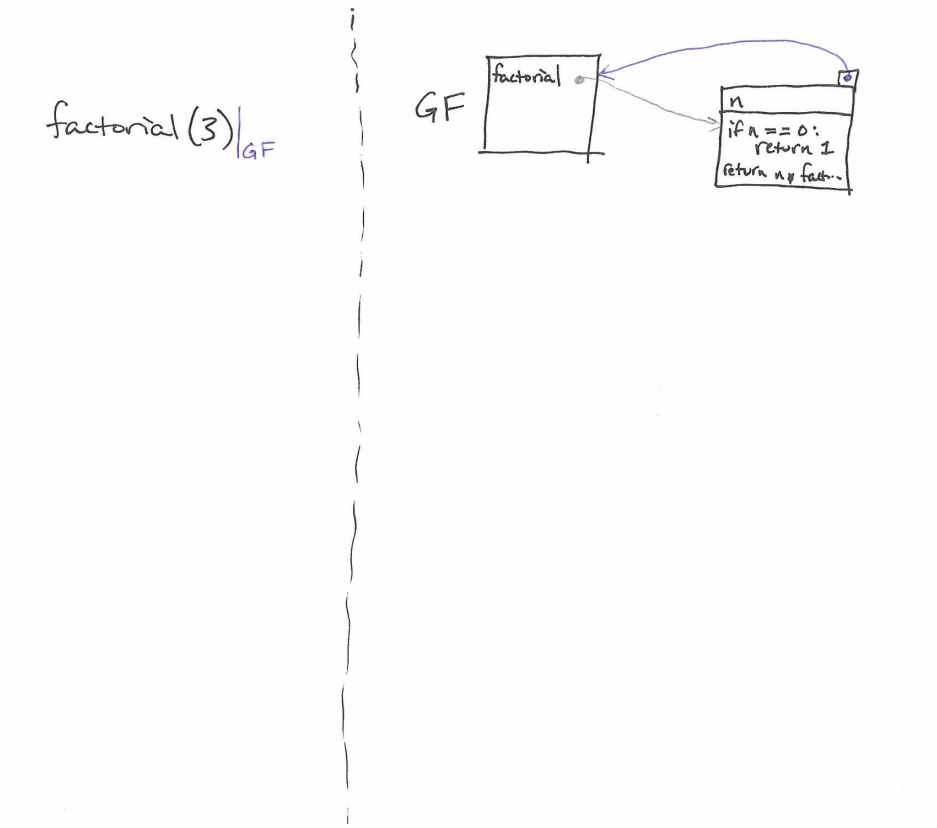
Before we move on to the details of how this works in Python, it's worth mentioning again that, depending on the extent of your prior exposure to these ideas, it may feel uncomfortable writing a function like this. And a big part of the weirdness here is that as we're writing `factorial`, we're using `factorial` as part of the body, despite the fact that *we've not finished writing `factorial` yet!* But the right strategy is to write your recursive case(s) *under the assumption that you have a complete, working version of the function*, and to think about, under those conditions, how would I take the result from that recursive call and combine it with other things to produce the result I'm interested in? And we don't need to have blind faith here; if we've designed things such that we have our base cases set up, and such that our recursive calls are working down toward one of those bases cases, then things will work out for us.

Another source of weirdness here is that in the process of evaluating some call to `factorial`, I'm going to need to call `factorial` again and maybe many times; and each of these calls has its own value of `n`. So, how can we be sure that Python is going to keep those things separate and not get confused?

Recursion in the Environment Model

Importantly, Python *is* able to keep those things separate, and its ability to do so is a natural effect of our normal rules for function evaluation (that is, Python does not treat recursive functions any differently from nonrecursive functions). Let's take a look at how this plays out using an environment diagram below, for evaluating `factorial(3)`. To start with, **only proceed through step 5 below, and then answer the following question:**

Moving from step 5 to step 6 in the environment diagram below, where will F_2's parent pointer go?



<< First Step | < Previous Step | Next Step > | Last Step >>

STEP 1

Here we have a diagram with two pieces: on the right-hand side, we have an environment diagram; and on the left-hand side, we have space to keep track of the expression we're evaluating. On the right-hand side, we see what the environment diagram looks like after executing the function definition from above, and on the left, we see the expression we are going to evaluate (`factorial(3)`), evaluated in the global frame (**GF**).

In order to perform this function call, we follow our usual steps. The first step is to figure out what function we're calling (in this case, evaluating the name `factorial` to find the function object on the right), as well as the arguments to the function (in this case, an integer `3`). The results are shown on the next diagram.

As a follow-on question, with factorial written as above, how many total frames (other than the global frame and the built-ins) are created when we call `factorial(7)`?

Primer: Recursion versus Iteration

Now, it's worth mentioning that, while the process above involved recursion, we could have computed factorials iteratively instead. Mathematically, we could have used either of the following definitions of factorial (note that the second one isn't recursive!):

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

or

$$n! = \prod_{i=1}^{n} i$$

and, in terms of code, we could have written either of the following (note that the second is not recursive, though it computes the same result!):

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

```
def factorial(n):
    out = 1
    for i in range(1, n+1):
        out *= i
    return out
```

> **Check Yourself:**
>
> Before lecture, try to think about these two programs as written (and maybe even try them out!). What are the benefits and drawbacks of each approach?

See you all in lecture on Monday! Of course, if you have questions about any of these notes or examples before then, please let us know!