recursion-note

February 14, 2019

1 Recursion, a worked example [beta]

Note written for LiU course 732A74 by Anders Märak Leffler in 2019. The author welcomes comments and questions.

1.1 Introduction.

Recursion is the main way of creating repeated operations in functional programming. It consists of functions calling themselves, instead of using iterative constructs such as `while`- or `for`-loops which implicity require some kind of state.

This is generally a more mathematical way of defining operations and many famous mathematical functions, such as the Fibonacci function, have a recursive definition, even though they usually can be implemented iteratively. Below we will use a toy example as a way of introducing linear recursion. This might be of particular interest if you haven't taken undergraduate programming (or theory of computation).

Although there are some self-study questions below, you do not need to hand in this notebook.

1.1.1 Taking a very imperative view (to start with)

Since it is likely that most students who haven't already seen this have an imperative view of these calculations, this note will start "top down" with a somewhat practical way of writing linearly recursive functions. This is only to provide an intuition. We will then return to the "proper" mathematical identities.

1.1.2 Is this useful knowledge outside functional languages?

We will be using a toy example, and a favourite of introductory courses. More practical examples of recursive functions or recursive identities abound in eg network optimisation, traversing decision trees (in machine learning), analysing text structure, programs, and the like. It is not also unusual that a recursive identity is given a fast imperative implementation. So whereas Pythonic code seldom overuses explicit recursion, or pure recursive functions (plenty of recursive traversal over tree structures might take place, but not in a purely functional manner), it is useful to know.

1.2 A worked example

Consider the (fairly standard) example of calculating triangular numbers. That is, those satisfying `triangle(n) = 1 + 2 + ... + n`. The goal here is to for any allowed n, express `triangle(n)` in terms of `triangle(k)` for some smaller input, and in terms of some "smallest" input.

In the case of numbers it might seem reasonable to use "the previous number" as the smaller case. In general, we might have several possible choices. For instance, if we express the classification we get from a binary decision tree as an identity of this kind, we might have to consider subtrees as "the possible smaller case**s**". In the case of this recursive identity, we however have only one choice.

1.3 The trick of "magic" reference functions

We start writing the function:

```
In [ ]: def triangle(n):
            pass
```

How do we express the solution in terms of a smaller case? Let's initially suppose, for the sake of argument, that we had provided **a "magic" reference function** `triangle_reference` which would always give us the answer for smaller cases. (I refrain from using the word "oracle" here, as this might in some settings suggest that the function runs in constant time.)

It's not necessarily the case that we can write such a function, but happily for this note we can implement one which enables us to test the code in this case. Otherwise it would be a theoretical construct. From earlier lab work (or lecture notes) we might conclude that it would look somewhat like:

```
In [ ]: def triangle_reference(n):
            result = 0
            for k in range(n+1):
                result = result + k
            return result

        # Or equivalently
        def triangle_reference(n):
            return sum(range(n+1))
```

How would we then start to express `triangle(n)`? If we decided that the "smaller" case if we had an input of n was always the penultimate number $n-1$, it might look something like this:

```
In [ ]: # This code is not finished yet!
        def triangle(n):
            prev_result = triangle_reference(n-1)    # using magic reference function
            # If n=5, prev_result will - magically - be the correct result for n=4.
            # What remains is to extend this result and return it.
            return "something good here" # Return something.
```

We might extend this to cover the case n straight away if we see how.

In other cases, it might be useful to **write out a few simple cases**, and what they are supposed to return. We take a few examples:

`triangle(0) = 0`
`triangle(1) = 0 + 1 = 1`
`triangle(2) = 0 + 1 + 2 = 3`
`triangle(3) = 0 + 1 + 2 + 3 = 6`

If we knew the answer for $n = 2$ and wanted to extend this into an answer for $n = 3$, we see that the difference is adding 3. If we we knew the answer for $n = 1$ and wanted to extend this into an answer for $n = 2$ we would add 2. And so on.

If we have it written out we can sometimes (not always, but sometimes) match this up fairly easily:

`triangle(2) = 0 + 1 + 2 = 3`
`triangle(3) = 0 + 1 + 2 + 3 = 6`

So `triangle(3) = (0 + 1 + 2) + 3 = triangle(2) + 3 = 6`.

We can see that for $n > 0$ we would get the result by adding n. In our case, the code could now be extended to

```
In [ ]: # This code is not finished yet!
        def triangle(n):
            prev_result = triangle_reference(n-1)     # using magic reference function
            return prev_result + n
```

This still has a few issues, in terms of our solution. * It only says something about the cases where $n > 0$. $n < 0$ is irrelevant, but $n = 0$ might turn out to be useful. * Even more obvious is that it contains references to our "magic" reference function. We can't depend on such skyhooks. In particular, when we solve problems where we don't have a reference implementation, this will be impossible.

1.3.1 A base case

Let's say that we needed to cover $n = 0$ as well, and thus fix our "smallest" input. That is, we add a **base case**. The question then becomes what this should yield. Looking at it mathematically, it seems reasonable that the answer is 0. This can either be seen directly from the problem statement, or by noting that `triangle(1)` is considered to be 1 + the reference value for 0. If `triangle(1)` is supposed to return 1, then `triangle(0)` + 1 should be 1.

```
In [ ]: # This code is not finished yet!
        def triangle(n):
            if n == 0:
                return 0
            else:
                prev_result = triangle_reference(n-1)            # using magic reference funct
                return prev_result + n
```

1.3.2 Replacing the magic reference function with...

Now we take the scaffolding off, and replace it with our own function:

```
In [ ]: # This code is not finished yet!
        def triangle(n):
            if n == 0:
```

```
        return 0
    else:
        prev_result = triangle(n-1)            # <- Now using triangle!
        return prev_result + n
```

Since we use this only once, we can skip the `prev_result` binding altogether.

```
In [ ]: def triangle(n):
            if n == 0:
                return 0
            else:
                return triangle(n-1)  + n      # prev_result replaced by the expression.
```

Try it out. It seems to work.

```
In [ ]: triangle(0)
```

```
In [ ]: triangle(1)
```

```
In [ ]: triangle(5)
```

The leap of faith here is that we trust our `triangle` to calculate the value for $n-1$. We knew that our function would work if we were allowed to consult a magic reference function, but why does it make sense to trust our function now?

1.4 How does it work?

Why does it seem to work? It might be useful to follow the calls:

```
In [ ]: # Run this cell!

        def triangle(n):
            print("Hey, someone called triangle({})".format(n))
            if n == 0:
                return 0
            else:
                return triangle(n-1)  + n

        triangle(3)
```

This is useful, but it doesn't tell us what happens in terms of return values. How do we generate these?

We rewrite the structure slightly, to see what the function calls return.

```
In [ ]: # Run this cell!

        def triangle(n):
            print("Hey, someone called triangle({})".format(n))
            if n == 0:
                retval =0
```

```
        else:
            retval = triangle(n-1)  + n
        print("Now we're back in triangle({}), and (after possibly adding {}) return {}.".
        return retval

    triangle(3)
```

If we want this to be even more readable, consider the following code, which groups information from the same function call at the same indentation level:

```
In [ ]: # Run this cell!

    def triangle(n, indent=0):
        print(" "*indent, end="")   # Indent the printout for readability.
        print("Hey, someone called triangle({})".format(n))

        if n == 0:
            retval = 0
        else:
            retval = triangle(n-1, indent+1)  + n


        print(" "*indent, end="")   # Indent the printout for readability.
        print("Now we're back in triangle({}), and (after possibly adding {}) return {}.".
        return retval

    triangle(3)
```

Look at the call structure from the innermost call and out. Who called `triangle(0)`? What happened to the return value? What happened to that function call's return value, in turn? And so on.

[Press enter in this cell and remove the

1.4.1 What does it express?

Above we see operationally, in the working of the program, that `triangle(n) = triangle(n-1) + n` for $n > 0$ and `triangle(0) = 0`. This is actually just a rewriting in Python of the mathematical identity which defines triangle numbers:

$$triangle(n) = \begin{cases} triangle(n-1) + n & n > 0, \\ 0 & \text{for} \quad n = 0. \end{cases}$$

1.5 A note on Pythonisms

If `val` is in some sense an "empty" value in Python - the number 0, the empty list, the empty string, ... - we will by convention be able to detect it by `not val` being `True`. So the code above could be written

```
In [ ]: def triangle(n):
            if not n:
```

```
        return 0
    else:
        return triangle(n-1) + n
```

Or even simpler

```
In [ ]: def triangle(n):
    if not n:
        return n          # Note that one character changed here.
    else:
        return triangle(n-1) + n
```

1.6 Summing up the strategy

Above we see a few things of note: * A base case. Some smallest case (possibly several), here $n = 0$.

In many cases in Python this will be the `if not <input variable>` case (the empty list, string, ...). Here we have some fixed value.

- Recursive case(s).

Find out what the previous case(s) might be, make recursive call(s) and combine or extend the result somehow. As a strategy it might be helpful to make reference to a "magic" function until you're done with the code. Here the previous case was fairly obvious, but it might typically be "every element except the current one we're looking at", "the child nodes in the tree" or the like.

(We naturally don't need the reference function. We might as well have used `triangle` immediately. But the "assume that your function works for smaller values" step often turns out to be rather hard to grasp at first. Having a more or less concrete "magic" other function to outsource this part of the thinking to might help until one has got the hang of this.)

- Writing small example calculations on paper. Sometimes it is a surprisingly helpful tool to debug one's ideas before coding.
- Tracing function calls. Above we simply print the input values that a function has been called with, and the return values.

1.7 Matematical aside (not required)

This all depends on the structure we're working with having some useful order and some smallest element(s). It's possible to find a smallest natural number, a leaf in the tree or the like. Operationally, that means that we don't have to have infinite loops by definition. It is also useful - indeed required - to find a discrete "previous case" to extend (we do not end up with impossible questions such as "which is the next real number after π?").

Those who revel in the theory of different kinds of recursion and the theoretical underpinnings of computer science might be interested in the book "Computability: an introduction to recursive function theory" by Cutland. At the time of writing, it is available at the LiU library. This is quite outside the scope of this course.

1.7.1 Imperative style

Above we write the code in a fairly imperative style with `if` statements grouping sequences of instructions together (although the sequences in the end were turned into a single `return` in both branches). Since this is a mathematical identity where we always expect some form of return value, we could of course write it using expressions instead.

```
In [ ]: def triangle(n):
            return 0 if not n else triangle(n-1) + n
```

1.8 Debugging and common mistakes

[Note: it might be worth noting that we can cancel all (runaway) computations in Jupyter notebooks and restart the kernel. Note the stop-sign button and Run button above.]

The following is a stripped-down version of a a fairly common mistake. What is the problem with it (mathematically), and what might happen?

```
In [ ]: def bad_triangle(n):
            return bad_triangle(n-1) + n
```

Inserting a `print` call (or an `assertion`) might help you show what calls are performed. What is the problem?

```
In [ ]:
```

We might also encounter this one:

```
In [ ]: def triangle(n):
            if not n:
                return 0
            return triangle(n) + n
```

What is the problem? Read the code carefully!

```
In [ ]:
```

Another version might be the following (as we might rewrite the code above without an else statement):

```
In [ ]: def bad_triangle(n):
            if n > 0:
                return bad_triangle(n-1) + n
```

Since we have almost entirely consistently written our code with if-then-else, it's fairly obvious what is missing here. But it might be useful to try to predict in *what way* this code may fail. Will it be wrong in a way that is different from the one above? Try it out.

```
In [ ]:
```

1.9 A note on tail recursive functions

Tail recursive functions are those recursive functions where there are no delayed computations before the value is returned. In the code above, we had the line `return triangle(n-1) + n`. Before returning the value, we needed to perform that last addition. This ensures that we can't perform certain optimisations that allow the function to run in constant memory*.

* Python doesn't support tail call optimisation (TCO or TRE) anyway, according to the standard (due to a design decision by Guido van Rossum). A testament to Python's flexibility is that can get around this by means of decorators.

1.9.1 A tail recursive version of `triangle`

Can we rewrite the code above into a solution which achieves the same result, but in a tail recursive fashion? Yes. If we add upp the results along the way, instead of adding them up after the recursive call has finished, we won't have any saved computations. Introducing an accumulator or results parameter allows us to do this in a fairly simple way, at least when we don't need tree recursion.

```
In [ ]: # Takes a , and has a result parameter.

        def triangle_it(n, res=???):
            if not n:
                return res
            else:
                return triangle_it(????, ????)
```

Note that in the end, we simply return the accumulator/result value. This also gives us a hint about what the value ought to be to start with (if $n = 0$, then *res* should be whatever $triangle\_it(0)$ ought to be).

```
In [ ]: def triangle_it(n, res=0):
            if not n:
                return res
            else:
                return triangle_it(????, ????)
```

Thus we get to

```
In [ ]: def triangle_it(n, res=0):
            if not n:
                return res
            else:
                return triangle_it(n - 1, ???)
```

And finally

```
In [ ]: def triangle_it(n, res=0):
            if not n:
                return res
            else:
                return triangle_it(n - 1, res + n)
```

It might be helpful to print the values of *n* and *res* along a simple call, to understand how this works.

```
In [ ]:
```

Note the close relationship between this and the iterative reference function we had earlier in this document. What we have essentially done is to make all the state updates - before the first iteration of the loop `k` has this value and `result` has this value, afterwards they might have some other values - explicit and passed them along as arguments.

We could of course write a version of the code above which "counts up" instead of down. Defining a recursive helper function within the function might be of interest then. You might want to do that as an exercise (no handin required!).

```
In [ ]: def triangle_it2(n):
            def iter(k, res=0):
                pass # your code here

            return iter(k=0)
```