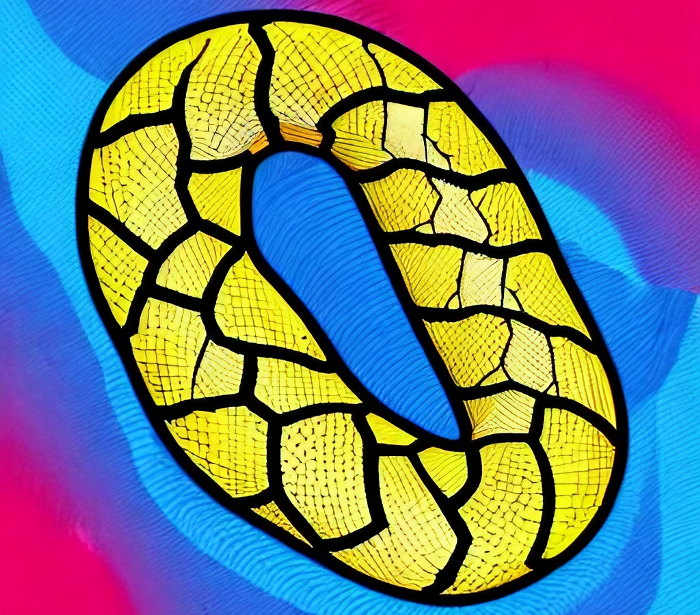
**Lazy Evaluation Using Recursive Python Generators**

**Can recursive functions use *“lazy evaluation”*? — Yes, they can — Using Python’s generator functions!**



Generated with Stable Diffusion

We all are familiar with Python’s generators and all their benefits. But, what if I told you that we can make them even better by combining them with recursion? So, let’s see how we can use them to implement *“lazy*…

**Why Even Bother?**

Before we get into the code, let’s first ask ourselves *“Why even bother? Do we really need recursive generators?”*. And the answer is… it depends. Naturally, a recursive generator will share both pros and cons of both generators as well as normal recursive functions.

For the generators, the number one reason why one would use them is *“lazy”* evaluation — that is — computing elements one at the time rather than all at once. As for the recursion, it simply makes sense for certain algorithms or problems which it can solve elegantly and succinctly, such as tree traversal.

Therefore, a situations where recursive generators would make sense are naturally recursive algorithms that might process large amount of data or elements, and therefore consume a lot of memory if run *“eagerly”*.

**The Basic Example**

Now that we know *why* we would use a recursive generator, let’s take a look at a *“simple”* example to understand *how* we can write one:

This short function — as the name suggests — yields successive numbers in binary. When called, it first simply yields "1" and after which comes the recursive call. The recursive call also yields "1", but that's given to the previous, non-recursive call as a prefix. With the prefix computed, the non-recursive call yields 2 values "10" and "11". After that, the recursive call continues execution by making another recursive call, going a level deeper and so the loop continues - prefixes bubble upwards so the outer frame is always yielding some result ending first with "0" and then "1".

Now, if we were to run it, we would get:

When it comes to recursion, explaining the code isn’t always sufficient for really understanding what’s happening. So, if you’re not sure how the binary\_counter actually works, then let's try working out individual steps:

The above, modified version adds a depth parameter and a couple of prints to help demonstrate what the code does. If we now call this code, we will get the following:

I hope this makes it a bit clearer, if not, consider manually working out the steps, or maybe using debugger in your IDE of choice, so that you can see the stack frames and variables in realtime.

You might be also asking, *“What’s the point of computing binary numbers this way?”* — and the answer is, well, there’s no good reason. There are definitely better and more readable ways to do that, but I think it demonstrates the concept fairly well. With that said, let’s now look at more useful examples of how we can use recursive generators…

**Putting It To Good Use**

When it comes to recursion, the obvious candidates for examples are various mathematical functions or — as shown here — combinatorics, more specifically [power-set](https://en.wikipedia.org/wiki/Power_set):

The function here uses similar flow as the binary counter earlier. To better understand it, we can translate the recursive part as:

* For every result in a smaller power-set (sequence[1:]) ...
* … Return the not used value ([sequence[0]]) + the result (item)
* … Then return result alone (item)

While mathematical functions can be nicely implemented using recursion, they aren’t really something we use on daily basis, so let’s now take a look at something different:

The above accumulate function computes a running total (sum) of elements of its list parameter. While the above code works, I don't recommend using it in practice, because you can and should use the following instead:

While on the topic of itertools, let’s also see how we can re-implement other common function:

The flatten function can be used to un-nest a nested list (or other iterable). I show this one here because it uses a bit of a different flow than the earlier ones - it leverages try/ except to separate the base/non-recursive part and the recursive code.

It can however, be rewritten without try/ expect if desired:

When talking about recursion, we obviously have to show an examples of recursive data structures, in this case a binary tree:

The above code implements a binary tree, including the recursive generator in a form of the \_\_iter\_\_ method. The same is also implemented in the inorder function, which makes the recursive calls a little clearer.

To demonstrate the usage of the above code, let’s create a simple tree:

Similar to the traversal of (binary) trees, we can use recursive generators also for examples when traversing JSON:

Traversing JSON this way might be practical if you’re working with very large data that would consume a lot of memory if loaded all at once.

By now, you might be getting the hang of how these weird generators work, but let’s anyway look at what happens when we call the above code:

And finally, another tree-like data structure which is commonly traversed recursively is a file-tree:

Here we implement get\_paths function that recursively yields all files in specified path. With that said, you're better off using the builtin path.rglob("\*") for this task as it also returns generator.

Also, while not useful in this instance, it’s good to note that send() function can be also used with recursive generators. So, an alternative implementation of above function:

This style of generator can be useful in case you need to control the recursion or if you need to communicate with the coroutine.

**Conclusion**

The examples in this article — in my opinion — show an elegant solutions to many problems that can be expressed recursively. However, elegant doesn’t always mean better. Oftentimes, using less *“elegant”* or succinct solution will produce much more readable and generally better code.

So, let’s not try to *“shoehorn”* recursive generators into code wherever possible and only use it where appropriate — that is — when implementing a recursive function that would benefit from lazy evaluation.