**Sorting out Python generator functions and yield from in my head**

March 15, 2017

Through a chain of reading, I wound up at [How the heck does async/await work in Python 3.5?](https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/) (via [the Trio tutorial](https://trio.readthedocs.io/en/latest/tutorial.html)). As has happened before when I started reading about Python 3's new async and await stuff, my head started hurting when I hit the somewhat breezy discussion of yield from and I felt the need to slow down and try to solidly understand this, which I haven't really before.

Generator functions are functions that contain a yield statement:

def fred(a):

r = yield 10

print("fred got:", r)

yield a

A straightforward generator function is there to produce a whole series of values without having to ever materialize all of them at once in a list or the like.

Calling a generator function does not return its result. Instead, it returns a generator object, which is a form of iterator:

>>> fred(10)

<generator object fred at 0x7f52a75ea1a8>

This generator object is in part a closure that captures the argument fred() was called with (and in general will preserve fred()'s state while it is repeatedly iterated). Note that [fred()'s code doesn't start executing until you try to get the first value from the iterator](https://utcc.utoronto.ca/~cks/space/blog/python/WhenGeneratorsRun).

One common pattern with a stack of generator functions (including [needing to modify or filter part of a generator's results](https://utcc.utoronto.ca/~cks/space/blog/python/GeneratorListModification)) was that you would have one generator function that wanted to call another one for a while. In the beginning this was done with explicit for loops and the like, but then Python added yield from. yield from takes a generator or iterator and exhausts it for you, repeatedly yield'ing the result.

def barney(a):

yield from fred(a)

(You can intermix yield and yield from and use both of them more than once in a function.)

Because generator functions actually return generators, not any sort of result, 'yield from func()' is essentially syntactic sugar for calling the function, getting a generator object back, and then calling yield from on the generator object. There is no special magic involved in that:

def barney(a):

gen = fred(a)

yield from gen

Because generator objects are ordinary objects, they can be returned through functions that are not generators themselves, provided that intermediate functions don't really attempt to manipulate them and simply return them as-is:

def jim(a):

return fred(a)

def bob(a):

yield from jim(a)

(If jim() actually iterated through fred()'s results, things would quietly go sideways in ways that might or might not be visible.)

When yield started out, it was a statement; however, that got revised so that it was an expression and could thus have a value, as we see in fred() where the value of one yield is assigned to r and then used later. You (the holder of the generator object) inject that value by calling .send() on the generator:

>>> g = fred(2)

>>> \_ = g.send(None); \_ = g.send("a")

fred got: a

(The first .send() starts the generator running and must be made with a None argument.)

As part of adding yield from, Python arranged it so that if you had a stack of yield from invocations and you called .send() on the outer generator object, the value you sent did not go to the outer generator object; instead it goes all the way down to the eventual generator object that is doing a yield instead of a yield from.

def level3(a):

# three levels of yield from

# and we pass through a normal

# functions too

yield from bob(a)

>>> g = level3(10)

>>> \_ = g.send(None); \_ = g.send("down there")

fred got: down there

This means that if you have a stack of functions that all relay things back up using 'yield from', you have a direct path from your top level code (here that's our interactive code where we called level3()) all the way down to the core generator function at the bottom of the call stack (here, the fred() function). You and it can communicate with each other through the values it yields and the values you send() to it without any function in the middle having to understand anything about this; it's entirely transparent to them.

(Don't accidentally write 'yield' instead of 'yield from', though. The good news about that mistake is that you'll catch it fast.)

Hopefully writing this has anchored yield from's full behavior and the logic behind it sufficiently solidly in my head that it will actually stick this time around.

**Sidebar: yield from versus yield of a generator**

Suppose that we have a little mistake:

def barney2(a):

yield fred(a)

What happens? Basically what you'd expect:

>>> list(barney(20))

fred got: None

[10, 20]

>>> list(barney2(20))

[<generator object fred at 0x7f0fdf4b9258>]

When we used yield instead of yield from, we returned a value instead of iterating through the generator. The value here is what we get as the result of calling fred(), which is a generator object.

By the way, a corollary to [strings being iterable](https://utcc.utoronto.ca/~cks/space/blog/python/StringsRecursivelyIterable) is that accidentally calling 'yield from' instead of 'yield' on a string won't fail the way that eg 'yield from 10' does but will instead give you a sequence of single characters. You'll probably notice that error fairly fast, though.

This behavior of yield from is pretty much a feature, because it means that you can yield from another function without having to care about whether it's an actual generator function or it merely returns an iterable object of some sort; either will work.