From: Reuven M. Lerner reuven@lerner.co.il
Subject: [Better developers] Coroutines
Date: Jun 24, 2019 at 8:00:02 AM
To: klockhart2@cogeco.ca

Last time, we took a look at generator functions -- which, when you call them, don't return a result right away. Rather, they return generators (i.e., iterable objects).

Put such a generator in a "for" loop, and the function will execute as written. Until it gets to a "yield" statement, that is. When that happens, the generator goes to sleep, keeping its state (and stack frame).

When the function restarts (thanks to the next iteration happening), it picks up from where it left off, as if nothing had happened, and continues to execute.

In other words, "yield" is sort of like "return", except that it only works within an iterator and pauses the function's execution at that point.

Everyone's favorite generator example is one that produces the Fibonacci sequence:

```
def fib():
    first = 0
    second = 1
    while True:
        yield first
        first, second = second, first+second

        if first > 10000:
            break
```

And sure enough, if we run this in a "for" loop and limit the number of times it can run, we'll get a bunch of Fibonacci number:

```
    for i in fib():
        print(i, end=' ')
```

I get:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
4181 6765
```

This is great, but let's make a modification: Instead of stopping when we get to 10,000, let's stop when we print 20 numbers:

```
def fib():
    first = 0
    second = 1
    for i in range(20):
        yield first
        first, second = second, first+second
```

And now let's make a further modification: I'd like to be able to change the value of "first" while things are running. Maybe I want to give a bigger number (or a smaller number), and calculate the Fibonacci sequence from there. Why should it always be the first 20, after all?

But here, we hit a roadblock: There isn't really any way for us to send a message into our generator, at least the way that it's written. The "yield" statement always returns a value to us, but it doesn't want to get a value from us. The communication is unidirectional.

Python provides a way around this, for communication in two directions. Normally, when we want to get the next value from an iterator, we (or Python) will use the "next" builtin function, which involves the "__next__" method on the iterator. But we can instead use the "send" method, which lets us send a message into the generator. For example:

```
In [7]: deffoo():
   ...:     x = 100
   ...:     while True:
   ...:         print(f"At top of loop, x = {x}")
   ...:         x = yield x
```

```
...:         print(f"Just got {x} from yield")
...:         x = x * 5
...:
```

What does the above generator do? It first sets "x" to be 100. And in the first iteration (i.e., when we first run "next" and ask for a value), then it runs up to and including the first "yield", returning "x". Sure enough:

```
In [8]: g = foo()

In [9]: next(g)  # start the generator
At top of loop, x = 100
Out[9]: 100
```

But now the generator is paused, and is waiting for another value. We can provide it with such a value by sending something to it:

```
In [10]: g.send(7)
Just got 7 from yield
At top of loop, x = 35
Out[10]: 35
```

Notice that as before, the generator is stopping whenever it hits the "yield". The difference is that whatever we send to the generator becomes the right-hand-side of that assignment, and thus lets us influence the output:

```
In [11]: g.send(9)
Just got 9 from yield
At top of loop, x = 45
Out[11]: 45
```

If we use "next" (which is the equivalent of "send(None)") or run "send" without any arguments, things fall apart:

```
    In [12]: g.send()

---------------------------------------------------------------------
------------
    TypeError                                 Traceback (most
recent call last)
    <ipython-input-13-1bfac9d701cf> in <module>
    ----> 1 g.send()

    TypeError: send() takes exactly one argument (0 given)
```

Used in this way, a generator is a "coroutine" -- a function with state that can go to sleep and re-awaken. If functionality takes a while to start up, this can allow us to avoid the startup costs.

Consider if I had a bunch of coroutines, each of which was able to retrieve data from the stock market, via the Internet. Now, the Internet might be fast where you live, but even the fastest connection is no match for a locally running CPU and memory. Which means that it might make sense, after we have made a request, for the function to go to sleep, waking up later on to check if there's an answer waiting.

You could further imagine a loop of such coroutines, with a Python program going through each of them in turn, re-awakening each coroutine and checking if it got anything from the network. If so, then great. And if not, then we can put this coroutine back to sleep, and try someone else.

Now, you might be wondering what all of this has to do with asyncio. After all, if you've read anything about asyncio, then you know that we no longer explicitly use generators and coroutines. And yet, the functionality is still tied to this sort of idea. And while the modern syntax hides some of this from us, the fact is that a loop is still there, going to sleep is still there, and coroutines are still there.

How does this all look in modern Python? We'll see next week.

Until then,

Reuven

PS: A **new cohort of Weekly Python Exercise** B2 (i.e., intermediate/advanced, part 2) starts on July 2nd. It's currently $100, but will go up to $120 on June 28th. **Don't miss out** on this chance to improve your Python fluency! (And no, you didn't need to be a part of B1 to participate.) More info is at WeeklyPythonExercise.com.

Or just reply to this message if you have any questions, including about discounts for students, seniors/retirees/pensioners, or if you're living outside of the 30 richest countries in the world.

If you're tired of getting this newsletter, then you can just unsubscribe by clicking Unsubscribe. No hard feelings! And in case you're so intrigued by my newsletter that you want to meet me at home, I publish this at 14 Migdal Oz Street, Apt. #1, Modi'in 7170334 Israel .