Arguments to Python generator functions

Posted on Tue 14 March 2017 in Code

In Python, a *generator function* is one that contains a `yield` statement inside the function body. Although this language construct has many fascinating use cases (PDF), the most common one is creating concise and readable iterators.

A typical case

Consider, for example, this simple function:

```python
def multiples(of):
    """Yields all multiples of given integer."""
    x = of
    while True:
        yield x
        x += of
```

which creates an (infinite) iterator over all multiples of given integer. A sample of its output looks like this:

```python
>>> from itertools import islice
>>> list(islice(multiples(of=5), 10))
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

If you were to replicate in a language such as Java or Rust — neither of which supports an equivalent of `yield` — you'd end up writing an *iterator* class. Python also has them, of course:

```python
class Multiples(object):
    """Yields all multiples of given integer."""

    def __init__(self, of):
        self.of = of
        self.current = 0

    def __iter__(self):
        return self

    def next(self):
        self.current += self.of
        return self.current

    __next__ = next  # Python 3
```

but they are usually not the first choice[1].

It’s also pretty easy to see why: they require explicit bookkeeping of any auxiliary state between iterations. Perhaps it’s not too much to ask for a trivial walk over integers, but it can get quite tricky if we were to iterate over recursive data structures, like trees or graphs. In `yield`-based generators, this isn’t a problem, because the state is stored within local variables on the coroutine stack.

Lazy!

It’s important to remember, however, that generator functions behave differently than regular functions do, even if the surface appearance often says otherwise.

The difference I wanted to explore in this post becomes apparent when we add some argument checking to the initial example:

```
def multiples(of):
    """Yields all multiples of given integer."""
    if of < 0:
        raise ValueError("expected a natural number, got %r" % (of,))

    x = of
    while True:
        yield x
        x += of
```

With that `if` in place, passing a negative number shall result in an exception. Yet when we attempt to do just that, it will seem as if nothing is happening:

```
>>> m = multiples(-10)
>>>
```

And to a certain degree, this is pretty much correct. Simply *calling* a generator function does comparatively little, and doesn’t actually execute any of its code! Instead, we get back a *generator object*:

```
>>> m
<generator object multiples at 0x10f0ceb40>
```

which is essentially a built-in analogue to the `Multiples` iterator instance. Commonly, it is said that both generator functions and iterator classes are *lazy*: they only do work when we asked (i.e. iterated over).

Getting eager

Oftentimes, this is perfectly okay. The laziness of generators is in fact one of their great strengths, which is particularly evident in the immense usefulness of the`itertools` module.

On the other hand, however, delaying argument checks and similar operations until later may hamper debugging. The classic engineering principle of failing fast applies here very fittingly: any errors should be signaled immediately. In Python, this means raising exceptions as soon as problems are detected.

Fortunately, it is possible to reconcile the benefits of laziness with (more) defensive programming. We can make the generator functions only a *little* more eager, just enough to verify the correctness of their arguments.

The trick is simple. We shall extract an *inner* generator function and only call it after we have checked the arguments:

```
def multiples(of):
    """Yields all multiples of given integer."""
    if of < 0:
        raise ValueError("expected a natural number, got %r" % (of,))

    def multiples():
        x = of
        while True:
            yield x
            x += of

    return multiples()
```

From the caller's point of view, nothing has changed in the typical case:

```
>>> multiples(10)
<generator object multiples at 0x110579190>
```

but if we try to make an incorrect invocation now, the problem is detected *immediately*:

```
>>> multiples(-5)

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    multiples(of=-5)
  File "<pyshell#0>", line 4, in multiples
    raise ValueError("expected a natural number, got %r" % (of,))
ValueError: expected a natural number, got -5
```

Pretty neat, especially for something that requires only two lines of code!

The last (micro)optimization

Indeed, we didn't even have to pass the arguments to the inner (generator) function, because they are already captured by the closure.

Unfortunately, this also has a slight performance cost. A captured variable (also known as a *cell variable*) is stored on the function object itself, so Python has to emit a different bytecode instruction (`LOAD_DEREF`) that involves an extra pointer dereference. Normally, this is not a problem, but in a tight generator loop it can make a difference.

We can eliminate this extra work[2] by passing the parameters explicitly:

```
# (snip)

def multiples(of):
    x = of
    while True:
        yield x
        x += of

return multiples(of)
```

This turns them into local variables of the inner function, replacing the `LOAD_DEREF` instructions with (aptly named) `LOAD_FAST` ones.

---

1. Technically, the *Multiples* class is here is both an *iterator* (because it has the `next`/`__next__` methods) and *iterable* (because it has `__iter__` method that returns an iterator, which happens to be the same object). This is common feature of iterators that are not associated with any collection, like the ones defined in the built-in `itertools` module. ↩

2. Note that if you engage in this kind of microoptimizations, I'd assume you have already changed your global lookup into local ones :) ↩

Python generators functions arguments closures