Python

PETER NORTON

Peter works on automating cloud environments. He loves using Python to solve problems. He has contributed to books on Linux and Python, helped with the New York Linux Users Group, and helped to organize past DevOpsDays NYC events. In addition to Python, Peter is slowly improving his knowledge of Rust, Clojure, and maybe other fun things. Even though he is a native New Yorker, he is currently living in and working from home in the northeast of Brazil. pcnorton@rbox.co.

This column is being written in December, which ends another year, which brings end of year holiday plans, deadlines (like the one for this column), and a chance to challenge yourself to do something different before the year is fully out and done.

For my part, over the last few years I've had a great time participating in the annual Advent of Code (adventofcode.com), which is a great way to take a break from work where you have to solve problems on a deadline, and... well, solve problems on a different deadline. But for fun.

It's a great opportunity to learn more about your preferred language, to try out a new language, or revisit how things work in a language you haven't used in a while. You can also compare notes with others and see how different languages can give you the tools you need (or how hard it is to build them from scratch if that's more to your taste).

For anyone who hasn't participated in one of these online advent calendars, this one involves creating a puzzle around Santa, elves, and a story arc adventure that you are on that gets Santa closer to delivering presents for all the good girls and boys. Each day you get a story, a problem description, an example of the data and what the results will be of the problem being presented (yes, tests), and a data set that's created for you so that your answer shouldn't work for anyone else (though the solutions should, of course). The answers are usually an integer, summing up all the work you've done. And you're rate limited to one answer per minute, so you can't just brute force the answer.

The problems are very much programming puzzler/interview type questions designed to let you stretch your computer science legs—data structures, complexity, etc. without having any serious stakes—and if you complete the puzzle you move on; it's all just for a good time. The problems are introduced day-by-day, but if you haven't done the challenge already, you can always visit the site as you read this column and participate if you feel like it.

What I enjoy about this is that it is so well executed. Very few programming interviews that I've seen are as well thought out as the Advent of Code, which speaks volumes for the organizers. The organizers have some themes—a variety of problems that require some knowledge that may be common in some jobs and problem domains but which in others can be novel and outside of the comfort zone.

Big-O Traps

One of the things you notice quickly is that solving the problems naively will lead you to quadratic solutions that will take forever with the size of the input you're provided. So one of the fun parts is getting to think about each particular problem, to think about the big-O characteristics of your code, and realizing your input is large enough to cause your computer to spin and struggle uselessly until the heat death of the universe.

These problems often run over familiar themes—some will involve iterating over lists, finding your way around other data structures forward and backward, over and over. As you may imagine, if you start with a little bit of bookkeeping, that sometimes turns into a lot of bookkeeping, which is a lot of hassle. When that starts to happen, it's helpful to step back. When

you can, sometimes stepping back includes treating the data like streams. In Python this basically means iterators and generators are your friends who take away the tedium. What's interesting and disappointing about this great and fun approach is how as you get more sophisticated with using iterators, you can sometimes get subtle and surprising behaviors, which aren't particularly well-documented (at least as far as I've seen).

Iterator Side Effects

With all that said, this year's Advent of Code had me encounter one of these side effects, one that I found quite surprising. It is simple, but I do think that in real-world usage it would cause hard-to-find bugs.

The specific behavior is in the `zip()` built-in function. If you've never used it before, it's sometimes easier to think of as syntactic sugar sprinkled over having to assign multiple variables in a loop. It can turn the following somewhat tedious code:

```
def odious(l1, l2, l3, l4, l5):
    """each argument is a list"""
    min_len = min(map(len, (l1, l2, l3, l4, l5,)))
    for iteration in range(min_len):
        v1 = l1[iteration]
        v2 = l2[iteration]
        v3 = l3[iteration]
        v4 = l4[iteration]
        v5 = l5[iteration]
        print(f"{v1}, {v2}, {v3}, {v4}, {v5}")
```

into something much simpler. This prints each element of the lists in the arguments as a group—first, all of the first elements, then all of the second elements, etc. The short, `zip()`-ified way of doing this looks like:

```
def melodious(l1, l2, l3, l4, l5):
    for v1, v2, v3, v4, v5 in zip(l1, l2, l3, l4, l5):
        print(f"{v1}, {v2}, {v3}, {v4}, {v5}")
```

Which is still clear and easy to understand. Since zip can work with any number of iterables, it's pretty flexible. It's been in Python since 2.0, and there's a lot more to read about it in PEP 201 at https://www.python.org/dev/peps/pep-0201/.

I also found the behavior of iterators interesting. Iterators are thoroughly ingrained in Python and feel very natural to use. However, they have a very specific definition, and if you want to know exactly what that is, I encourage you to read PEP 234: https://www.python.org/dev/peps/pep-0234/.

As I mentioned above, iterators allow us as Python programmers to have a potentially lazy stream of items, with only a few tradeoffs. On the upside, you can have infinite input that you can iterate over easily with `for` or `next()`; you can compose them with comprehensions and with really cool functions available in the `itertools` module! And iterators have led to generators with `yield` and generator comprehensions. A lot has been written in these pages about iterators, generators, co-routines, etc., so I will refer anyone interested to the excellent material in past *;login:* issues, which have gone into a lot of depth and breadth on the matter.

The downside of the tradeoff for how excellent iterators are is that we lose some of the flexibility of having a list or a special type or class whose position and indexability puts it entirely under our control. For an iterator to be useful, we must know that we're going to use it from beginning to end in a linear fashion—no rewinding, arbitrary glances at indexes, etc. In so many cases this is not a limitation but is specifically and exactly what we want, which is why iterators are so fantastic.

So, with that said, let me talk about the interesting problem that I ran into. The code involved looks something like this (in Python 3.7):

```
import itertools
def walk_forward(char_iter):
    """Consume input_iter, which is an iterator that provides
    one character at a time. When two characters match the
    filter criteria, remove them both and break so that the
    data can be walked backward to see if the new state has
    affected the keep_list.

    returns a list of characters that we want to keep
    """
    first_char = next(char_iter)
    keep_list = list()
    for second_char in char_iter:
        result = keep_or_remove(first_char, second_char)
        if not result:
            # Don't put the result into the keep list
            return keep_list
        keep_list.append(first_char)
        first_char = second_char
    return keep_list

def walk_backward(keep_list, char_iter):
    """A match has been found, and now we want to know if the
    combination of the last letter in the keep list, and the
    first letter in the char_iter could start eliminating each
    other. Essentially this works from the middle out as long
    as the characters would be eliminated. Once we find a pair
    that are keepers, we can exit from here and resume walking
    forward.

    Returns a list of characters - those that we still want to
    keep.
    """
    #Walk the keep_list backward
    first_gen = (x for x in keep_list[-1::-1])
```

```
for first, second in zip(first_gen, char_iter):
    result = keep_or_remove(first, second)
    # Return the results in the same order we got them
    if result:
        return list(itertools.chain([second, first], \
         first_gen])[-1::-1]
```

This works fine—with a `main()` function that walks forward until there is some elimination, then walks backward, then forward, and so on. This should basically work to eliminate pairs of letters that match the `keep_or_remove()` function, which I haven't included here.

The hidden problem in `walk_backward` is that the use of `zip` will always try to consume the first element from each iterator. So when the `keep_list` is shorter than the remaining contents of `char_iter` (as it is likely to be towards the beginning), everything is fine. However, if it's the second iterator that becomes exhausted, as may happen, then `zip` will have already consumed from the `first_gen`, and you can't put it back. So, in this case, you may have lost data. It's only one datapoint, which is exactly enough to make people very upset in the right circumstances, that is, outside the world of fun puzzles.

Now that we've looked at this with some more context, let's look at a simpler reproducer case:

```
>>> a = (x for x in 'abcde')
>>> for first, second in zip(a, ()):
...    print(f"{first}, {second}")
...
>>> rest = list(a)
>>> print(f"{rest}")
['b', 'c', 'd', 'e']
```

Working Around the Problem

Once I understood the issue, it bothered me because working around it made the program harder to read since the obvious workaround is tedious. Tedious solutions beg for better ones, especially when they're for fun. However, in this case it also led me to wonder why there isn't already a better solution, and maybe a bit about whether my idea of a better solution was in fact better at all.

If this were a problem that a lot of people cared about, a PEP on it would probably have appeared. I expect that since this is a small wart in one tiny part of the language, most people with work to do would solve this by avoiding `zip`, or by not using iterators, relying instead on lists or similar types with known, queryable lengths and ensuring that these lengths were uniform for each argument to `zip`, which is the sweet spot for a safe and reliable `zip`. This thought makes me sad because it would be nice if Python offered a better way to handle this.

So let's think about it a bit more and see what comes out of it.

One simple approach to fixing this problem would be to make a more robust iterator, and doing that is pretty easy. However, to be useful it would require the iterator protocol to be more robust. For example, you could envision a new class that allows some interrogation, like peeking or, maybe a bit less ambitious, the ability to ask whether it's primed (by which I mean it still may have more values in the future) or stopped (`StopIteration` has been raised) without losing a value.

Unfortunately, these aren't small self-contained decisions. A fundamental thing like altering the behavior of the iterator protocol would probably, in the worst case, mean that every battery-included function or expression that consumes an iterator and handles `StopIteration` would have to know that there is this new capability, which is now a lot of work with a lot of sharp edges ready to poke you.

So let's just start with the easy part for now, and we can explore the harder parts later.

Taking advantage of the iterator protocol, let's start with a naive first try—we'll write an iterator that lets us ask whether there's more data while otherwise behaving like a regular iterator.

```
class SnitchIterator(object):
    def __next__(self):
        while True:
            return next(self.iterator)

    def __iter__(self):
        return self

    def __init__(self, src):
        """Using a source iterator, list, etc. create a new
        iterator that lets you non-destructively ask if there
        is a next element or not"""
        self.iterator = iter(src)
    def more(self):
        try:
            res = next(self)
            if res:
                self.iterator = itertools.chain([res], \
                 self.iterator)
                return True
        except StopIteration:
            return False
```

Now we can ask "Is there more to this?" and get an answer. But to solve the earlier problem, we'll also need a slightly different `zip` function to take advantage of this new feature, or else we're at a dead end. The special-case `zip`, or `snitch_zip`, would look like this:

```
def snitch_zip(*args):
    """Iterables must be a container, not an iterator. We must
    be able to go through them more than one time"""
    if False in ['__iter__' in dir(it) for it in args]:
        raise TypeError('All variables in *args must have \
         __iter__')
    while True:
        for series in args:
            if not series.more():
                raise StopIteration
        yield [next(series) for series in args]
```

You can see that creating a modified `zip` is pretty easy. However, this becomes a special case, which detracts from the simplicity of the iterator model, is going to perform worse than the built-in `zip`, and will probably have issues that we will cut ourselves on. There's nothing wrong with doing this for yourself when the use is appropriate, but it feels like something that, to be useful, would be better if it were in the language or at least in the standard library.

Doing something like this in the core language might have some niche usefulness but would come with the potential to break a lot of existing code, or at least make that code confusing. Some languages have macros and other practices to enable extending existing functionality for experimentation, and Python has at least one project that does this as well. If I can, I'll see if I can get `zip` to work with the `SnitchIterator` and discuss that next time.

Governance Follow-Up

Also, as a follow-up to the last column, the vote for the new governance model for Python has been counted, and PEP 8016, the steering council model, has been accepted: https://www.python.org/dev/peps/pep-8016/.

This means that the BDFL model will be replaced by a five-person elected steering committee with the goal of taking care of the language, and they will be subject to oversight by the core team members—those who actively contribute to the community.

You can see the results of the actual vote at https://discuss.python.org/t/python-governance-vote-december-2018-results/546.

Again, I encourage anyone interested to follow this process closely.

Happy New Year!