10-Iterators

September 28, 2016

1 Iterators

Often an important piece of data analysis is repeating a similar calculation, over and over, in an automated fashion. For example, you may have a table of a names that you'd like to split into first and last, or perhaps of dates that you'd like to convert to some standard format. One of Python's answers to this is the *iterator* syntax. We've seen this already with the `range` iterator:

```
In [1]: for i in range(10):
            print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

Here we're going to dig a bit deeper. It turns out that in Python 3, `range` is not a list, but is something called an *iterator*, and learning how it works is key to understanding a wide class of very useful Python functionality.

1.1 Iterating over lists

Iterators are perhaps most easily understood in the concrete case of iterating through a list. Consider the following:

```
In [2]: for value in [2, 4, 6, 8, 10]:
            # do some operation
            print(value + 1, end=' ')
```

```
3 5 7 9 11
```

The familiar "`for x in y`" syntax allows us to repeat some operation for each value in the list. The fact that the syntax of the code is so close to its English description ("*for [each] value in [the] list*") is just one of the syntactic choices that makes Python such an intuitive language to learn and use.

But the face-value behavior is not what's *really* happening. When you write something like "`for val in L`", the Python interpreter checks whether it has an *iterator* interface, which you can check yourself with the built-in `iter` function:

```
In [3]: iter([2, 4, 6, 8, 10])
```

```
Out[3]: <list_iterator at 0x104722400>
```

It is this iterator object that provides the functionality required by the `for` loop. The `iter` object is a container that gives you access to the next object for as long as it's valid, which can be seen with the built-in function `next`:

```
In [4]: I = iter([2, 4, 6, 8, 10])

In [5]: print(next(I))

2


In [6]: print(next(I))

4


In [7]: print(next(I))

6
```

What is the purpose of this level of indirection? Well, it turns out this is incredibly useful, because it allows Python to treat things as lists that are *not actually lists*.

1.2 `range()`: A List Is Not Always a List

Perhaps the most common example of this indirect iteration is the `range()` function in Python 3 (named `xrange()` in Python 2), which returns not a list, but a special `range()` object:

```
In [8]: range(10)

Out[8]: range(0, 10)
```

`range`, like a list, exposes an iterator:

```
In [9]: iter(range(10))

Out[9]: <range_iterator at 0x1045a1810>
```

So Python knows to treat it *as if* it's a list:

```
In [10]: for i in range(10):
             print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

The benefit of the iterator indirection is that *the full list is never explicitly created!* We can see this by doing a range calculation that would overwhelm our system memory if we actually instantiated it (note that in Python 2, `range` creates a list, so running the following will not lead to good things!):

```
In [11]: N = 10 ** 12
         for i in range(N):
             if i >= 10: break
             print(i, end=', ')
```

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

If `range` were to actually create that list of one trillion values, it would occupy tens of terabytes of machine memory: a waste, given the fact that we're ignoring all but the first 10 values!

In fact, there's no reason that iterators ever have to end at all! Python's `itertools` library contains a `count` function that acts as an infinite range:

```
In [12]: from itertools import count

         for i in count():
             if i >= 10:
                 break
             print(i, end=', ')
```

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Had we not thrown-in a loop break here, it would go on happily counting until the process is manually interrupted or killed (using, for example, `ctrl-C`).

1.3 Useful Iterators

This iterator syntax is used nearly universally in Python built-in types as well as the more data science-specific objects we'll explore in later sections. Here we'll cover some of the more useful iterators in the Python language:

1.3.1 `enumerate`

Often you need to iterate not only the values in an array, but also keep track of the index. You might be tempted to do things this way:

```
In [13]: L = [2, 4, 6, 8, 10]
         for i in range(len(L)):
             print(i, L[i])
```

```
0 2
1 4
2 6
3 8
4 10
```

Although this does work, Python provides a cleaner syntax using the `enumerate` iterator:

```
In [14]: for i, val in enumerate(L):
             print(i, val)
```

```
0 2
1 4
2 6
3 8
4 10
```

This is the more "Pythonic" way to enumerate the indices and values in a list.

1.3.2 zip

Other times, you may have multiple lists that you want to iterate over simultaneously. You could certainly iterate over the index as in the non-Pythonic example we looked at previously, but it is better to use the `zip` iterator, which zips together iterables:

```
In [15]: L = [2, 4, 6, 8, 10]
         R = [3, 6, 9, 12, 15]
         for lval, rval in zip(L, R):
             print(lval, rval)
```

```
2 3
4 6
6 9
8 12
10 15
```

Any number of iterables can be zipped together, and if they are different lengths, the shortest will determine the length of the `zip`.

1.3.3 map and filter

The `map` iterator takes a function and applies it to the values in an iterator:

```
In [16]: # find the first 10 square numbers
         square = lambda x: x ** 2
         for val in map(square, range(10)):
             print(val, end=' ')
```

```
0 1 4 9 16 25 36 49 64 81
```

The `filter` iterator looks similar, except it only passes-through values for which the filter function evaluates to True:

```
In [17]: # find values up to 10 for which x % 2 is zero
         is_even = lambda x: x % 2 == 0
         for val in filter(is_even, range(10)):
             print(val, end=' ')
```

```
0 2 4 6 8
```

The map and filter functions, along with the reduce function (which lives in Python's functools module) are fundamental components of the *functional programming* style, which, while not a dominant programming style in the Python world, has its outspoken proponents (see, for example, the pytoolz library).

1.3.4 Iterators as function arguments

We saw in *args and **kwargs: Flexible Arguments. that *args and **kwargs can be used to pass sequences and dictionaries to functions. It turns out that the *args syntax works not just with sequences, but with any iterator:

```
In [18]: print(*range(10))

0 1 2 3 4 5 6 7 8 9
```

So, for example, we can get tricky and compress the map example from before into the following:

```
In [19]: print(*map(lambda x: x ** 2, range(10)))

0 1 4 9 16 25 36 49 64 81
```

Using this trick lets us answer the age-old question that comes up in Python learners' forums: why is there no unzip() function which does the opposite of zip()? If you lock yourself in a dark closet and think about it for a while, you might realize that the opposite of zip() is... zip()! The key is that zip() can zip-together any number of iterators or sequences. Observe:

```
In [20]: L1 = (1, 2, 3, 4)
         L2 = ('a', 'b', 'c', 'd')

In [21]: z = zip(L1, L2)
         print(*z)

(1, 'a') (2, 'b') (3, 'c') (4, 'd')


In [22]: z = zip(L1, L2)
         new_L1, new_L2 = zip(*z)
         print(new_L1, new_L2)

(1, 2, 3, 4) ('a', 'b', 'c', 'd')
```

Ponder this for a while. If you understand why it works, you'll have come a long way in understanding Python iterators!

1.4 Specialized Iterators: `itertools`

We briefly looked at the infinite `range` iterator, `itertools.count`. The `itertools` module contains a whole host of useful iterators; it's well worth your while to explore the module to see what's available. As an example, consider the `itertools.permutations` function, which iterates over all permutations of a sequence:

```
In [23]: from itertools import permutations
         p = permutations(range(3))
         print(*p)
```

```
(0, 1, 2) (0, 2, 1) (1, 0, 2) (1, 2, 0) (2, 0, 1) (2, 1, 0)
```

Similarly, the `itertools.combinations` function iterates over all unique combinations of `N` values within a list:

```
In [24]: from itertools import combinations
         c = combinations(range(4), 2)
         print(*c)
```

```
(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)
```

Somewhat related is the `product` iterator, which iterates over all sets of pairs between two or more iterables:

```
In [25]: from itertools import product
         p = product('ab', range(3))
         print(*p)
```

```
('a', 0) ('a', 1) ('a', 2) ('b', 0) ('b', 1) ('b', 2)
```

Many more useful iterators exist in `itertools`: the full list can be found, along with some examples, in Python's online documentation.