12-Generators

September 28, 2016

1 Generators

Here we'll take a deeper dive into Python generators, including *generator expressions* and *generator functions*.

1.1 Generator Expressions

The difference between list comprehensions and generator expressions is sometimes confusing; here we'll quickly outline the differences between them:

1.1.1 List comprehensions use square brackets, while generator expressions use parentheses

This is a representative list comprehension:

```
In [1]: [n ** 2 for n in range(12)]
```

```
Out[1]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

While this is a representative generator expression:

```
In [2]: (n ** 2 for n in range(12))
```

```
Out[2]: <generator object <genexpr> at 0x104a60518>
```

Notice that printing the generator expression does not print the contents; one way to print the contents of a generator expression is to pass it to the `list` constructor:

```
In [3]: G = (n ** 2 for n in range(12))
        list(G)
```

```
Out[3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

1.1.2 A list is a collection of values, while a generator is a recipe for producing values

When you create a list, you are actually building a collection of values, and there is some memory cost associated with that. When you create a generator, you are not building a collection of values, but a recipe for producing those values. Both expose the same iterator interface, as we can see here:

```
In [4]: L = [n ** 2 for n in range(12)]
        for val in L:
            print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121

In [5]: G = (n ** 2 for n in range(12))
        for val in G:
            print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121
```

The difference is that a generator expression does not actually compute the values until they are needed. This not only leads to memory efficiency, but to computational efficiency as well! This also means that while the size of a list is limited by available memory, the size of a generator expression is unlimited!

An example of an infinite generator expression can be created using the `count` iterator defined in `itertools`:

```
In [6]: from itertools import count
        count()

Out[6]: count(0)

In [7]: for i in count():
            print(i, end=' ')
            if i >= 10: break

0 1 2 3 4 5 6 7 8 9 10
```

The `count` iterator will go on happily counting forever until you tell it to stop; this makes it convenient to create generators that will also go on forever:

```
In [8]: factors = [2, 3, 5, 7]
        G = (i for i in count() if all(i % n > 0 for n in factors))
        for val in G:
            print(val, end=' ')
            if val > 40: break

1 11 13 17 19 23 29 31 37 41
```

You might see what we're getting at here: if we were to expand the list of factors appropriately, what we would have the beginnings of is a prime number generator, using the Sieve of Eratosthenes algorithm. We'll explore this more momentarily.

1.1.3 A list can be iterated multiple times; a generator expression is single-use

This is one of those potential gotchas of generator expressions. With a list, we can straightforwardly do this:

```
In [9]: L = [n ** 2 for n in range(12)]
        for val in L:
            print(val, end=' ')
        print()

        for val in L:
            print(val, end=' ')
```

```
0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

A generator expression, on the other hand, is used-up after one iteration:

```
In [10]: G = (n ** 2 for n in range(12))
         list(G)

Out[10]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]

In [11]: list(G)

Out[11]: []
```

This can be very useful because it means iteration can be stopped and started:

```
In [12]: G = (n**2 for n in range(12))
         for n in G:
             print(n, end=' ')
             if n > 30: break

         print("\ndoing something in between")

         for n in G:
             print(n, end=' ')
```

```
0 1 4 9 16 25 36
doing something in between
49 64 81 100 121
```

One place I've found this useful is when working with collections of data files on disk; it means that you can quite easily analyze them in batches, letting the generator keep track of which ones you have yet to see.

1.2 Generator Functions: Using `yield`

We saw in the previous section that list comprehensions are best used to create relatively simple lists, while using a normal `for` loop can be better in more complicated situations. The same is true of generator expressions: we can make more complicated generators using *generator functions*, which make use of the `yield` statement.

Here we have two ways of constructing the same list:

```
In [13]: L1 = [n ** 2 for n in range(12)]

         L2 = []
         for n in range(12):
             L2.append(n ** 2)

         print(L1)
         print(L2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Similarly, here we have two ways of constructing equivalent generators:

```
In [14]: G1 = (n ** 2 for n in range(12))

         def gen():
             for n in range(12):
                 yield n ** 2

         G2 = gen()
         print(*G1)
         print(*G2)
```

```
0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

A generator function is a function that, rather than using `return` to return a value once, uses `yield` to yield a (potentially infinite) sequence of values. Just as in generator expressions, the state of the generator is preserved between partial iterations, but if we want a fresh copy of the generator we can simply call the function again.

1.3 Example: Prime Number Generator

Here I'll show my favorite example of a generator function: a function to generate an unbounded series of prime numbers. A classic algorithm for this is the *Sieve of Eratosthenes*, which works something like this:

```
In [15]: # Generate a list of candidates
         L = [n for n in range(2, 40)]
         print(L)
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24
```

```
In [16]: # Remove all multiples of the first value
         L = [n for n in L if n == L[0] or n % L[0] > 0]
         print(L)
```

```
[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39]
```

```
In [17]: # Remove all multiples of the second value
         L = [n for n in L if n == L[1] or n % L[1] > 0]
         print(L)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37]
```

```
In [18]: # Remove all multiples of the third value
         L = [n for n in L if n == L[2] or n % L[2] > 0]
         print(L)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

If we repeat this procedure enough times on a large enough list, we can generate as many primes as we wish.

Let's encapsulate this logic in a generator function:

```
In [19]: def gen_primes(N):
             """Generate primes up to N"""
             primes = set()
             for n in range(2, N):
                 if all(n % p > 0 for p in primes):
                     primes.add(n)
                     yield n

         print(*gen_primes(100))
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

That's all there is to it! While this is certainly not the most computationally efficient implementation of the Sieve of Eratosthenes, it illustrates how convenient the generator function syntax can be for building more complicated sequences.