From loop to comprehension in Python

Philip Klein

Basics

Let's say you want a list consisting of the first 10 nonnegative integers. Here's the traditional way to write this.

```
>>> L = []
>>> i = 0
>>> while i < 10:
...  L.append(i)
...  i += 1
...
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python provides *ranges*, which make this easier. A slightly slicker way to write it is to use a *for*-loop together with a range:

```
>>> L = []
>>> for i in range(10):
...  L.append(i)
...
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Here the variable i is called the *loop control variable*.

There is an even easier way to get this list: convert the range to a list:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Now let's consider a different goal. We still want a list but instead of the list consisting of the integers from 0 through 9, we want the list that consists of squares of these integers. The traditional way to obtain this list is with a for-loop:

```
>>> L = []
>>> for i in range(10):
...  L.append(i*i)
...
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

However, Python provides a more concise way to construct this list, namely a *list comprehension*:

```
>>> [i*i for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Let's compare the two forms. In each form, there is a loop control variable (and in each form the variable is `i`). In each form, the variable iterates over the elements of `range(10)`. These similarities are reflected by the occurence of the code fragment `for i in range(10)` in both forms.

In both forms, the expression `i*i` appears. This expression specifies how a typical value of the final list is derived from the corresponding value of the control variable.

One difference that throws people off a bit is that in the comprehension that expression appears *before* the part starting `for i ...` whereas in the loop that expression comes after.

The advantage of the comprehension over the loop is that it more precisely and directly captures the goal. In contrast, the loop code includes two statements—the initial assignment of the empty list to `L` and the appending of each element—that are only indirectly related to the goal.

The two statements that are in the loop and not in the comprehension—the statement initializing the variable `L` and the statement appending an element to the list that `L` refers to—are there because a loop is a more general computation than a comprehension. A list comprehension is *only* useful for creating a list consisting of elements derived one at a time from a control variable. Since it more specifically targets this goal, it requires less specification to achieve it. In fact, using a comprehension tells Python more—and, as a result, Python can usually execute a list comprehension more quickly than it can execute the corresponding loop code.

Note also that the comprehension doesn't need a variable `L` to keep track of the growing list. The value of the comprehension is the list. (Of course, you could choose to assign the value to a variable—but you could also use it as the argument to a procedure or more generally include it in a larger expression.)

Variations in the expression for a typical element

The expression specifying a typical element need not depend on the loop variable. Let's create a list of pseudorandom numbers using Python's *random* module. First, the loop:

```
>>> from random import random
>>> L = []
>>> for i in range(10):
...  L.append(random())
...
>>> L
[0.5816737902086452, ... , 0.006916891440394313]
```

Now the comprehension:

```
>>> [random() for i in range(10)]
[0.9971398088326107, ... , 0.9061110236553993]
```

In each case, the expression specifying a typical element is `random()`, which doesn't depend on the value of the control variable. For this reason, in each form we can replace the control variable with an underscore, which tells Python we don't want to name the control variable because we don't need its value.

```
>>> [random() for _ in range(10)]
[0.5128972594823706, ... , 0.06741888378048344]
```

The expression specifying a typical element can depend on variables defined outside the comprehension. Here's an example in which we assign to a variable `c`, and then evaluate a comprehension in which the expression specifying a typical element depends on the value of `c`:

```
>>> c = 2
>>> [c*i for i in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Similarly, the expression specifying a typical element can include a procedure defined previously:

```
>>> def f(x): return 2*x
...
>>> [f(i) for i in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Conditionals

Now we introduce conditionals. Let's say we want the list to contain the squares of only *some* of the nonnegative integers less than 10—those that are not divisible by 3. Here's the loop form:

```
>>> L = []
>>> for i in range(10):
...  if i % 3 != 0:
...   L.append(i*i)
...
>>> L
[1, 4, 16, 25, 49, 64]
```

All we did was put the `append` statement inside an `if` (or *conditional*) statement. We can achieve the same result in a comprehension:

```
>>> [i*i for i in range(10) if i % 3 != 0]
[1, 4, 16, 25, 49, 64]
```

Nested loops and double comprehensions

What about nested loops? There are two kinds of lists one might want to obtain using a nested loop. First, let's create a list consisting of all pairs (i, j) where i is in `range(10)` and j is in $\{2, 5, 9\}$. Here's the loop form:

```
>>> L = []
>>> for i in range(10):
...   for j in {2, 5, 9}:
...     L.append((i,j))
...
```

```
>>> L
[(0, 9), (0, 2), (0, 5), (1, 9), (1, 2), (1, 5), (2, 9), (2, 2), (2,5),
 (3, 9), (3, 2), (3, 5), (4, 9), (4, 2), (4, 5), (5, 9), (5, 2), (5, 5),
 (6, 9), (6, 2), (6, 5), (7, 9), (7, 2), (7, 5), (8, 9), (8, 2), (8, 5),
 (9, 9), (9, 2), (9, 5)]
```

Here's the comprehension form:

```
>>> [(i,j) for i in range(10) for j in {2, 5, 9}]
[(0, 9), ... , (9, 5)]
```

In each case, we create one list whose elements have the form `(i,j)`. I refer to a comprehension of this form as a *double* comprehension.

Nested loops and nested comprehensions

In another use of nested loops, one might want to create a list of lists. Each iteration of the inner loop builds a list, and the outer loop uses these lists as elements of a bigger list.

```
>>> list_of_lists = []
>>> for i in range(3):
...     L = []
...     for j in range(4):
...         L.append(j)
...     list_of_lists.append(L)
...
>>> list_of_lists
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```

In the example, each iteration of the inner loop creates a list `[0,1,2,3]`, and the outer loop creates a three-element list each element of which is `[0,1,2,3]`. This is a silly example since the inner lists are all the same. Here's a version in which the inner lists are different:

```
>>> list_of_lists = []
>>> for i in range(3):
...     L = []
...     for j in range(4):
...         L.append(i*j)
...     list_of_lists.append(L)
...
>>> list_of_lists
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

We can simplify this code by replacing the inner loop with a comprehension:

```
>>> list_of_lists = []
>>> for i in range(3):
...     list_of_lists.append([i*j for j in range(4)])
```

```
...
>>> list_of_lists
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

In this example, each element appended to `list_of_lists` is the value of comprehension `[i*j for j in range(4)]`. The expression `i*j` specifying the typical element of the list depends not just on the control variable of the comprehension, `j`, but also on another variable, `i`, which is assigned to outside the comprehension. In this case, that other variable `i` happens to be the control variable of the outer loop, but that has no significance for the meaning of the comprehension.

To make the dependence on `i` more explicit, I will define a procedure that takes `x` as an argument, and use that procedure in the loop:

```
>>> def f(x): return [x*j for j in range(4)]
...
>>> list_of_lists = []
>>> for i in range(3):
...     list_of_lists.append(f(i))
...
>>> list_of_lists
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

When we look at this loop, we might recognize that it can be replaced by a comprehension:

```
>>> [f(i) for i in range(3)]
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

Finally, this comprehension is equivalent to one in which the invocation of the procedure `f` is replaced with the body of the procedure:

```
>>> [[i*j for j in range(4)] for i in range(3)]
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6]]
```

This *nested* comprehension mimics the nested loop we started with. One syntactic difference is that the specification of the outer comprehension occurs *after* the specification of the inner comprehension; in the nested loop, the specification of the outer loop occurs before the specification of the inner loop.

Even those who are used to nested loops sometimes have difficulty writing nested comprehensions. Try writing the nested loop first, then moving the inner loop into a procedure. Then you can separately (a) rewrite the procedure as a comprehension and (b) rewrite the outer loop as a comprehension.

To review, there are two ways we might use a nested loop to construct a list. One is equivalent to a double comprehension and the other is equivalent to a nested comprehension.