11-List-Comprehensions

February 11, 2017

1 List Comprehensions

If you read enough Python code, you'll eventually come across the terse and efficient construction known as a *list comprehension*. This is one feature of Python I expect you will fall in love with if you've not used it before; it looks something like this:

```
In [1]: [i for i in range(20) if i % 3 > 0]
```

```
Out[1]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

The result of this is a list of numbers which excludes multiples of 3. While this example may seem a bit confusing at first, as familiarity with Python grows, reading and writing list comprehensions will become second nature.

1.1 Basic List Comprehensions

List comprehensions are simply a way to compress a list-building for-loop into a single short, readable line. For example, here is a loop that constructs a list of the first 12 square integers:

```
In [2]: L = []
        for n in range(12):
            L.append(n ** 2)
        L
```

```
Out[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

The list comprehension equivalent of this is the following:

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

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

As with many Python statements, you can almost read-off the meaning of this statement in plain English: "construct a list consisting of the square of `n` for each `n` up to 12".

This basic syntax, then, is `[`*`expr`* `for` *`var`* `in` *`iterable`*`]`, where *`expr`* is any valid expression, *`var`* is a variable name, and *`iterable`* is any iterable Python object.

1.2 Multiple Iteration

Sometimes you want to build a list not just from one value, but from two. To do this, simply add another `for` expression in the comprehension:

```
In [4]: [(i, j) for i in range(2) for j in range(3)]

Out[4]: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Notice that the second `for` expression acts as the interior index, varying the fastest in the resulting list. This type of construction can be extended to three, four, or more iterators within the comprehension, though at some point code readibility will suffer!

1.3 Conditionals on the Iterator

You can further control the iteration by adding a conditional to the end of the expression. In the first example of the section, we iterated over all numbers from 1 to 20, but left-out multiples of 3. Look at this again, and notice the construction:

```
In [5]: [val for val in range(20) if val % 3 > 0]

Out[5]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

The expression `(i % 3 > 0)` evaluates to `True` unless `val` is divisible by 3. Again, the English language meaning can be immediately read off: "Construct a list of values for each value up to 20, but only if the value is not divisible by 3". Once you are comfortable with it, this is much easier to write – and to understand at a glance – than the equivalent loop syntax:

```
In [6]: L = []
        for val in range(20):
            if val % 3:
                L.append(val)
        L

Out[6]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

1.4 Conditionals on the Value

If you've programmed in C, you might be familiar with the single-line conditional enabled by the `?` operator:

```
int absval = (val < 0) ? -val : val
```

Python has something very similar to this, which is most often used within list comprehensions, `lambda` functions, and other places where a simple expression is desired:

```
In [7]: val = -10
        val if val >= 0 else -val

Out[7]: 10
```

We see that this simply duplicates the functionality of the built-in `abs()` function, but the construction lets you do some really interesting things within list comprehensions. This is getting pretty complicated now, but you could do something like this:

```
In [8]: [val if val % 2 else -val
         for val in range(20) if val % 3]

Out[8]: [1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

Note the line break within the list comprehension before the `for` expression: this is valid in Python, and is often a nice way to break-up long list comprehensions for greater readibility. Look this over: what we're doing is constructing a list, leaving out multiples of 3, and negating all mutliples of 2.

Once you understand the dynamics of list comprehensions, it's straightforward to move on to other types of comprehensions. The syntax is largely the same; the only difference is the type of bracket you use.

For example, with curly braces you can create a `set` with a *set comprehension*:

```
In [11]: {n**2 for n in range(12)}

Out[11]: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121}
```

Recall that a `set` is a collection that contains no duplicates. The set comprehension respects this rule, and eliminates any duplicate entries:

```
In [12]: {a % 3 for a in range(1000)}

Out[12]: {0, 1, 2}
```

With a slight tweak, you can add a colon (`:`) to create a *dict comprehension*:

```
In [13]: {n:n**2 for n in range(6)}

Out[13]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Finally, if you use parentheses rather than square brackets, you get what's called a *generator expression*:

```
In [15]: (n**2 for n in range(12))

Out[15]: <generator object <genexpr> at 0x1027a5a50>
```

A generator expression is essentially a list comprehension in which elements are generated as-needed rather than all at-once, and the simplicity here belies the power of this language feature: we'll explore this more next.