Effective SOFTWARE DEVELOPMENT SERIES
Scott Meyers, Consulting Editor

Effective PYTHON

125 Specific Ways to Write Better Python

THIRD EDITION

Brett Slatkin

Effective Python

Third Edition

With a list comprehension, I can achieve the same outcome in a single line by specifying the expression for my computation along with the input sequence variable to loop over:

```
squares = [x**2 for x in a]  # List comprehension
print(squares)
```

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

Unless you're applying a single-argument function, list comprehensions are clearer than the `map` built-in function for simple cases. `map` requires the creation of a `lambda` function for the computation (see Item 39: "Prefer `functools.partial` over `lambda` Expressions for Glue Functions"), which is visually noisy in comparison:

```
alt = map(lambda x: x**2, a)
```

Unlike `map`, list comprehensions let you easily filter items from the input list to remove corresponding outputs from the result. For example, say that I want to compute the squares of the numbers that are divisible by 2. Here, I do this by adding an `if` clause to the list comprehension after the loop:

```
even_squares = [x**2 for x in a if x % 2 == 0]
print(even_squares)
```

```
>>>
[4, 16, 36, 64, 100]
```

The `filter` built-in function can be used along with `map` to achieve the same outcome, but it is much harder to read due to nesting and boilerplate:

```
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

Dictionaries and sets have their own equivalents of list comprehensions (called *dictionary comprehensions* and *set comprehensions*, respectively). These make it easy to create other types of derivative data structures when writing algorithms:

```
even_squares_dict = {x: x**2 for x in a if x % 2 == 0}
threes_cubed_set = {x**3 for x in a if x % 3 == 0}
print(even_squares_dict)
print(threes_cubed_set)
```

```
>>>
{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
{216, 729, 27}
```

Achieving the same outcome is possible with `map` and `filter` if you wrap each call with a corresponding constructor. These statements get so long that you have to break them up across multiple lines, which is even noisier and should be avoided:

```
alt_dict = dict(
    map(
        lambda x: (x, x**2),
        filter(lambda x: x % 2 == 0, a),
    )
)
alt_set = set(
    map(
        lambda x: x**3,
        filter(lambda x: x % 3 == 0, a),
    )
)
```

However, one benefit of the `map` and `filter` built-in functions is that they return iterators that incrementally produce one result at a time. This enables these functions to be composed together efficiently with minimal memory usage (see Item 43: "Consider Generators Instead of Returning Lists" and Item 24: "Consider `itertools` for Working with Iterators and Generators" for background). List comprehensions, in contrast, materialize the entire result upon evaluation, which consumes much more memory. Luckily, Python also provides a syntax that's very similar to list comprehensions that can create infinitely long, memory-efficient streams of values (see Item 44: "Consider Generator Expressions for Large List Comprehensions").

Things to Remember

- List comprehensions are clearer than the `map` and `filter` built-in functions because they don't require `lambda` expressions.
- List comprehensions allow you to easily skip items from the input `list` by using `if` clauses, a behavior that `map` doesn't support without help from `filter`.
- Dictionaries and sets may also be created using comprehensions.
- List comprehensions materialize the full result when evaluated, which can use a significant amount of memory compared to an iterator that produces each output incrementally.

Item 41: Avoid More Than Two Control Subexpressions in Comprehensions

Beyond basic usage (see Item 40: "Use Comprehensions Instead of `map` and `filter`"), comprehensions also support multiple levels of looping. For example, say that I want to simplify a matrix (a list containing other list instances) into one flat list of all items. Here, I do this with a list comprehension by including two `for` subexpressions. These subexpressions run in the order provided, from left to right:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]
flat = [x for row in matrix for x in row]
print(flat)
```

```
>>>
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This example is simple, readable, and a reasonable usage of multiple loops in a comprehension. Another reasonable usage of multiple loops involves replicating the two-level-deep layout of the input list. For example, say that I want to square the value in each cell of a two-dimensional matrix. This comprehension is noisier because of the extra `[]` characters, but it's still relatively easy to read:

```
squared = [[x**2 for x in row] for row in matrix]
print(squared)
```

```
>>>
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

If this comprehension included another loop, it would get so long that I'd have to split it over multiple lines:

```
my_lists = [
    [[1, 2, 3], [4, 5, 6]],
    ...
]
flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
```

At this point, the multiline comprehension isn't much shorter than the alternative. Here, I produce the same result using normal loop

statements. The indentation of this version makes the looping clearer than the three-level-list comprehension above:

```
flat = []
for sublist1 in my_lists:
    for sublist2 in sublist1:
        flat.extend(sublist2)
```

Comprehensions support multiple `if` conditions. Multiple conditions at the same loop level have an implicit `and` expression. For example, say that I want to filter a list of numbers to only even values greater than 4. These two list comprehensions are equivalent:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

Conditions can be specified at each level of looping after the `for` subexpression. For example, say that I want to filter a matrix so the only cells remaining are those divisible by 4 in rows that sum to 10 or higher. Expressing this with a list comprehension does not require a lot of code, but it is extremely difficult to read:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]
filtered = [[x for x in row if x % 4 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)
```

```
>>>
[[4], [8]]
```

Although this example is a bit convoluted, in practice you'll see situations arise where such comprehensions seem like a good fit. I strongly encourage you to avoid using list, dictionary, or set comprehensions that look like this. The resulting code is very difficult for new readers to understand. The potential for confusion is especially great with a dictionary comprehension since it already needs an extra parameter to represent both the key and the value for each item.

The rule of thumb is to avoid using more than two control subexpressions in a comprehension. This could be two conditions, two loops, or one condition and one loop. As soon as it gets more complicated than that, you should use normal `if` and `for` statements and write a helper function (see Item 43: "Consider Generators Instead of Returning Lists").

Things to Remember

- Comprehensions support multiple levels of loops and multiple conditions per loop level.
- Comprehensions with more than two control subexpressions are very difficult to read and should be avoided.

Item 42: Reduce Repetition in Comprehensions with Assignment Expressions

A common pattern with comprehensions—including list, dictionary, and set variants—is the need to reference the same computation in multiple places. For example, say that I'm writing a program to manage orders for a fastener company. As new orders come in from customers, I need to be able to tell them whether or not I can fulfill their orders. Concretely, imagine that I need to verify that a request is sufficiently in stock and above the minimum threshold for shipping (e.g., in batches of 8), like this:

```
stock = {
    "nails": 125,
    "screws": 35,
    "wingnuts": 8,
    "washers": 24,
}

order = ["screws", "wingnuts", "clips"]

def get_batches(count, size):
    return count // size

result = {}
for name in order:
    count = stock.get(name, 0)
    batches = get_batches(count, 8)
    if batches:
        result[name] = batches

print(result)
```

```
>>>
{'screws': 4, 'wingnuts': 1}
```

Here, I implement this looping logic more succinctly by using a dictionary comprehension (see Item 40: "Use Comprehensions Instead of `map` and `filter`" for best practices):

```
found = {name: get_batches(stock.get(name, 0), 8)
         for name in order
         if get_batches(stock.get(name, 0), 8)}
print(found)
```

```
>>>
{'screws': 4, 'wingnuts': 1}
```

Although this code is more compact, the problem with it is that the `get_batches(stock.get(name, 0), 8)` expression is repeated. This hurts readability by adding visual noise and is technically unnecessary. The duplication also increases the likelihood of introducing a bug if the two expressions aren't kept in sync. For example, here I've changed the first `get_batches` call to have `4` as its second parameter instead of `8`, which causes the results to be different:

```
has_bug = {name: get_batches(stock.get(name, 0), 4)  # Wrong
           for name in order
           if get_batches(stock.get(name, 0), 8)}

print("Expected:", found)
print("Found:   ", has_bug)
```

```
>>>
Expected: {'screws': 4, 'wingnuts': 1}
Found:    {'screws': 8, 'wingnuts': 2}
```

An easy solution to these problems is to use an assignment expression—often called the walrus operator—as part of the comprehension (see Item 8: "Prevent Repetition with Assignment Expressions" for background):

```
found = {name: batches for name in order
         if (batches := get_batches(stock.get(name, 0), 8))}
```

The assignment expression (`batches := get_batches(...)`) allows me to look up the value for each `order` key in the `stock` dictionary a single time, call `get_batches` once, and then store its corresponding value in the `batches` variable. I can then reference that variable elsewhere in the comprehension to construct the dictionary's contents instead of having to call `get_batches` a second time. Eliminating the redundant calls to `get` and `get_batches` may also improve performance by avoiding unnecessary computations for each item in `order`.