When to Use a List Comprehension in Python

About James Timmins James is a software consultant and Python developer. When he's not writing Python, he's usually writing about it in blog or book form. » More about James Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are: Aldren Geir Arne Jaya Joanna Mike

Python is famous for allowing you to write code that's elegant, easy to write, and almost as easy to read as plain English. One of the language's most distinctive features is the **list comprehension**, which you can use to create powerful functionality within a single line of code. However, many developers struggle to fully leverage the more advanced features of a list comprehension in Python. Some programmers even use them too much, which can lead to code that's less efficient and harder to read.

By the end of this tutorial, you'll understand the full power of Python list comprehensions and how to use their features comfortably. You'll also gain an understanding of the trade-offs that come with using them so that you can determine when other approaches are more preferable.

In this tutorial, you'll learn how to:

- Rewrite loops and `map()` calls as a **list comprehension** in Python
- **Choose** between comprehensions, loops, and `map()` calls
- Supercharge your comprehensions with **conditional logic**
- **Use comprehensions** to replace `filter()`
- **Profile** your code to solve performance questions

Free Download: Get a sample chapter from Python Tricks: The Book that shows you Python's best practices with simple examples you can apply instantly to write more beautiful + Pythonic code.

AD

How to Create Lists in Python

There are a few different ways you can create lists in Python. To better understand the trade-offs of using a list comprehension in Python, let's first see how to create lists with these approaches.

Using `for` Loops

The most common type of loop is the for loop. You can use a `for` loop to create a list of elements in three steps:

1. Instantiate an empty list.

2. Loop over an iterable or <u>range</u> of elements.
3. Append each element to the end of the list.

If you want to create a list containing the first ten perfect squares, then you can complete these steps in three lines of code:

```
>>>

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

Here, you instantiate an empty list, `squares` . Then, you use a `for` loop to iterate over `range(10)` . Finally, you multiply each number by itself and append the result to the end of the list.

Using `map()` Objects

<u>map()</u> provides an alternative approach that's based in <u>functional programming</u>. You pass in a function and an iterable, and `map()` will create an object. This object contains the output you would get from running each iterable element through the supplied function.

As an example, consider a situation in which you need to calculate the price after tax for a list of transactions:

```
>>>

>>> txns = [1.09, 23.56, 57.84, 4.56, 6.78]
>>> TAX_RATE = .08
>>> def get_price_with_tax(txn):
...     return txn * (1 + TAX_RATE)
>>> final_prices = map(get_price_with_tax, txns)
>>> list(final_prices)
[1.1772000000000002, 25.4448, 62.467200000000005, 4.9248, 7.322400000000001]
```

Here, you have an iterable `txns` and a function `get_price_with_tax()` . You pass both of these arguments to `map()` , and store the resulting object in `final_prices` . You can easily convert this map object into a list using `list()` .

Using List Comprehensions

List comprehensions are a third way of making lists. With this elegant approach, you could rewrite the `for` loop from the first example in just a single line of code:

```
>>>
```

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

Rather than creating an empty list and adding each element to the end, you simply define the list and its contents at the same time by following this format:

```
>>>
```

```
new_list = [expression for member in iterable]
```

Every list comprehension in Python includes three elements:

1. **expression** is the member itself, a call to a method, or any other valid expression that returns a value. In the example above, the expression `i * i` is the square of the member value.
2. **member** is the object or value in the list or iterable. In the example above, the member value is `i`.
3. **iterable** is a list, <u>set</u>, sequence, <u>generator</u>, or any other object that can return its elements one at a time. In the example above, the iterable is `range(10)`.

Because the **expression** requirement is so flexible, a list comprehension in Python works well in many places where you would use `map()`. You can rewrite the pricing example with its own list comprehension:

```
>>>
```

```
>>> txns = [1.09, 23.56, 57.84, 4.56, 6.78]
>>> TAX_RATE = .08
>>> def get_price_with_tax(txn):
...     return txn * (1 + TAX_RATE)
>>> final_prices = [get_price_with_tax(i) for i in txns]
>>> final_prices
[1.1772000000000002, 25.4448, 62.467200000000005, 4.9248, 7.322400000000001]
```

The only distinction between this implementation and `map()` is that the list comprehension in Python returns a list, not a map object.

Benefits of Using List Comprehensions

List comprehensions are often described as being more <u>Pythonic</u> than loops or `map()`. But rather than blindly accepting that assessment, it's worth it to understand the benefits of using a list comprehension in Python when compared to the alternatives. Later on, you'll learn about a few scenarios where the alternatives are a better choice.

One main benefit of using a list comprehension in Python is that it's a single tool that you can use in many different situations. In addition to standard <u>list creation</u>, list comprehensions can also be used for mapping and filtering. You don't have to use a different approach for each scenario.

This is the main reason why list comprehensions are considered **Pythonic**, as Python embraces simple, powerful tools that you can use in a wide variety of situations. As an added side benefit, whenever you use a list comprehension in Python, you won't need to remember the proper order of arguments like you would when you call `map()`.

List comprehensions are also more **declarative** than loops, which means they're easier to read and understand. Loops require you to focus on how the list is created. You have to manually create an empty list, loop over the elements, and add each of them to the end of the list. With a list comprehension in Python, you can instead focus on *what* you want to go in the list and trust that Python will take care of *how* the list construction takes place.

How to Supercharge Your Comprehensions

In order to understand the full value that <u>list comprehensions</u> can provide, it's helpful to understand their range of possible functionality. You'll also want to understand the changes that are coming to the list comprehension in Python 3.8.

Using Conditional Logic

Earlier, you saw this formula for how to create list comprehensions:

>>>

```
new_list = [expression for member in iterable]
```

While this formula is accurate, it's also a bit incomplete. A more complete description of the comprehension formula adds support for optional **conditionals**. The most common way to add <u>conditional logic</u> to a list comprehension is to add a conditional to the end of the expression:

>>>

```
new_list = [expression for member in iterable (if conditional)]
```

Here, your conditional statement comes just before the closing bracket.

Conditionals are important because they allow list comprehensions to filter out unwanted values, which would normally require a call to `filter()`:

>>>

```
>>> sentence = 'the rocket came back from mars'
>>> vowels = [i for i in sentence if i in 'aeiou']
>>> vowels
['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']
```

In this code block, the conditional statement filters out any characters in `sentence` that aren't a vowel.

The conditional can test any valid expression. If you need a more complex filter, then you can even move the conditional logic to a separate function:

```
>>>
```

```
>>> sentence = 'The rocket, who was named Ted, came back \
... from Mars because he missed his friends.'
>>> def is_consonant(letter):
...     vowels = 'aeiou'
...     return letter.isalpha() and letter.lower() not in vowels
>>> consonants = [i for i in sentence if is_consonant(i)]
['T', 'h', 'r', 'c', 'k', 't', 'w', 'h', 'w', 's', 'n', 'm', 'd', \
'T', 'd', 'c', 'm', 'b', 'c', 'k', 'f', 'r', 'm', 'M', 'r', 's', 'b', \
'c', 's', 'h', 'm', 's', 's', 'd', 'h', 's', 'f', 'r', 'n', 'd', 's']
```

Here, you create a complex filter `is_consonant()` and pass this function as the conditional statement for your list comprehension. Note that the member value `i` is also passed as an argument to your function.

You can place the conditional at the end of the statement for simple filtering, but what if you want to *change* a member value instead of filtering it out? In this case, it's useful to place the conditional near the *beginning* of the expression:

```
>>>
```

```
new_list = [expression (if conditional) for member in iterable]
```

With this formula, you can use conditional logic to select from multiple possible output options. For example, if you have a list of prices, then you may want to replace negative prices with `0` and leave the positive values unchanged:

```
>>>
```

```
>>> original_prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
>>> prices = [i if i > 0 else 0 for i in original_prices]
>>> prices
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

Here, your expression `i` contains a conditional statement, `if i > 0 else 0`. This tells Python to output the value of `i` if the number is positive, but to change `i` to `0` if the number is negative. If this seems overwhelming, then it may be helpful to view the conditional logic as its own function:

```
>>>
```

```
>>> def get_price(price):
...     return price if price > 0 else 0
>>> prices = [get_price(i) for i in original_prices]
>>> prices
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

Now, your conditional statement is contained within `get_price()`, and you can use it as part of your list comprehension expression.

Using Set and Dictionary Comprehensions

While the list comprehension in Python is a common tool, you can also create set and dictionary comprehensions. A **set comprehension** is almost exactly the same as a list comprehension in Python. The difference is that set comprehensions make sure the output contains no duplicates. You can create a set comprehension by using curly braces instead of brackets:

```
>>>
>>> quote = "life, uh, finds a way"
>>> unique_vowels = {i for i in quote if i in 'aeiou'}
>>> unique_vowels
{'a', 'e', 'u', 'i'}
```

Your set comprehension outputs all the unique vowels it found in `quote`. Unlike lists, sets don't guarantee that items will be saved in any particular order. This is why the first member of the set is `a`, even though the first vowel in `quote` is `i`.

Dictionary comprehensions are similar, with the additional requirement of defining a key:

```
>>>
>>> squares = {i: i * i for i in range(10)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

To create the `squares` dictionary, you use curly braces (`{}`) as well as a key-value pair (`i: i * i`) in your expression.

Using the Walrus Operator

Python 3.8 will introduce the <u>assignment expression</u>, also known as the **walrus operator**. To understand how you can use it, consider the following example.

Say you need to make ten requests to an API that will return temperature data. You only want to return results that are greater than 100 degrees Fahrenheit. Assume that each request will return different data. In this case, there's no way to use a list comprehension in Python to solve the problem. The formula `expression for member in iterable (if conditional)` provides no way for the conditional to assign data to a variable that the expression can access.

The **walrus operator** solves this problem. It allows you to run an expression while simultaneously assigning the output value to a variable. The following example shows how this is possible, using `get_weather_data()` to generate fake weather data:

```
>>>

>>> import random
>>> def get_weather_data():
...     return random.randrange(90, 110)
>>> hot_temps = [temp for _ in range(20) if (temp := get_weather_data()) >= 100]
>>> hot_temps
[107, 102, 109, 104, 107, 109, 108, 101, 104]
```

You won't often need to use the assignment expression inside of a list comprehension in Python, but it's a useful tool to have at your disposal when necessary.

When Not to Use a List Comprehension in Python

List comprehensions are useful and can help you write elegant code that's easy to read and debug, but they're not the right choice for all circumstances. They might make your code run more slowly or use more memory. If your code is less performant or harder to understand, then it's probably better to choose an alternative.

Watch Out for Nested Comprehensions

Comprehensions can be **nested** to create combinations of lists, dictionaries, and sets within a collection. For example, say a climate laboratory is tracking the high temperature in five different cities for the first week of June. The perfect data structure for storing this data could be a Python list comprehension nested within a dictionary comprehension:

```
>>>

>>> cities = ['Austin', 'Tacoma', 'Topeka', 'Sacramento', 'Charlotte']
>>> temps = {city: [0 for _ in range(7)] for city in cities}
>>> temps
{
    'Austin': [0, 0, 0, 0, 0, 0, 0],
    'Tacoma': [0, 0, 0, 0, 0, 0, 0],
    'Topeka': [0, 0, 0, 0, 0, 0, 0],
    'Sacramento': [0, 0, 0, 0, 0, 0, 0],
    'Charlotte': [0, 0, 0, 0, 0, 0, 0]
}
```

You create the outer collection `temps` with a dictionary comprehension. The expression is a key-value pair, which contains yet another comprehension. This code will quickly generate a list of data for each city in `cities`.

Nested lists are a common way to create **matrices**, which are often used for mathematical purposes. Take a look at the code block below:

```
>>>
```

```
>>> matrix = [[i for i in range(5)] for _ in range(6)]
>>> matrix
[
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4]
]
```

The outer list comprehension `[... for _ in range(6)]` creates six rows, while the inner list comprehension `[i for i in range(5)]` fills each of these rows with values.

So far, the purpose of each nested comprehension is pretty intuitive. However, there are other situations, such as **flattening** nested lists, where the logic arguably makes your code more confusing. Take this example, which uses a nested list comprehension to flatten a matrix:

```
>>>

matrix = [
...     [0, 0, 0],
...     [1, 1, 1],
...     [2, 2, 2],
... ]
>>> flat = [num for row in matrix for num in row]
>>> flat
[0, 0, 0, 1, 1, 1, 2, 2, 2]
```

The code to flatten the matrix is concise, but it may not be so intuitive to understand how it works. On the other hand, if you were to use `for` loops to flatten the same matrix, then your code will be much more straightforward:

```
>>>

>>> matrix = [
...     [0, 0, 0],
...     [1, 1, 1],
...     [2, 2, 2],
... ]
>>> flat = []
>>> for row in matrix:
...     for num in row:
...         flat.append(num)
...
>>> flat
[0, 0, 0, 1, 1, 1, 2, 2, 2]
```

Now you can see that the code traverses one row of the matrix at a time, pulling out all the elements in that row before moving on to the next one.

While the single-line nested list comprehension might seem more Pythonic, what's most important is to write code that your team can easily understand and modify. When you choose your approach, you'll have to make a judgment call based on whether you think the comprehension helps or hurts readability.

Choose Generators for Large Datasets

A list comprehension in Python works by loading the entire output list into memory. For small or even medium-sized lists, this is generally fine. If you want to sum the squares of the first one-thousand integers, then a list comprehension will solve this problem admirably:

```
>>>
```

```
>>> sum([i * i for i in range(1000)])
332833500
```

But what if you wanted to sum the squares of the first *billion* integers? If you tried then on your machine, then you may notice that your computer becomes non-responsive. That's because Python is trying to create a list with one billion integers, which consumes more memory than your computer would like. Your computer may not have the resources it needs to generate an enormous list and store it in memory. If you try to do it anyway, then your machine could slow down or even crash.

When the size of a list becomes problematic, it's often helpful to use a <u>generator</u> instead of a list comprehension in Python. A **generator** doesn't create a single, large data structure in memory, but instead returns an iterable. Your code can ask for the next value from the iterable as many times as necessary or until you've reached the end of your sequence, while only storing a single value at a time.

If you were to sum the first billion squares with a generator, then your program will likely run for a while, but it shouldn't cause your computer to freeze. The example below uses a generator:

```
>>>
```

```
>>> sum(i * i for i in range(1000000000))
333333332833333333500000000
```

You can tell this is a generator because the expression isn't surrounded by brackets or curly braces. Optionally, generators can be surrounded by parentheses.

The example above still requires a lot of work, but it performs the operations **lazily**. Because of lazy evaluation, values are only calculated when they're explicitly requested. After the generator yields a value (for example, `567 * 567`), it can add that value to the running sum, then discard that value and generate the next value (`568 * 568`). When the sum function requests the next value, the cycle starts over. This process keeps the memory footprint small.

`map()` also operates lazily, meaning memory won't be an issue if you choose to use it in this case:

```
>>>
```

```
>>> sum(map(lambda i: i*i, range(1000000000)))
333333332833333333500000000
```

It's up to you whether you prefer the generator expression or `map()`.

Profile to Optimize Performance

So, which approach is faster? Should you use list comprehensions or one of their alternatives? Rather than adhere to a single rule that's true in all cases, it's more useful to ask yourself whether or not performance **matters** in your specific circumstance. If not, then it's usually best to choose whatever approach leads to the cleanest code!

If you're in a scenario where performance is important, then it's typically best to **profile** different approaches and listen to the data. timeit is a useful library for timing how long it takes chunks of code to run. You can use `timeit` to compare the runtime of `map()`, `for` loops, and list comprehensions:

```
>>>
```

```
>>> import random
>>> import timeit
>>> TAX_RATE = .08
>>> txns = [random.randrange(100) for _ in range(100000)]
>>> def get_price(txn):
...     return txn * (1 + TAX_RATE)
...
>>> def get_prices_with_map():
...     return list(map(get_price, txns))
...
>>> def get_prices_with_comprehension():
...     return [get_price(txn) for txn in txns]
...
>>> def get_prices_with_loop():
...     prices = []
...     for txn in txns:
...         prices.append(get_price(txn))
...     return prices
...
>>> timeit.timeit(get_prices_with_map, number=100)
2.0554370979998566
>>> timeit.timeit(get_prices_with_comprehension, number=100)
2.3982384680002724
>>> timeit.timeit(get_prices_with_loop, number=100)
3.0531821520007725
```

Here, you define three methods that each use a different approach for creating a list. Then, you tell `timeit` to run each of those functions 100 times each. `timeit` returns the total time it took to run those 100 executions.

As the code demonstrates, the biggest difference is between the loop-based approach and `map()`, with the loop taking 50% longer to execute. Whether or not this matters depends on the needs of your application.

Conclusion

In this tutorial, you learned how to use a **list comprehension** in Python to accomplish complex tasks without making your code overly complicated.

Now you can:

- Simplify loops and `map()` calls with declarative **list comprehensions**
- Supercharge your comprehensions with **conditional logic**
- Create **set** and **dictionary** comprehensions
- Determine when code clarity or performance dictates an **alternative approach**

Whenever you have to choose a list creation method, try multiple implementations and consider what's easiest to read and understand in your specific scenario. If performance is important, then you can use profiling tools to give you actionable data instead of relying on hunches or guesses about what works the best.

Remember that while Python list comprehensions get a lot of attention, your intuition and ability to use data when it counts will help you write clean code that serves the task at hand. This, ultimately, is the key to making your code Pythonic!

🐍 Python Tricks 🖂

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```python
1  # How to merge two dicts
2  # in Python 3.5+
3
4  >>> x = {'a': 1, 'b': 2}
5  >>> y = {'b': 3, 'c': 4}
6
7  >>> z = {**x, **y}
8
9  >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

About **James Timmins**

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:
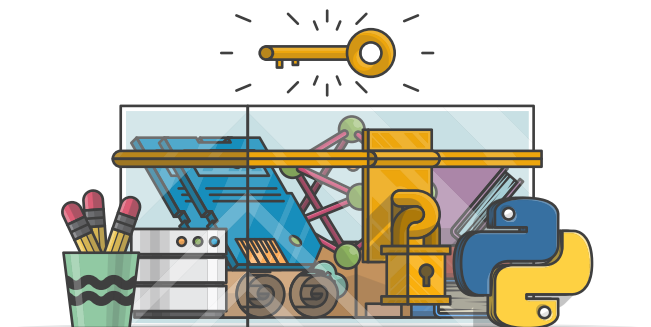
Aldren

Geir Arne

Jaya

Joanna

Mike

Master <u>Real-World Python Skills</u>
With Unlimited Access to Real Python

Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here. What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.