**A Guide to Python Comprehensions**

**Learn the intricacies of list comprehensions (listcomps), set comprehensions (setcomps), dictionary comprehensions (dictcomps) — and even generator expressions**



Photo by [Zuzana Ruttkay](https://unsplash.com/es/@zuzi_ruttkay?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

Comprehensions constitute probably the most popular syntactic sugar in Python. They are so powerful that when I finally understood them, they blew my mind, and I love using them ever since.

Yes, you got that right. The first time I *understood* comprehensions, not *saw* them. This is because, at first glance, they do not seem that readable. Many people making their first steps in Python find them neither clear nor understandable. Some may prefer for loops; [some others may prefer the map() function](https://medium.com/towards-data-science/does-python-still-need-the-map-function-96787ea1fb05); still others may prefer whatever else will work — just not comprehensions.

If you’re among them, I hope this article will convince you that comprehensions can come in handy in different situations, sometimes being much more readable than any other tool that Python offers. If you aren’t among them, I hope this article will help you learn more about how comprehensions work; this can help you both understand and create both simple and advanced comprehensions. In addition, we will discuss a significant aspect of comprehensions: when a comprehension cross the too-difficult line, in which case you should give up using it. Sometimes, in order to do that, you have to be able to convince yourself to forego a particular comprehension, so advanced that using it might show you as an experienced Python programmer.

This is because in some situations, a different tool can do better. We’ll dig deep into the practicalities of comprehensions, and I hope you will not only learn from this article but also enjoy reading it. I will try to write everything you need to know about them, which does not mean I will write everything about them. But once you’ve understood their essentials and want to learn more, you will have sufficient knowledge to do that.

Thus, this article aims to explain what Python comprehensions are, and how and when to use them. It also provides some significant intricacies of comprehensions and of using them.

I will also introduce a new term, a Python incomprehension. While the general Python codebase based is full of both comprehensions and incomprehensions, you should strive for a reasonable number of the former and none of the latter.

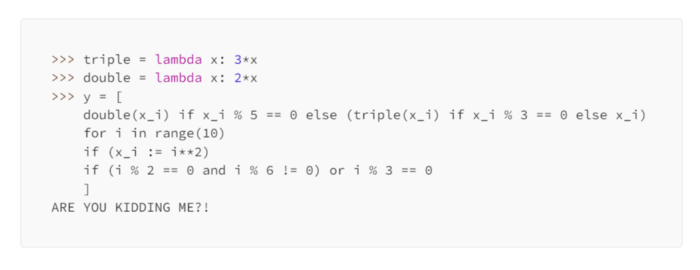
**Introduction to comprehensions**

The first thing about comprehensions I’d like you to understand is the meaning of the word *comprehension*. While I had known the meaning of the word for years before even learning Python comprehensions, I failed to put this meaning in the Python context. [The Cambridge Dictionary](https://dictionary.cambridge.org/dictionary/english/comprehension) provides the following definition of the word *comprehension*:

the ability to understand completely and be familiar with a situation, facts…

Now, this is something I’d like you to remember — this is *exactly* what Python comprehensions do: they help understand the action the code is doing. So, you can do this action in another way, but this action is done via a comprehension, it helps understand what’s happening. It helps both their *author* and *reader*.

Nevertheless, be aware of the word [*incomprehension*](https://dictionary.cambridge.org/dictionary/english/incomprehension), the antonym of *comprehension*. While Python does not offer *incomprehensions*, when we overdo with comprehensions and make them unreadable, they will become *incomprehensions .* The image below introduces the term *Python incomprehension*, which is a Python comprehension turned into something incomprehensible; that is, to something that goes against of what comprehensions exist for: to help understand the action the code is doing. Instead, incomprehensions make it difficult to understand what the code is to be doing.



A Python incomprehension, meaning “*don’t try to understand me!”*. Image by author.

We will discuss how to keep our comprehensions comprehensible, instead of making them incomprehensible; how to *not* turn them into Python incomprehensions; and so, in turn, how to make your code Pythonic, easy and understandable.

I treat comprehensions as syntactic sugar that enables me to use cleaner and shorter code instead of a different coding tool, such as a for loop, to create a list, dictionary, set, or generator expression. As you see, you can use comprehensions to create objects of four different types. Before going further, let’s analyze a simple example, similar to those you’ve probably seen in many other resources.

In the first example, I will show you how to use the simplest type of comprehension to create a list, a set, a generator expression, and a dictionary. Later on, we will work mainly with lists or dictionaries, depending on what we will be discussing. Generator expressions, although created using the same syntactic-sugar syntax, offer quite different behavior, so I will save them for another article, dedicated to them and their comparison with list comprehensions.

*List comprehensions (aka listcomps)*

Imagine there are no comprehensions in Python, and we want to do the following. We have a list of numerical values, say x, and we want to create a list that contains squared x values. Let’s do it:

>>> x = [1, 2, 3]  
>>> x\_squared = []  
>>> for x\_i in x:  
... x\_squared.append(x\_i\*\*2)  
>>> x\_squared  
[1, 4, 9]

Now let’s return to reality: Python does offer list comprehensions. We can achieve the same with much shorter and, for that matter, much more understandable code:

>>> x = [1, 2, 3]  
>>> x\_squared = [x\_i\*\*2 for x\_i in x]  
>>> x\_squared  
[1, 4, 9]

The list comprehension here is [x\_i\*\*2 for x\_i in x]. You can read it as follows: calculate x\_i\*\*2 for each value in x, and return the new values in a list.

To compare lengths of both approaches, let me define the comprehension ratio:

n\_for / n\_comprehension

where n\_for stands for the number of characters in a for loop while n\_comprehension the number of characters in the corresponding comprehension. To make this fair, I will use only one-character names. The ratio can be provided in percentages.

The comprehension ratio shows how shorter comprehension code is compared to that of the for loop. *Warning*: It represents only one aspect of the story: a comprehension’s brevity. While for some examples this brevity is a good thing, in some others it is not, as the code can become too difficult to understand. We will use the ratio anyway, so that you have the idea of how shorter the comprehension is compared to the for loop counterpart. Besides, sometimes another coding tool can do better than a for loop, e.g., [the map() function](https://towardsdatascience.com/does-python-still-need-the-map-function-96787ea1fb05).

In this simple example, the original for loop required 26 characters (without spaces); the list comprehension, 15 characters. Just to make this clear, I counted characters in the following texts:

for loop:  
y=[]foriinx:y.append(i\*\*2)  
  
list comprehension:  
y=[i\*\*2foriinx]

In this very example, the comprehension ratio was 173%. Not bad!

*Set comprehensions (aka setcomps)*

>>> x = [1, 2, 3]  
>>> x\_squared = set()  
>>> for x\_i in x:  
... x\_squared.add(x\_i\*\*2)  
>>> x\_squared  
{1, 4, 9}

Here’s the corresponding set comprehension:

>>> x = [1, 2, 3]  
>>> x\_squared = {x\_i\*\*2 for x\_i in i}  
>>> x\_squared  
{1, 4, 9}

The comprehension ratio is 186%.

*Dictionary comprehensions (aka dict comprehensions, dictcomps)*

>>> x = [1, 2, 3]  
>>> x\_squared = {}  
>>> for x\_i in x:  
... x\_squared[x\_i] = x\_i\*\*2  
>>> x\_squared  
{1: 1, 2: 4, 3: 9}

A dict comprehension becomes now:

>>> x = [1, 2, 3]  
>>> x\_squared = {x\_i: x\_i\*\*2 for x\_i in x}  
>>> x\_squared  
{1: 1, 2: 4, 3: 9}

Its comprehension ratio is smaller than it was for listcomps and setcomps, as it’s 124%.

*Generator expressions*

You may wonder, what does the term *generator expression* has to do with comprehensions? Should it be something like *generator comprehension*?

A valid point. I think… I think it could be. To me, it’d make much sense. But that’s how we call it: *generator expressions*. They’re here because generator expressions are created using the same syntactic sugar as the other comprehensions, though their behavior is very different. I will not discuss this difference in behavior, as this topic is far too important to hide it inside a general text on comprehensions. Hence, I will only show generator expressions here, with a promise that I will write more about them later.

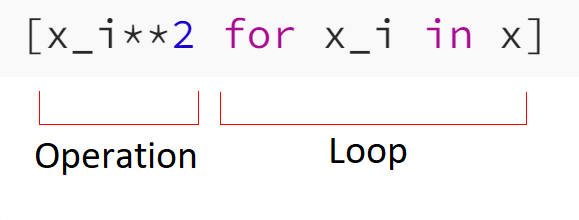
To create a generator expression, it’s enough to take the code inside the square brackets ([]) surrounding listcomp code, and surround it with parentheses (()):

>>> x = [1, 2, 3]  
>>> x\_squared = (x\_i\*\*2 for x\_i in x)

Here, x\_squared is a generator expression created based on a list. Easy to see that a comprehension ratio of a generator expression is the same as that of the corresponding list comprehension.

**Extending comprehensions**

The above comprehensions were the simplest one, since they included an operation done for each element of an iterable. By *operation* I mean this part of the comprehension: x\_i\*\*2; and whatever else we are doing with x\_i — see the image below, which explains what *operation* is. We can extend this part, but not only that; there are quite a lot of possibilities to extend Python comprehensions, and this is where the power of this tool comes from.



The structure of a list comprehension. Image by author.

The following list shows how we can extend the simplest comprehensions. We can do so using

* one or more functions in the operation
* one or more filters for the original data, using the if statement
* one or more filters for the output of the operation, using the if statement
* using a conditional expression in the operation, to use one or more filters for the original data; or to use one or more filters for the output of the operation
* using advanced looping
* the combinations of the above

This is where things can get complicated, and it’s our task to ensure that our comprehensions do not turn into incomprehensions. Before discussing why, let’s jump into examples of the above scenarios. Analyze each example and, if possible, run it in your Python interpreter, especially when you’re new to Python comprehensions.

https://miro.medium.com/max/700/1*xH6QGdiZkx0M-UOaXZK5BA.png

The classification above aims to help you *understand how comprehensions work*. It’s not formal, and to be honest, you do not even need to remember it. I use it to show you how different and how powerful comprehensions can be. So, analyze the examples, understand them, and if you think they can help you, try to remember them.

https://miro.medium.com/max/369/1*Hmjgga08e00oI6NQWIfEkg.png

**Using function(s) in the operation**

>>> def square(x):  
... return x\*\*2  
>>> x = [1, 2, 3]  
>>> x\_squared\_list = [square(x\_i) for x\_i in x]  
>>> x\_square\_list  
[1, 4, 9]  
>>> x\_squared\_dict = {x\_i: square(x\_i) for x\_i in x}  
{1: 1, 2: 4, 3: 9}

Read the [square(x\_i) for x\_i in x] comprehension, for instance, as: calculate square(x\_i) for each value in x and return the results as a list.

Above, we used one function. We can use more:

>>> def multiply(x, m):  
... return x\*m  
>>> def square(x):  
... return x\*\*2  
>>> x = [1, 2, 3]  
>>> x\_squared\_list = [multiply(square(x\_i), 3) for x\_i in x]  
>>> x\_squared\_dict = {x\_i: multiply(square(x\_i), 3) for x\_i in x}  
>>> x\_square\_list  
[3, 12, 27]  
>>> x\_squared\_dict = {x\_i: square(x\_i) for x\_i in x}  
{1: 3, 2: 12, 3: 27}

From now on, I will present only list comprehensions. I hope at this point you know how the different types of comprehensions work, so there is no point in repeating them over and over again and clutter the code that way.

**Filtering the original data**

>>> x = [1, 2, "a", 3, 5]  
>>> integers = [x\_i for x\_i in x if isinstance(x\_i, int)]  
>>> integers  
[1, 2, 3, 5]

We create here the list from x, by taking only integers (so, when x\_i has the type of int, that is, if isinstance(x\_i, int)).

Do you see how similar this version is to the previous one? It’s because both of them use the if statement to filter the original data; they just do it in a slightly different way. Previously, we added it to the operation; here, we added it after the loop.

This and the previous versions filter the original data, in contrast to what we will do in the next point, that is, filter the results of the operation. In other words, here you could achieve the same by first filtering the data and applying the list comprehension for the filtered data. In the version below, you would first use the list comprehension and then filter its values.

**Filtering the output of the operation**

>>> x = [1, 2, 3, 4, 5, 6]  
>>> x\_squared = [x\_i\*\*2 for x\_i in x if x\_i\*\*2 % 2 == 0]

Here, we keep only those results that are even, and reject odd ones.

Note that this code is not perfect: we run the same operation *twice*, first in the operation and then in the conditional expression (which starts with if). As of Python 3.8, we can improve this code using the walrus operator:

>>> x = [1, 2, 3, 4, 5, 6]  
>>> x\_squared = [y for x\_i in x if (y := x\_i\*\*2) % 2 == 0]

As you see, the walrus operator enables us to save half of the calculations in this list comprehension. That’s why, if you want to use more advanced comprehensions than the simplest one, you should become friends with the walrus operator.

For fun and to learn something, we can use [timeit](https://medium.com/towards-data-science/benchmarking-python-code-with-timeit-80827e131e48) to benchmark these two comprehensions, to see whether the walrus operator helps us indeed to improve the listcomp’s performance. For the benchmarks, I used code presented in the *Performance* section below, with the following code snippets:

setup = """x = list(range(1000))"""  
code\_comprehension = """[x\_i\*\*2 for x\_i in x if x\_i\*\*2 % 2 == 0]"""  
code\_alternative = """[y for x\_i in x if (y := x\_i\*\*2) % 2 == 0]"""

This is what I got on my machine (32 GB RAM, Windows 10, WSL 1):

Time for the comprehension : 26.3394  
Time for the alternative : 31.1244  
comprehension-to-alternative ratio: 0.846

Clearly, it’s definitely worth using the walrus operator — though the performance gain is not close 50%, as one might have anticipated and hoped. Once you know how this operator and this version of comprehensions work, you can make the code not only more performant, but also easier to understand.

This comprehension is a little more difficult to read than the previous ones, though the meaning is just as simple: calculate x\_i\*\*2 for each value in x and reject all even values of the output (x\_i\*\*2); return the results as a list.

As you see, you need to understand the intricacies of comprehensions to read this one, but once they do not pose problems to you, such a comprehension becomes easy to read. As you will see later, however, some comprehensions are anything but simple to read…

**Filtering the data using a conditional expression in the operation**

>>> x = [1, 2, "a", 3, 5]  
>>> res = [x\_i\*\*2 if isinstance(x\_i, int) else x\_i for x\_i in x]  
>>> res  
[1, 4, 'a', 9, 25]

This list comprehension should be understood as follows: For each value in x, use the following algorithm: when it’s an integer, square it; leave it untouched otherwise; collect the results in a list.

This version is quite similar to data filtering; in fact, it’s a specific type of data filtering, one that’s implemented as a conditional expression (x if condition else y) inside the operation (instead of outside of it). To be honest, you can easily rewrite this list comprehension using data filtering as presented above. I will not show it here, in order to avoid confusion. But you can consider this a good exercise: rewrite this comprehension using data filtering using the if condition outside of the operation, that is, after the comprehension loop.

*Note*: See also a note block in the next subsection, to learn when it’s fine to use a conditional expression and when it’s better to avoid it.

**Filtering the results using a conditional expression in the operation**

The above listcomp filtered the data; we can also use the conditional expression to filter the results of the operation, like here:

>>> x = [1, 2, 70, 3, 5]  
>>> res = [y if (y := x\_i\*\*2) < 10 else 10 for x\_i in x]  
>>> res  
[1, 4, 10, 9, 10]

We used the walrus operator again, in order to avoid using the same calculation twice, as here:

>>> res = [x\_i\*\*2 if x\_i\*\*2 < 10 else 10 for x\_i in x]

While this version is not *incorrect* (particularly for Python 3.7 and earlier), it is not performant.

https://miro.medium.com/max/700/1*xH6QGdiZkx0M-UOaXZK5BA.png

Since a conditional expression is added into the operation code, it can be useful when the condition is short. Otherwise, it will clutter the operation code. In such a case, it’s better to use the regular filtering, with the if condition presented after the comprehension loop.

https://miro.medium.com/max/700/1*2Y2kUU4KKM-rvvnaAfLN5g.png

**Using advanced looping**

You do not have to use only simple looping, as we did till now. For example, we can use enumerate() in a comprehension’s loop, the same way you would do it in a for loop:

>>> texts = ["whateveR! ", "\n\tZulugula RULES!\n", ]  
>>> d = {i: txt for i, txt in enumerate(texts)}  
>>> d  
{1: "whateveR! ", 2: "\n\tZulugula RULES!\n"}

Now, let’s loop over key-value pairs of the d dictionary:

>>> [(k, v.lower().strip()) for k, v in d.items()]  
[(1, 'whatever!'), (2, 'zulugula rules!')]

Basically, you can use any approach to looping that would work in a for loop.

**Combinations of the above scenarios**

The above scenarios were basic, and usually you need not think whether such comprehensions are or are not too difficult; they aren’t. Nevertheless, a situation can become far more difficult, especially when it includes more than one just operation or filtering; in other words, when it combines the above scenarios (including using the same scenario twice or even more in one comprehension).

If you’ve heard that Python comprehensions can become too complex, you’ve heard it right. That’s why you can often read that you should not overuse comprehensions; and that’s also more than true. This is what I have proposed the term *incomprehensions* for. You should *definitely* avoid using incomprehensions in your Python code.

Let’s consider a couple of examples so that you can see for yourself that comprehensions can get difficult. Not all of these examples will be too complex, though — but some definitely will.

Consider this listcomp, which combines filtering the data with filtering the output:

>>> x = range(50)  
>>> [y for x\_i in x if x\_i % 4!= 0 if (y := x\_i\*\*2) % 2 == 0]  
[4, 36, 100, 196, 324, 484, 676, 900, 1156, 1444, 1764, 2116]

Look what we’re doing here: calculate x\_i\*\*2 for all values of x that are divisible by 4, and take only those resulting output values that are divisible by 2.

We have two if conditions here. In that case, we can try to simplify the comprehension by joining the two conditions using and:

>>> [y for x\_i in x if x\_i % 4!= 0 and (y := x\_i\*\*2) % 2 == 0]  
[4, 36, 100, 196, 324, 484, 676, 900, 1156, 1444, 1764, 2116]

To be honest, for me two conditions presented in two if blocks seem easier to understand than one if block with two conditions joined with the and operator. If you disagree, so if in your opinion and helped increase readability, please share this in the comments. This *is* to some extent a subjective matter, so I do not expect all of you to agree with me.

Irrespective of what you think about using and, we can improve the readability of this comprehension by splitting it up into several rows, each row representing a block that *means something* and/or *does something*:

>>> [y for x\_i in x  
... if x\_i % 4 != 0  
... and (y := x\_i\*\*2) % 2 == 0]  
[4, 36, 100, 196, 324, 484, 676, 900, 1156, 1444, 1764, 2116]

Or

>>> [  
... y for x\_i in x  
... if x\_i % 4 != 0  
... and (y := x\_i\*\*2) % 2 == 0  
... ]  
[4, 36, 100, 196, 324, 484, 676, 900, 1156, 1444, 1764, 2116]

or even

>>> [  
... y  
... for x\_i in x  
... if x\_i % 4 != 0  
... and (y := x\_i\*\*2) % 2 == 0  
... ]  
[4, 36, 100, 196, 324, 484, 676, 900, 1156, 1444, 1764, 2116]

This last version is more useful when we apply one or more functions in the first row, so the row is longer than just one name, like here.

As I mentioned, two if blocks are more readable for me:

>>> [  
... y for x\_i in x  
... if x\_i % 4 != 0  
... if (y := x\_i\*\*2) % 2 == 0  
... ]  
[4, 36, 100, 196, 324, 484, 676, 900, 1156, 1444, 1764, 2116]

What I particularly like this this very version is the visual symmetry of the if blocks on the left side. Do you see it? If not, return to the previous block of code, and note the difference between this part:

... if x\_i % 4 != 0  
... if (y := x\_i\*\*2) % 2 == 0

and this part:

... if x\_i % 4 != 0  
... and (y := x\_i\*\*2) % 2 == 0

While these aspects are not too important, they help, at least for me. I mean, when I work on a longer comprehension, I take them into account — and I’d like to suggest you taking such aspects into account, too. First, believe me or not, this gives me fun. Second, when you do pay attention even to such small aspects of your comprehensions, you will feel you know them very well; you will feel they are (or are not, when they are not ready) ready, finished, complete.

This comprehension was not that difficult after all, was it? So, let’s make the situation — and hence the list comprehension — a little more complex:

>>> [  
... y for x\_i in x  
... if x\_i % 4 != 0 and x\_i % 5 != 0  
... if (y := square(multiply(x\_i, 3))) % 2 == 0  
... ]  
[36, 324, 1764, 2916, 4356, 6084, 10404, 12996, 15876, 19044]

This *is* more complex, but I think that put that way, even this comprehension is quite readable. Nevertheless, with more and more conditions being added, it’d be getting more and more complex — eventually, at some point, crossing the line of being *too complex*.

**Alternatives and examples**

As I already wrote, sometimes a for loop will be more readable than a comprehension. There are, however, more alternatives, such as the map() and filter() functions. You can read more about map() in [this article](https://medium.com/towards-data-science/does-python-still-need-the-map-function-96787ea1fb05), in which I explain why it’s good to know how this function works, and when it can be a better solution than a comprehension.

These functions can be even quicker than the corresponding comprehensions, so whenever performance counts, you may wish to check whether map()- and filter()-based code does not work better than the comprehension you’ve used.

In simple scenarios, comprehensions are usually better; in complicated ones, this does not have to be the case. It’s best to show this using examples. Thus, below I show a couple of examples of a comprehension (including generator expressions) and the corresponding alternative solution.

*Example 1.* The map() function instead of a simple generator expression.

>>> x = range(100)  
>>> gen = (square(x\_i) for x\_i in x)  
>>> other = map(square, x)

In this situation, both versions seem equally readable to me, but the comprehension looks somewhat more familiar. The map() function is an abstraction: you need to know how it works, which argument must goes first (the callable) and which must go then (the iterable)¹.

Unlike map(), in my eyes simple comprehensions, like the one above, do not look abstract. I can read them directly, from left to write: I am applying the function square() to x\_i, and x\_i’s are the subsequent elements of x; I store the results in a generator object.

Sounds easy, but I know that to read comprehensions like that (yes, I do read them that way!), you need to know this specific Python syntactic sugar. This is why for advanced (even intermediate) Python programmers, this generator expression (and other such simple comprehensions) look simple, but they seldom look simple for beginning Python developers. So, comprehensions can be abstract, too. As we will see soon, however, this syntax can be much clearer than its direct alternatives.

The use of the map() function becomes less clear when you use a lambda function inside map(). Compare:

>>> x = range(100)  
>>> gen = (x\_i\*\*2 for x\_i in x)  
>>> other = map(lambda x\_i: x\_\*\*2, x)

While lambdas do have their place in Python and can be highly useful, in some situations they can decrease readability — and this is such a situation. Here, I definitely prefer the generator expression over the map() version.

*Example 2*: The filter() function used instead of a generator expression with an if block for data filtering.

>>> x = range(100)  
>>> gen = (x\_i for x\_i in x if x\_i % 5 == 0)  
>>> other = filter(lambda x\_i: x\_i % 5 == 0, x)

All the above comments for map() apply to the filter() function. Like map(), it can be used with (like here) or without lambda. Here, again, I prefer the generator expression.

*Example 3*: Combining map() and filter() instead of a generator expression that uses a function to process the elements and an if block to filter data.

>>> x = range(100)  
>>> gen = (square(x\_i) for x\_i in x if x\_i % 5 == 0)  
>>> other = map(square, filter(lambda x\_i: x\_i % 5 == 0, x))

You can do the same in two steps:

>>> x\_filtered = filter(lambda x\_i: x\_i % 5 == 0, x)  
>>> other = map(square, x\_filtered)

This is one of the situations when you combine the map() and filter() functions in order to process the data and filter them. But it is also one of those situations in which I need some time to understand what this code is doing, at the same time being able to understand the corresponding comprehensions (here, a generator expression) almost *immediately*. Here, I’d *definitely* choose the generator expression version.

*Example 4:* Using map() with a wrapper function instead of a dictionary comprehension.

Dictionary comprehensions make creating dictionaries quite simple. Consider this example:

>>> def square(x):  
... return x\*\*2  
>>> y = {x: square(x) for x in range(5)}  
>>> y  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

This looks simple, doesn’t it? Now try to achieve the same using map().

The problem is that you need to create key-value pairs, so, if we want to use map(), it’s not enough for the function to return the output; it needs to return the key as well. A nice solution is to use a wrapper function around square():

>>> def wrapper\_square(x):  
... return x, square(x)  
>>> y = dict(map(wrapper\_square, range(5)))  
>>> y  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

Will you agree that the above dict comprehension was *much* more readable, *much* easier to write, and *much* easier to read than this map()-based version? Here, a for loop would be easier to understand than the map() version:

>>> y = {}  
>>> for x in range(5):  
... y[x] = square(x)  
>>> y  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

*Example 5*: A for loop for filtering the data combined with filtering the output.

Let’s return to the example in which we implemented quite a complex listcomp:

>>> [  
... y for x\_i in x  
... if x\_i % 4 != 0 and x\_i % 5 != 0  
... if (y := square(multiply(x\_i, 3))) % 2 == 0  
... ]  
[36, 324, 1764, 2916, 4356, 6084, 10404, 12996, 15876, 19044]

This is what the corresponding for loop looks here like:

>>> output = []  
>>> for x\_i in x:  
... y = square(multiply(x\_i, 3))  
... if y % 2 == 0 and x\_i % 4 != 0 and x\_i % 5 != 0:  
... output.append(y)  
>>> output  
[36, 324, 1764, 2916, 4356, 6084, 10404, 12996, 15876, 19044]

It’s not that bad, actually. This is I think a borderline situation, one in which some will still choose the listcomp version while others will consider it unreadable and choose the for loop instead. For me, the listcomp is still fine, but I am pretty sure for some it has already crossed the line of *too difficult*.

*Example 6*: Splitting a listcomp.

Let’s use the very same example as above. Let’s try to do something I am not a great fan of; namely, let’s split this complex comprehension into several ones, with the hope that this will simplify the code:

>>> y\_1 = [x\_i for x\_i in x]  
>>> y\_2 = [  
... y\_1\_i  
... for y\_1\_i in y\_1  
... if y\_1\_i % 4 != 0 and y\_1\_i% 5 != 0  
... ]  
>>> y = [  
... y\_final  
... for y\_2\_i in y\_2  
... if (y\_final := square(multiply(y\_2\_i, 3))) % 2 == 0  
... ]  
>>> y  
[36, 324, 1764, 2916, 4356, 6084, 10404, 12996, 15876, 19044]

Despite the splitting, this version is in my opinion even less readable than the previous one. It’s not only much longer; it is also less readable to me. I’d never choose this one, but I wanted to show it to you, just in case you have been wondering whether an approach like that would work. This time, it wouldn’t.

*Example 7*: Simplifying operations and filters using functions.

You can sometimes simplify a comprehension by exporting some of the work to be done to external functions. In our case, we can do so with both the operations and the filters:

>>> def filter\_data(x\_i):  
... return x\_i % 4 != 0 and x\_i % 5 != 0  
>>> def filter\_output(y\_i):  
... return y\_i % 2 == 0  
>>> def pipeline(x\_i):  
... return square(multiply(x\_i, 3))  
>>> def get\_generator(x):  
... return (  
... y for x\_i in x  
... if filter\_output(y := pipeline(x\_i))  
... and filter\_data(x\_i)  
.. )  
>>> list(get\_generator(x))  
[36, 324, 1764, 2916, 4356, 6084, 10404, 12996, 15876, 19044]

I think this makes the comprehension much simpler to read. We can do the same with the for loop, however:

>>> output = []  
>>> for x\_i in x:  
... y = pipeline(x\_i)  
... if filter\_output(y) and filter\_data(x\_i):  
... output.append(y)  
>>> output  
[36, 324, 1764, 2916, 4356, 6084, 10404, 12996, 15876, 19044]

But is this for loop better than the corresponding generator below (copied from the above snippet)?

>>> (  
... y for x\_i in x  
... if filter\_output(y := pipeline(x\_i))  
... and filter\_data(x\_i)  
... )

I’d choose the listcomp. It’s more concise and looks more readable, at least to me.

**Naming**

It’s a good moment to mention naming. Look once more at the function names used in the above comprehension: filter\_data, filter\_output and pipeline. These are clear names, ones that inform what they are responsible for; that way, they should help us understand what’s happening inside the comprehension.

Perhaps some would say that they are poor choices for the functions, as comprehensions should be short, and shorter function names would help us write more concise comprehensions. They would be right in only one thing: that the comprehensions would be more concise. But writing *short* comprehensions does not mean writing *good-because-readable* comprehensions. Compare the above comprehension with the one below, in which I used shorter but at the same time much less meaningful names:

>>> def fd(x\_i):  
... return x\_i % 4 != 0 and x\_i % 5 != 0  
>>> def fo(y\_i):  
... return y\_i % 2 == 0  
>>> def pi(x\_i):  
... return square(multiply(x\_i, 3))  
>>> def g\_g(x):  
... return (y for x\_i in x if fo(y := pi(x\_i)) and fd(x\_i))  
>>> list(get\_generator(x))  
[36, 324, 1764, 2916, 4356, 6084, 10404, 12996, 15876, 19044]

Definitely shorter — and definitely worse.

Note that while the author of such a comprehension will know that fd stands for filter\_data while pi stands for pipeline, will anyone else know it? Will the author remember it after a month? Or a year? I don’t think so. This is thus poor naming, one which carries no real connotation with what the functions are responsible for, and one whose secret code (fo stands for filter\_output) will quickly be forgotten.

If you decide to go for using external functions in comprehensions, do remember to use good names — good meaning *short and meaningful*. The same applies for all names used in a comprehension. Here, however, we should use names that are as short as possible, and not longer. There is an unwritten rule that when a variable has a narrow scope, it does not need a long name. While not always is this rule sensible, it often makes sense in comprehensions. For instance, consider these list comps:

>>> y = [square(x\_i) for x\_i in x if x\_i % 2 == 0]  
>>> y = [square(xi) for xi in x if xi % 2 == 0]  
>>> y = [square(p) for p in x if p% 2 == 0]  
>>> y = [square(element\_of\_x) for element\_of\_x in x if element\_of\_x % 2 == 0]

The first two work almost equally well for me, though I’d choose the first one. This is because x\_i looks like x with subscript i; xi does not look like that. So, I am ready to use the additional \_ character to join x and i, to carry this additional meaning. That way, x\_i looks sort of mathematically, as x\_i, meaning that x\_i belongs to x. While the xi version looks quite similar, it misses this nice resemblance of mathematical equations. Hence my choice of the former.

I do not like the third version. Why use p as a looping variable’s name? What does p mean? If it means something from a point of view of the algorithm we implement, then fine. But otherwise, it is not fine. Basically, it’s good to use names that mean something — but remember that in comprehensions, you should use *short* names that mean something.

In comprehensions, you should use *short* names that mean something.

This is why I don’t like the fourth version, either. While the name element\_of\_x does carry correct meaning, the name is unnecessarily long. Do not use a long name when a short one will be equally informative. This is the case here: x\_i is much shorter and at least as informative as element\_of\_x, if not more — thanks to how x\_i resonates with mathematical equations.

There’s one more thing. When you do *not* use the looping variable, use \_ as its name. For instance:

>>> template = [(0, 0) for \_ in range(10)]

No need to use i or whatever we want here.

**Too complex or not yet?**

With time and experience, you will learn that sometimes it’s difficult to decide whether a particular comprehension is still comprehensible or has reached, if not crossed, the *too-difficult* borderline. If you’re at a loss, it’s safer to choose the for loop, because it will be understood by almost everyone — while not everyone will be able to understand a complex listcomp.

Remember, however, to make the comprehension as easy as possible. For example, we did that by moving some or all of the calculations to well-named external functions (in the example above, these were pipeline(), filter\_data() and filter\_output()). Do not overuse this approach, however, since it’s more wordy than a pure comprehension. More importantly, it’s seldom a good idea to define functions that are used just once in the code.

The point is, anytime you’ve written a comprehension that looks complex, you should analyze it and decide whether it’s easy enough to understand or not. If not, replace it with a different approach. Sometimes generator pipelines based on function composition can be a good solution, as proposed [here](https://medium.com/towards-data-science/building-generator-pipelines-in-python-8931535792ff).

It’s not always easy to decide which comprehension is *too complex*. The number of operations does not have to be a good indicator, as sometimes a long comprehension can be much easier to understand than a shorter one, depending on what it aims to achieve and the naming used.

One more thing to remember is that a difficult comprehension means *difficult for the average developer*, not difficult for you. So, if you have written an extremely complex comprehension that you understand without problems, it does not mean you should go for it. This may be the case that you understand it because you spent three hours on writing these couple of lines of code! So, remember to make your comprehensions — and code, for that matter — understandable to others, too.

**Variable scope**

The way variable scope works in comprehensions makes them different from for loops — different in a good way: *The looping variable used in a comprehension is not visible in the comprehension’s outer scopes, nor does it overwrite variables from outer scopes*.

It’ll be easier to explain this using an example:

>>> x = [i\*\*2 for i in range(10)]  
>>> x  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
>>> i  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
NameError: name 'i' is not defined

Do you see what’s happening? Although we used i as a looping variable inside the list comprehension, it’s alive only within the scope of this comprehension. It’s not visible outside of this scope — it was deleted once the comprehension finished creating the list.

*The looping variable used in a comprehension is not visible in the comprehension’s outer scopes, nor does it overwrite variables from outer scopes*.

This is not all. As the below snippet shows, you can use a name inside a comprehension even if a variable with the same name is used in an outer scope. So, there will be two different objects with the same name at the same time! But these names will be visible only in their scopes².

>>> i = "this is i"  
>>> x = [i\*\*2 for i in range(10)]  
>>> x  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
>>> i  
'this is i'

Of course, you cannot use the outer-scope variable inside a comprehension if the looping variable has the same name. In other words, you do not have access to the outer-scope i from inside the list comprehension that uses an inner-scope i variable for looping (or for anything else).

*Warning*: Remember that this rule does not work with variables being assigned using [the walrus operator](https://towardsdatascience.com/the-walrus-operator-in-python-a315e4f84583):

>>> i = "this is i"  
>>> x = [i for j in range(10) if (i := j\*\*2) < 10]  
>>> x  
[0, 1, 4, 9]  
>>> i  
81

As you see, using the walrus operator puts the assigned variable (here, i) in the comprehension’s outer scope.

One more warning. Do not overuse this feature, as the resulting code can become difficult to read, like here:

>>> x = range(10)  
>>> x = [x\*\*2 for x in range(10)]  
>>> x  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

This goes too far.

This approach to variable scopes makes using comprehensions safe, as you do not have to worry that the comprehension’s looping variable will overwrite an important variable. As mentioned above, this is *not* how for loops work:

>>> i = "this is i"  
>>> for i in range(3):  
... print(i)  
...  
0  
1  
2  
>>> i  
2

As you see, if you use the same name in a for loop, a variable that was named the same way in the outer scope will be overwritten with the current value of the looping variable; eventually, with its last value.

**Performance**

One of the aspects many developers consider when deciding to use syntactic sugar is performance. Some say it’s not that important; others says it is.

My opinion is somewhere in between. Please don’t laugh at me, but I think that performance matters when… when performance matters. And when it does not, no need to take it into account.

performance matters when… when performance matters.

Even if this may sound a little funny, if not stupid, it makes perfect sense. If it does not matter whether or not your application is fast, then why should you worry about performance? In such a situation, it’s better to use other factors to choose coding style, like brevity and readability. However, when performance does matter, and especially when it matters a lot (for instance, in dashboards) — you can optimize your code at the expense of readability.

Another thing that matters is development time. If it matters, then perhaps you should not spend two full days optimizing the code that would enable you to save two seconds of the app execution time.

Okay, now that we know when to consider performance, let’s discuss how performant comprehensions are. Generally, performance strongly depends on the data and the comprehension itself, so if performance matters for a particular comprehension, you should benchmark this very comprehension (if this is possible).

Here, I will show you how to perform such benchmarking using the built-in [timeit](https://docs.python.org/3/library/timeit.html) module. You can read more about that from [this introductory article](https://medium.com/towards-data-science/benchmarking-python-code-with-timeit-80827e131e48). You can use the snippet below as a template, to conduct your own benchmarks. In the near future, however, I will write another article, dedicated to such benchmarks. They will be designed in a way so that we can draw deep conclusions about the performance of comprehensions.

# benchmark\_comprehension.py  
import timeit  
  
  
def benchmark(code\_comprehension: str,  
 code\_alternative: str,  
 setup: str,  
 n: int,  
 rep: int) -> None:  
 t\_comprehension = timeit.repeat(  
 code\_comprehension,  
 setup=setup,  
 number=n,  
 repeat=rep  
 )  
 t\_alternative = timeit.repeat(  
 code\_alternative,  
 setup=setup,  
 number=n,  
 repeat=rep  
 )  
 print(  
 "Time for the comprehension :"  
 f" {round(min(t\_comprehension), 4)}"  
 "\nTime for the alternative : "  
 f"{round(min(t\_alternative), 4)}"  
 "\ncomprehension-to-alternative ratio: "  
 f"{round(min(t\_comprehension)/min(t\_alternative), 3)}"  
 )  
  
# timeit settings  
n = 100  
rep = 7  
  
# code  
setup = """pass"""  
code\_comprehension = """  
for \_ in [x\*\*2 for x in range(1\_000\_000)]:  
 pass  
"""  
code\_alternative = """  
for \_ in map(lambda x: x\*\*2, range(1\_000\_000)):  
 pass  
"""  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 benchmark(code\_comprehension, code\_alternative, setup, n, rep)

In order to run this code, use the following shell command (working both in Windows and Linux):

$ python benchmark\_comprehension.py

This will run the benchmarks and print the results to the console. This one, for example, provided the following output on my machine:

Time for the comprehension : 20.3807  
Time for the alternative : 19.9166  
comprehension-to-alternative ratio: 1.023

You can change the following parts of the snippet:

* The setup code → change setup; this is code that is ran *before* running the benchmarked code.
* The code → change code\_comprehension and code\_alternative; note that the code is written in triple quotes, so you can split it into more lines; you can, however, use one-liners, too.
* The number of repeats, passed to timeit.repeat() → change rep.
* The number of times to run the code, passed to timeit.repeat() → change n.

(If you are not sure how number and repeat arguments work in timeit.repeat(), you will find this, and more, in [this article](https://towardsdatascience.com/benchmarking-python-code-with-timeit-80827e131e48).)

I made the code as simple as possible, which is why I did not use command-line arguments; just a bare application for benchmarking. You can, of course, improve it however you want.

**Against what should we benchmark comprehensions?**

This is perhaps the most important question we need to ask before conducting such benchmarks. The obvious but unsatisfactory answer is, against the best corresponding approach; it’s unsatisfactory because it does not really answer the question. Let’s think about that.

* *Generator expressions*

It’s easy to decide against what we should benchmark generator expressions: map() and filter() and/or their combinations, or against any other approach returning a generator. The two functions return map and filter objects, respectively, and they are generators, just like generator expressions.

* *List comprehensions*

One obvious approach to compare with is a for loop. We need to remember that in simple scenarios, list comprehensions will be more readable than the corresponding for loops; in complex scenarios, however, the opposite will usually be true.

Another approach can be the use of map() and/or filter(), with the subsequent use of list(). In that case, however, be aware that this is not a natural comparison, as when you need a list, you use a list comprehension, not a generator comprehension.

* *Dictionary comprehensions*

Like above, we have two most natural versions: the for loop and the map() and/or filter() functions. This time, however, using these two functions is a little tricky, as we need to use key-value pairs, so the function must return them, too. I’ve shown how to do this above, using a wrapper function. The truth is, nonetheless, that in this regard, dict comprehensions are much easier to use here.

* *Set comprehensions*

With sets, the situation is almost the same as with lists, so use the above suggestions for them.

**Some benchmarks**

As mentioned, we will not even attempt to analyze the performance of comprehensions here, as there is not an easy answer to a question as to whether they are more performant than their alternatives. The answer depends on the context. I want, however, to shed some light on this aspect of using comprehensions — at the very least to show you the basics.

In the following examples, I will provide

* the arguments n and rep to be passed to timeit.repeat() as number and repeat, respectively;
* setup, code\_comprehension and code\_alternative from the benchmark snippet provided above; and
* the output, as formatted in the example provided before.

*Benchmark 1*

n = 1\_000\_000  
rep = 7  
  
# code  
length = 10  
setup = """pass"""  
code\_comprehension = f"""y = [x\*\*2 for x in range({length})]"""  
code\_alternative = f"""  
y = []  
for x\_i in range({length}):  
 y.append(x\_i\*\*2)  
"""

Time for the comprehension : 1.6318  
Time for the alternative : 1.7296  
comprehension-to-alternative ratio: 0.943

The results for length = 100 (so for the lists of 100 elements) and n = 100\_000:

Time for the comprehension : 1.4767  
Time for the alternative : 1.6281  
comprehension-to-alternative ratio: 0.907

Now for length = 1000 (lists of 1000 elements) and n = 10\_000:

Time for the comprehension : 1.5612  
Time for the alternative : 1.907  
comprehension-to-alternative ratio: 0.819

And eventually, for length = 10\_000 (lists of 10\_000 elements) and n = 1000:

Time for the comprehension : 1.5692  
Time for the alternative : 1.9641  
comprehension-to-alternative ratio: 0.799

In the output, the most important element to look at is the ratio, as it’s unit-free while the times in the first two rows are not, depending on number (n in our case). As you see, the length of the constructed list makes a difference: the longer it is, the relatively slower the for loop is.

*Benchmark 2*

We will define a function length(x: Any) -> int, which

* returns 0 when x is an empty iterable or None;
* returns length of an object when it can be determined; and
* returns 1 otherwise.

Hence, for example, a number will have the length of 1. We will use this function to create a dictionary that has

* keys being values from the iterable; and
* values being a tuple of the length of an object, as defined above; and of the count of this element in the iterable x.

setup = """  
def length(x):  
 if not x:  
 return 0  
 try:  
 return len(x)  
 except:  
 return 1  
  
x = [1, 1, (1, 2, 3), 2, "x", "x", 1, 2, 1.1,  
 "x", 66, "y", 34, 34, "44", 690.222, "bubugugu", "44"]  
"""  
code\_comprehension = """  
y = {el: (length(el), x.count(el)) for el in set(x)}  
"""  
code\_alternative = """  
y = {}  
for el in set(x):  
 y[el] = (length(el), x.count(el))  
"""

Time for the comprehension : 0.5727  
Time for the alternative : 0.5736  
comprehension-to-alternative ratio: 0.998

When we made x consisting of 100 x lists, we got comprehension-to-alternative ratio: 1.009. As we see, the length of the list does not matter in this case: both methods are equally performant — though the dict comprehension is much shorter and elegant, but I think you need to know Python comprehensions to be able to appreciate them.

You need to know Python comprehensions to be able to appreciate them.

**Use comprehensions when you need them**

This may seem strange, but it can be tempting to use comprehensions even when you do not need the resulting object. This is one example of this:

>>> def get\_int\_and\_float(x):  
... [print(i) for i in x]  
... return [i for i in x if isinstance(i, (int, float))]  
>>> y = get\_int\_and\_float([1, 4, "snake", 5.56, "water"])  
1  
4  
snake  
5.56  
water  
>>> y  
[1, 4, 5.56]

Note what this function does: it takes an iterable and filters the data, by removing those which are not of int and float types. This use of list comprehension, which we see in the return line, is just fine. Nonetheless, we can see also another list comprehension used in the function: [print(i) for i in x]. This list comprehension only *prints* all the items, and the resulting list is neither stored nor used anyhow. So, what was it created for? Do we need it? Of course not. We should use a for loop instead:

>>> def get\_int\_and\_float(x):  
... for i in x:  
... print(i)  
... return [i for i in x if isinstance(i, (int, float))]  
>>> y = get\_int\_and\_float([1, 4, "snake", 5.56, "water"])  
1  
4  
snake  
5.56  
water  
>>> y  
[1, 4, 5.56]

In this example, the for loop is a natural approach.

It’s a very simple example, and it may look a little too simple. Let‘s consider another example, then. Imagine a function that reads text from a file, processes the text somehow, and writes the output (the processed text) to another file:

import pathlib  
  
def process\_text\_from(input\_path: pathlib.Path  
 output\_path: pathlib.Path) -> None:  
 # text is read from input\_path, as string;  
 # it is then processed somehow;  
 # the resulting processed\_text object (also string)  
 # is written to an output file (output\_path)  
  
def make\_output\_path(path: pathlib.Path):  
 return path.parent / path.name.replace(".", "\_out.")

Important: Note that the process\_text\_from() function does not return anything.

Next, we need to implement a function that runs process\_text\_from() for each element of an iterable of paths, input\_paths:

from typing import List  
  
def process\_texts(input\_paths: List[pathlib.Path]) -> None:  
 [process\_text\_from(p, make\_output\_path(p)) for p in input\_paths]

Again, this is *not* a fully functional list comprehension: we create a list that is neither stored nor used. If you see something like this, it’s a hint that you should unlikely use the comprehension. Think of another approach.

Here, again, a for loop seems to be the best choice:

def process\_texts(input\_paths: List[pathlib.Path]) -> None:  
 for p in input\_paths:  
 process\_text\_from(p, make\_output\_path(p))

**Conclusion**

Python is simple and readable, or at least this is what we’re told — and this is what most Pythonistas think of it. Both simplicity and readability result, at least partially, from syntactic sugar the language offers. Perhaps the most significant element of this syntactic sugar is comprehensions.

I associate comprehensions with elegance. They enable us to write concise and readable code — and elegant code. Sometimes the code is tricky, but usually it’s simple — at least for those who know Python. That’s why Python beginners have problems with understanding comprehensions, and thus appreciating them; and that’s why they start learning them as soon as they can.

I associate comprehensions with elegance. They enable us to write concise and readable code — and elegant code.

It’s a good idea, because comprehensions are everywhere in Python code. I cannot imagine an intermediate Python developer, not to mention an advanced Python developer, who does not write comprehensions. They are so natural for Python that there is no Python without comprehensions. At least I cannot — and do not want to — imagine Python without them.

This article aimed to introduce Python comprehensions: list, dictionary and set comprehensions; and generator expressions. The last type is quite different from the others, so maybe this is why its name is so different. Generator expressions have similar syntax to that of the other types, but they produce generators. Hence they deserve their own article, and I am going to write it soon.

Here is a summary along with several additional take-away thoughts about comprehensions:

1. Use comprehensions to facilitate the user understand what the code is responsible for.
2. When you need to create a list, use a list comprehension, not a different type of comprehension. The same way, when you need to create an object of a particular type, use the corresponding list comprehension: dictcomp for a dictionary, setcomp for a set, and generator expression for a generator.
3. Sometimes it may be tempting to use a particularly complicated comprehension. Some succumb to this temptation, hoping that in this way, they’ll show they can write complex and advanced Python code. Resist such temptation at any cost. Instead of showing off, write clear code; when a comprehension is not clear, resist the temptation of keeping it in the code anyway. Never turn your comprehensions into incomprehensions, just like never turn your back on readability.
4. When you see that a comprehension is getting more complicated than readable, it’s time to think about simplifying the code — either by simplifying the comprehension code, or by using an alternative solution. Maybe better names will do the job? Or splitting the comprehension into several lines? Or, if you’ve already done this, maybe you could try to change this splitting? If nothing works, try a different solution. Sometimes a for loop will work best; sometimes something else.
5. If you choose to use a comprehension even though you know it’s far too difficult, be aware that most advanced programmers will think you wanted to show off. I know how tempting this can be! I’ve fallen into such temptation more often than I am willing to admit. If only you’re aware of this problem and try to fight it off, with time and experience you will notice that it’s easier to withstand the temptation.
6. In some cases, you can replace an overly complex comprehension with a [generator pipeline](https://towardsdatascience.com/building-generator-pipelines-in-python-8931535792ff); it is in this situation that you will show quite *deep* understanding of Python programming, unlike in the previous one.
7. Remember how comprehension scope works. Use it for your purposes — but don’t overuse it.
8. When working on a comprehension, pay attention to every single detail of it. This includes whether or not you split it into more lines; the number of lines to split it to, and how to do it; whether to use if or and; whether the comprehension looks visually fine; and the like. The consideration of every tiny detail of your comprehensions can help you understand them far better.
9. Naming is significant, for several reasons: comprehensions are concise and thus delicate, one mistake being able to mess the whole comprehension; they often carry a lot of responsibility, like applying one or more functions, filtering data and/or output, and looping over an iterable; and they need to use at least one variable of a local scope (i.e., one that is *limited* to the comprehension’s scope). Thus, it makes a difference how you name your objects inside a comprehension and outside of it. Use the following recommendation for naming variables inside comprehensions: *use short and meaningful names*.
10. Do not use comprehensions when you do not need the resulting object.

Never turn your comprehensions into incomprehensions, just like never turn your back on readability.