**Python List Comprehension Tutorial**

Learn how to effectively use list comprehension in Python to create lists, to replace (nested) for loops and the map(), filter() and reduce() functions, ...!

When doing data science, you might find yourself wanting to read lists of lists, filtering column names, removing vowels from a list or flattening a matrix. You can easily use a lambda function or a for loop; As you well know, there are multiple ways to go about this. One other way to do this is by using list comprehensions.

This tutorial will go over this last topic:

* You'll first get a short recap of what [Python lists](https://www.datacamp.com/community/tutorials/python-list-comprehension#pythonlist) are and how they compare to other Python data structures;
* Next, you'll dive into [Python lists comprehensions](https://www.datacamp.com/community/tutorials/python-list-comprehension#plc): you'll learn more about the mathematics behind Python lists, how you can construct list comprehensions, how you can rewrite them as for loops or lambda functions, .... You'll not only read about this, but you'll also make some exercises!
* When you've got the basics down, it's also time to fine-tune your list comprehensions by adding [conditionals](https://www.datacamp.com/community/tutorials/python-list-comprehension#conditionals) to them: you'll learn how you can include conditionals in list comprehensions and how you can handle multiple if conditions and if-else statements.
* Lastly, you'll dive into [nested list comprehensions](https://www.datacamp.com/community/tutorials/python-list-comprehension#nlc) to iterate multiple times over lists.

If you're also interested in tackling list comprehensions together with iterators and generators? Check out DataCamp's [Python Data Science Toolbox](https://www.datacamp.com/courses/python-data-science-toolbox-part-2) course!

**Python Lists**

By now, you will have probably played around with values that had several data types. You have saved each and every value in a separate variable: each variable represents a single value. However, in data science, you'll often work with many data points, which will make it hard to keep on storing every value in a separate variable. Instead, you store all of these values in a Python list.

Lists are one of the four built-in data structures in Python. Other data structures that you might know are tuples, dictionaries and sets. A list in Python is different from, for example, int or bool, in the sense that it's a compound data type: you can group values together in lists. In fact, these values don't need to be of the same type: they can be a combination of boolean, String, integer, ... values.

Important to note here is that lists are ordered collections of items or objects. This makes lists in Python "sequence types", as they behave like a sequence. This means that they can be iterated; Other examples of sequences are Strings, tuples, or sets.

**Tip**: if you'd like to know more, test or practice your knowledge of Python lists, you can do so by going through the most common questions on Python lists [here](https://www.datacamp.com/community/tutorials/18-most-common-python-list-questions-learn-python).

Now, on a practical note: you build up a list with two square brackets; Inside these brackets, you'll use commas to separate your values. You can then assign your list to a variable. The values that you put in a Python list can be of any data type, even lists!

Take a look at the following example of a list:

eyJsYW5ndWFnZSI6InB5dGhvbiIsInNhbXBsZSI6IiMgQXNzaWduIGludGVnZXIgdmFsdWVzIHRvIGBhYCBhbmQgYGJgXG5hID0gNFxuYiA9IDlcblxuIyBDcmVhdGUgYSBsaXN0IHdpdGggdGhlIHZhcmlhYmxlcyBgYWAgYW5kIGBiYCBcbmNvdW50X2xpc3QgPSBbMSwyLDMsYSw1LDYsNyw4LGIsMTBdIn0=

**Tip**: build your own list in the IPython shell that is contained within the above DataCamp Light chunk!

**Python List Comprehension**

With the recap of the Python lists fresh in mind, you can easily see that defining and creating lists in Python can be a tiresome job: typing in all the values separately can take quite some time and you can easily make mistakes.

List comprehensions in Python are constructed as follows:

list\_variable = [x for x in iterable]

But how do you get to this formula-like way of building and using these constructs in Python? Let's dig a little bit deeper.

**List Comprehension in Python: The Mathematics**

Luckily, Python has the solution for you: it offers you a way to implement a mathematical notation to do this: list comprehension.

**Remember** in maths, the common ways to describe lists (or sets, or tuples, or vectors) are:

S = {x² : x in {0 ... 9}}

V = (1, 2, 4, 8, ..., 2¹²)

M = {x | x in S and x even}

In other words, you'll find that the above definitions actually tell you the following:

* The sequence S is actually a sequence that contains values between 0 and 9 included that are raised to the power of two.
* The sequence V, on the other hand, contains the value 2 that is raised to a certain power. For the first element in the sequence, this is 0, for the second this is 1, and so on, until you reach 12.
* Lastly, the sequence M contains elements from the sequence S, but only the even ones.

If the above definitions give you a headache, take a look at the actual lists that these definitions would produce:

S = {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

V = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096}

M = {0, 4, 16, 36, 64}

You clearly see the result of each list and the operations that were described in them!

Now that you've understood some of the maths behind lists, you can translate or implement the mathematical notation of constructing lists in Python using list comprehensions! Take a look at the following lines of code:

eyJsYW5ndWFnZSI6InB5dGhvbiIsInNhbXBsZSI6IlMgPSBbeCoqMiBmb3IgeCBpbiByYW5nZSgxMCldXG5WID0gWzIqKmkgZm9yIGkgaW4gcmFuZ2UoMTMpXVxuTSA9IFt4IGZvciB4IGluIFMgaWYgeCAlIDIgPT0gMF0ifQ==

This all looks very similar to the mathematical definitions that you just saw, right?

No worries if you're a bit at lost at this point; Even if you're not a math genius, these list comprehensions are quite easy if you take your time to study them. Take a second, closer look at the Python code that you see in the code chunk above.

You'll see that the code tells you that:

* The list S is built up with the square brackets that you read above in the first section. In those brackets, you see that there is an element x, which is raised to the power of 10. Now, you just need to know for how many values (and which values!) you need to raise to the power of 2. This is determined in range(10). Considering all of this, you can derive that you'll raise all numbers, going from 0 to 9, to the power of 2.
* The list V contains the base value 2, which is raised to a certain power. Just like before, now you need to know which power or i is exactly going to be used to do this. You see that i in this case is part of range(13), which means that you start from 0 and go until 12. All of this means that your list is going to have 13 values - those values will be 2 raised to the power 0, 1, 2, ... all the way up to 12.
* Lastly, the list M contains elements that are part of S if -and only if- they can be divided by 2 without having any leftovers. The modulo needs to be 0. In other words, the list M is built up with the equal values that are stored in list S.

Now that you see this all written out, it makes a lot more sense right?

**Recap And Practice**

In short, you see that there are a couple of elements coming back in all these lines of code:

* The square brackets, which are a signature of Python lists;
* The for keyword, followed by a variable that symbolizes a list item; And
* The in keyword, followed by a sequence (which can be a list!).

And this results in the piece of code which you saw at the beginning of this section:

list\_variable = [x for x in iterable]

Now it's your turn now to go ahead and get started with list comprehensions in Python! Let's stick close to the mathematical lists that you have seen before:

Q = {$x^3$: x in {0 ... 10}}

eyJsYW5ndWFnZSI6InB5dGhvbiIsInNhbXBsZSI6IiMgRGVmaW5lIFFcblEgPSBfX19fX19fX19fX19fX19fX19fX19fX19fIiwic29sdXRpb24iOiIjIERlZmluZSBRXG5RID0gW3gqKjMgZm9yIHggaW4gcmFuZ2UoMTEpXSIsInNjdCI6IkV4KCkudGVzdF9vYmplY3QoXCJRXCIpIn0=

**List Comprehension as an Alternative to...**

List comprehension is a complete substitute to for loops, lambda function as well as the functions map(), filter() and reduce(). What's more, for some people, list comprehension can even be easier to understand and use in practice! You'll read more about this in the next section!

However, if you'd like to know more about functions and lambda functions in Python, check out our [Python Functions Tutorial](https://www.datacamp.com/community/tutorials/functions-python-tutorial).

**For Loops**

As you might already know, you use for loops to repeat a block of code a fixed number of times. List comprehensions are actually good alternatives to for loops, as they are more compact. Consider the following example that starts with the variable numbers, defined as a range from 0 up until 10 (not included).

**Remember** that the number that you pass to the range() function is actually the number of integers that you want to generate, starting from zero, of course. This means that range(10) will return [0,1,2,3,4,5,6,7,8,9].

# Initialize `numbers`

numbers = range(10)

If you now want to perform an operation on every element in numbers, you can do this with a for loop, just like this one:

# Initialize `new\_list`

new\_list = []

# Add values to `new\_list`

for n in numbers:

if n%2==0:

new\_list.append(n\*\*2)

# Print `new\_list`

print(new\_list)

[0, 4, 16, 36, 64]

This is all nice and well, but now consider the following example of a list comprehension, where you basically do the same with a more compact notation:

# Create `new\_list`

new\_list = [n\*\*2 for n in numbers if n%2==0]

# Print `new\_list`

print(new\_list)

[0, 4, 16, 36, 64]

Let's study the difference in performance between the list comprehension and the for loop with a small test: you can set this up very quickly with the timeit library, which you can use to time small bits of Python code in a simple way. In this case, the small pieces of code that you will test are the for loop, which you will put in a function called power\_two() for your convenience, and the exact list comprehension which you have formulated above.

**Note** that you also pass in the number of executions you want to consider. In this case, that's set to 10000 in the number argument.

# Import `timeit`

import timeit

# Print the execution time

print(timeit.timeit('[n\*\*2 for n in range(10) if n%2==0]', number=10000))

0.05234622399802902

# Define `power\_two()`

def power\_two(numbers):

for n in numbers:

if n%2==0:

new\_list.append(n\*\*2)

return new\_list

# Print the execution time

print(timeit.timeit('power\_two(numbers)', globals=globals(), number=10000))

0.07795589299712447

**Note** that in this last piece of code, you also add the globals argument, which will cause the code to be executed within your current global namespace. This is extremely handy if you have a User-Defined Function (UDF) such as the power\_two() function in the above example. Alternatively, you can also pass a setup parameter which contains an import statement. You can read more about that [here](https://docs.python.org/3/library/timeit.html).

**Tip**: check out DataCamp's [Loops in Python](https://www.datacamp.com/community/tutorials/python-list-comprehension) tutorial for more information on loops in Python.

**Lambda Functions with map(), filter() and reduce()**

Lambda functions are also called "anonymous functions" or "functions without name". That means that you only use this type of functions when they are created. Lambda functions borrow their name from the lambda keyword in Python, which is used to declare these functions instead of the standard def keyword.

You usually use these functions together with the map(), filter(), and reduce() functions.

**How to Replace map() in Combination with Lambda Functions**

You can rewrite the combination map() and a lambda function just like in the example below:

# Initialize the `kilometer` list

kilometer = [39.2, 36.5, 37.3, 37.8]

# Construct `feet` with `map()`

feet = map(lambda x: float(3280.8399)\*x, kilometer)

# Print `feet` as a list

print(list(feet))

[128608.92408000001, 119750.65635, 122375.32826999998, 124015.74822]

Now, you can easily replace this combination of functions that define the feet variable with list comprehensions, taking into account the components which you have read about in the previous section:

* Start with the square brackets.
* Then add the body of the lambda function in those square brackets: float(3280.8399)\*x.
* Next, add the for keyword and make sure to repeat the sequence element x, which you already referenced by adding the body of the lambda function.
* Don't forget to specify where x comes from: add the in keyword, followed by the sequence from where you're going to get x. In this case, you'll transform the elements of the kilometerlist.

If you do all of this, you'll get the following result:

# Convert `kilometer` to `feet`

feet = [float(3280.8399)\*x for x in kilometer]

# Print `feet`

print(feet)

[128608.92408000001, 119750.65635, 122375.32826999998, 124015.74822]

**filter() and Lambda Functions to List Comprehensions**

Now that you have seen how easily you can convert the map() function in combination with a lambda function, you can also tackle code that contains the Python filter() function with lambda functions and rewrite that as well.

Consider the following example:

# Map the values of `feet` to integers

feet = list(map(int, feet))

# Filter `feet` to only include uneven distances

uneven = filter(lambda x: x%2, feet)

# Check the type of `uneven`

type(uneven)

# Print `uneven` as a list

print(list(uneven))

[122375, 124015]

To rewrite the lines of code in the above example, you can actually use two list comprehensions, stored in both the feet and uneven variables.

First, you rewrite the map() function, which you use to convert the elements of the feet list to integers. Then, you tackle the filter() function: you take the body of the lambda function, use the for and in keywords to logically connect x and feet:

# Constructing `feet`

feet = [int(x) for x in feet]

# Print `feet`

print(feet)

# Get all uneven distances

uneven = [x%2 for x in feet]

# Print `uneven`

print(uneven)

[128608, 119750, 122375, 124015]

[0, 0, 1, 1]

**Reduce reduce() and Lambda Functions in Python**

Lastly, you can also rewrite lambda functions that are used with the reduce() function to more compact lines of code. Take a look at the following example:

# Import `reduce` from `functools`

from functools import reduce

# Reduce `feet` to `reduced\_feet`

reduced\_feet = reduce(lambda x,y: x+y, feet)

# Print `reduced\_feet`

print(reduced\_feet)

[128608, 119750, 122375, 124015]

494748

**Note** that in Python 3, the reduce() function has been moved to the functools package. You'll therefore need to import the module to use it, just like in the code example above.

The chunk of code above is quite lengthy, isn't it?

Let's rewrite this piece of code!

Be careful! You need to take into account that you can't use y. List comprehensions only work with one only element, such as the x that you have seen throughout the many examples of this tutorial.

How are you going to solve this?

Well, in cases like these, aggregating functions such as sum() might come in handy:

# Construct `reduced\_feet`

reduced\_feet = sum([x for x in feet])

# Print `reduced\_feet`

print(reduced\_feet)

494748

**Note** that when you think about it, the use of aggregating functions when rewriting the reduce()function in combination with a lambda function makes sense: it's very similar to what you do in SQL when you use aggregating functions to limit the number of records that you get back after running your query. In this case, you use the sum() function to aggregate the elements in feetto only get back one definitive value!

Note that even though [this approach might not be as performant in SQL](https://www.datacamp.com/community/tutorials/sql-tutorial-query), this is definitely the way to go when you're working in Python!

**List Comprehensions with Conditionals**

Now that you have understood the basics of list comprehensions in Python, it's time to adjust the control flow of your comprehensions with the help of conditionals.

# Define `uneven`

uneven = [x/2 for x in feet if x%2==0]

# Print `uneven`

print(uneven)

[64304.0, 59875.0]

**Note** that you can rewrite the above code chunk with a Python for loop easily!

# Initialize and empty list `uneven`

uneven = []

# Add values to `uneven`

for x in feet:

if x % 2 == 0:

x = x / 2

uneven.append(x)

# Print `uneven`

print(uneven)

[64304.0, 59875.0]

**Multiple If Conditions**

Now that you have understood how you can add conditions, it's time to convert the following for loop to a list comprehension with conditionals.

divided = []

for x in range(100):

if x%2 == 0 :

if x%6 == 0:

divided.append(x)

Be careful, you see that the following for loop contains two conditions! Think carefully on how you're going to solve this.

divided = [x for x in range(100) if x % 2 == 0 if x % 6 == 0]

print(divided)

[0, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96]

**If-Else Conditions**

Of course, it's much more common to work with conditionals that involve more than one condition. That's right, you'll more often see if in combination with elif and else. Now, how do you deal with that if you plan to rewrite your code?

Take a look at the following example of such a more complex conditional in a for loop:

[x+1 if x >= 120000 else x+5 for x in feet]

[128609, 119755, 122376, 124016]

Now look at the following code chunk, which is a rewrite of the above piece of code:

for x in feet:

if x >= 120000:

x + 1

else:

x+5

You see that this is basically the same code, but restructured: the last for x in feet now initializes the for loop. After that, you add the condition if x >= 120000 and the line of code that you want to execute if this condition is True: x + 1. If the condition is False instead, the last bit of code in your list comprehension is executed: x+5.

**Nested List Comprehensions**

Apart from conditionals, you can also adjust your list comprehensions by nesting them within other list comprehensions. This is handy when you want to work with lists of lists: generating lists of lists, transposing lists of lists or flattening lists of lists to regular lists, for example, becomes extremely easy with nested list comprehensions.

Take a look at the following example:

list\_of\_list = [[1,2,3],[4,5,6],[7,8]]

# Flatten `list\_of\_list`

[y for x in list\_of\_list for y in x]

[1, 2, 3, 4, 5, 6, 7, 8]

You assign a rather simple list of list to a variable list\_of\_list. In the next line, you execute a list comprehension that returns a normal list. What actually happens is that you take the list elements ( y ) of the nested lists ( x ) in list\_of\_list and return a list of those list elements ythat are comprised in x.

You see that most of the keywords and elements that are used in the example of the nested list comprehension are similar to the ones that you used in the simple list comprehension examples:

* Square brackets
* Two for keywords, followed by a variable that symbolizes an item of the list of lists (x) and a list item of a nested list (y); And
* Two in keywords, followed by a list of lists (list\_of\_list) and a list item (x).

Most of the components are just used twice and you go one level higher (or deeper, depends on how you look at it!).

It takes some time to get used to, but it's rather simple, huh?

Let's now consider another example, where you see that you can also use two pairs of square brackets to change the logic of your nested list comprehension:

matrix = [[1,2,3],[4,5,6],[7,8,9]]

[[row[i] for row in matrix] for i in range(3)]

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

Now practice: rewrite the code chunk above to a nested for loop. If you need some pointers on how to tackle this exercise, go to one of the previous sections of this tutorial.

transposed = []

for i in range(3):

transposed\_row = []

for row in matrix:

transposed\_row.append(row[i])

transposed.append(transposed\_row)

You can also use nested list comprehensions when you need to create a list of lists that is actually a matrix. Check out the following example:

matrix = [[0 for col in range(4)] for row in range(3)]

matrix

[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

**Tip**: practice your loop skills in Python and rewrite the above code chunk to a nested for loop!

You can find the solution below.

for x in range(3):

nested = []

matrix.append(nested)

for row in range(4):

nested.append(0)

If you want to get some extra work done, work on translating this for loop to a while loop. You can find the solution below:

x = 0

matrix =[]

while x < 3:

nested = []

y = 0

matrix.append(nested)

x = x+1

while y < 4:

nested.append(0)

y= y+1

Lastly, it's good to know that you can also use functions such as int() to convert the entries in your feet list to integers. By encapsulating [int(x) for x in feet] within another list comprehension, you construct a matrix or lists of your list pretty easily:

[[int(x) for x in feet] for x in feet]

[[128608, 119750, 122375, 124015],

[128608, 119750, 122375, 124015],

[128608, 119750, 122375, 124015],

[128608, 119750, 122375, 124015]]

**Master Python for Data Science**

Congrats! You have made it to the end of this tutorial, in which you tackled list comprehensions, a mechanism that's frequently used in Python for data science. Now that you understand the workings of this mechanism, you're ready to also tackle dictionary, set, ... comprehensions!

Don't forget that you can practice your Python skills on a daily basis with DataCamp's daily practice mode! You can find it right on your dashboard. If you don't know the daily practice mode yet, read up [here](https://www.datacamp.com/community/blog/introducing-daily-practice-mode)!

Though list comprehensions can make our code more succinct, it is important to ensure that our final code is as readable as possible, so very long single lines of code should be avoided to ensure that our code is user friendly.