## List Comprehension as an Alternative

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 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 9.

Remember that the number that you pass to the range() function is 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 operate on every element in numbers, you can do this with a for loop, just like the one below:

# Initialize `new\_list`

new\_list = []

# Add values to `new\_list`

for n in numbers:

if n%2==0: #check if the element is even

new\_list.append(n\*\*2) #raise that element to the power of 2 and append to the list

# 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 do the same with a more compact notation:

# Create `new\_list`

new\_list = [n\*\*2 for n in numbers if n%2==0] #expression followed by for loop followed by the conditional clause

# Print `new\_list`

print(new\_list)

[0, 4, 16, 36, 64]

**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 a name". That means that you only use these types 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 that 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 or the expression you want to calculate: float(3280.8399)\*x.
* Next, add the for keyword and make sure to repeat the sequence element or the item x in the for loop, that you referenced by adding the body of the lambda function in the expression part.
* Finally, a usual for loop followed by an in keyword which will specify from where you are going to fetch the x known as a collection. In this case, you will transform the elements of the kilometer list.

If you do all of this, you'll get the following result which should ideally match the result of lambda function output:

# 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.

In the following example, you will filter out even numbers and only keep the odd numbers which are not divisible by 2:

# 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 use two list comprehensions:

* One to convert the values of feet to integers;
* Second to filter out even values from the feet list.

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 for x in feet if x%2!= 0]

# Print `uneven`

print(uneven)

[128608, 119750, 122375, 124015]

[122375, 124015]

#### 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)

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 only one 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. The sum() function would just do a usual sum over the entire list and assign the output to reduced\_feet list:

# Construct `reduced\_feet`

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

# Print `reduced\_feet`

print(reduced\_feet)

494748

Another way of aggregating the elements is by using the sum() function on the list feet. Hence, you don't need to use list comprehension in this case, as shown below:

sum(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 feet to 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 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 your comprehension control flow with the help of conditionals.

Though you have already seen a few examples where you made use of conditional statements like if clause, now you will delve more deeply into it.

Let's use a list comprehension which will make use of the if statement and to construct it you will:

* First, you will define the expression that will divide the expression by 2;
* Then, you will write a for loop which will iterate over the collection feet;
* Finally, you will write the if statement that will check whether the number is even or odd.

# 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, however, the code will be verbose!

# 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! To solve this, all you need to do is add two if conditions one followed by another. Only when both conditions are satisfied, the expression will be added to the list. In the following example, the expression x will be added as a multiple of 6.

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 a more complex conditional list comprehension:

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

[128609, 119755, 122376, 124016]

In the above line of code, you have two expressions:

* The first expression is dependent on the if statement;
* While the second expression is dependent on the else statement.

Rest everything else in the above example is pretty self-explanatory.

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 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, it 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 lists 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 y that are comprised in x.

You see that most of the keywords and elements used in the example of the nested list comprehension are similar to those 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)

transposed

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

You can also use nested list comprehensions when you need to create a list of lists that is 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]]