# Answer 2

# List Comprehensions in Python 3

### Introduction

**List comprehensions** offer a succinct way to create [lists](https://www.digitalocean.com/community/tutorials/understanding-lists-in-python-3) based on existing lists. When using list comprehensions, lists can be built by leveraging any [iterable](https://docs.python.org/3/glossary.html" \l "term-iterable), including [strings](https://www.digitalocean.com/community/tutorials/an-introduction-to-working-with-strings-in-python-3) and [tuples](https://www.digitalocean.com/community/tutorials/understanding-tuples-in-python-3).

Syntactically, list comprehensions consist of an iterable containing an expression followed by a for clause. This can be followed by additional for or if clauses, so familiarity with [for loops](https://www.digitalocean.com/community/tutorials/how-to-construct-for-loops-in-python-3) and [conditional statements](https://www.digitalocean.com/community/tutorials/how-to-write-conditional-statements-in-python-3-2) will help you understand list comprehensions better.

List comprehensions provide an alternative syntax to creating lists and other sequential [data types](https://www.digitalocean.com/community/tutorials/understanding-data-types-in-python-3). While other methods of iteration, such as for loops, can also be used to create lists, list comprehensions may be preferred because they can limit the number of lines used in your program.

## List Comprehensions

In Python, list comprehensions are constructed like so:

list\_variable = [x for x in iterable]

A list, or other iterable, is assigned to a variable. Additional variables that stand for items within the iterable are constructed around a for clause. The in keyword is used as it is in for loops, to iterate over the iterable.

Let’s look at an example that creates a list based on a string:

shark\_letters = [letter for letter in 'shark']

print(shark\_letters)

Here, the new list is assigned to the variable shark\_letters, and letter is used to stand in for the items contained in the iterable string 'shark'.

For us to confirm what the new list shark\_letters looks like, we call for it to print() and receive the following output:

Output

['s', 'h', 'a', 'r', 'k']

The list we created with the list comprehension is comprised of the items in the string 'shark', that is, one string for each letter.

List comprehensions can be rewritten as for loops, though not every for loop is able to be rewritten as a list comprehension.

Using our list comprehension that created the shark\_letters list above, let’s rewrite it as a for loop. This may help us better understand how the list comprehension works.

shark\_letters = []

for letter in 'shark':

shark\_letters.append(letter)

print(shark\_letters)

Copy

When creating a list with a for loop, the variable assigned to the list needs to be initialized with an empty list, as it is in the first line of our code block. The for loop then iterates over the item, using the variable letter in the iterable string 'shark'. Within the for loop, each item within the string is [added to the list with the list.append(x) method](https://www.digitalocean.com/community/tutorials/how-to-use-list-methods-in-python-3#listappend()).

Rewriting the list comprehension as a for loop provides us with the same output:

Output

['s', 'h', 'a', 'r', 'k']

List comprehensions can be rewritten as for loops, and some for loops can be rewritten to be list comprehensions to make code more succinct.

**Using Conditionals with List Comprehensions**

List comprehensions can utilize conditional statements to modify existing lists or other sequential data types when creating new lists.

Let’s look at an example of an if statement used in a list comprehension:

fish\_tuple = ('blowfish', 'clownfish', 'catfish', 'octopus')

fish\_list = [fish for fish in fish\_tuple if fish != 'octopus']

print(fish\_list)

Copy

The list comprehension uses the tuple fish\_tuple as the basis for the new list called fish\_list. The keywords of for and in are used, as they were in the [section above](https://www.digitalocean.com/community/tutorials/understanding-list-comprehensions-in-python-3#list-comprehensions), and now an if statement is added. The if statement says to only add those items that are not equivalent to the string 'octopus', so the new list only takes in items from the tuple that do not match 'octopus'.

When we run this, we’ll see that fish\_list contains the same string items as fish\_tuple except for the fact that the string 'octopus' has been omitted:

Output

['blowfish', 'clownfish', 'catfish']

Our new list therefore has every item of the original tuple except for the string that is excluded by the conditional statement.

We’ll create another example that uses [mathematical operators](https://www.digitalocean.com/community/tutorials/how-to-do-math-in-python-3-with-operators), [integers](https://www.digitalocean.com/community/tutorials/understanding-data-types-in-python-3#numbers), and the [range() sequence type](https://www.digitalocean.com/community/tutorials/how-to-construct-for-loops-in-python-3#for-loops-using-range()).

number\_list = [x \*\* 2 for x in range(10) if x % 2 == 0]

print(number\_list)

Copy

The list that is being created, number\_list, will be populated with the squared values of each item in the range from 0-9 if the item’s value is divisible by 2. The output is as follows:

Output

[0, 4, 16, 36, 64]

To break down what the list comprehension is doing a little more, let’s think about what would be printed out if we were just calling x for x in range(10). Our small program and output would then look like this:

number\_list = [x for x in range(10)]

print(number\_list)

Copy

Output

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Now, let’s add the conditional statement:

number\_list = [x for x in range(10) if x % 2 == 0]

print(number\_list)

Copy

Output

[0, 2, 4, 6, 8]

The if statement has limited the items in the final list to only include those items that are divisible by 2, omitting all of the odd numbers.

Finally, we can add the operator to have each x squared:

number\_list = [x \*\* 2 for x in range(10) if x % 2 == 0]

print(number\_list)

Copy

So each of the numbers in the previous list of [0, 2, 4, 6, 8] are now squared:

Output

[0, 4, 16, 36, 64]

You can also replicate [nested if statements](https://www.digitalocean.com/community/tutorials/how-to-write-conditional-statements-in-python-3-2#nested-if-statements) with a list comprehension:

number\_list = [x for x in range(100) if x % 3 == 0 if x % 5 == 0]

print(number\_list)

Copy

Here, the list comprehension will first check to see if the number x is divisible by 3, and then check to see if x is divisible by 5. If x satisfies both requirements it will print, and the output is:

Output

[0, 15, 30, 45, 60, 75, 90]

Conditional if statements can be used to control which items from an existing sequence are included in the creation of a new list.

**Nested Loops in a List Comprehension**

[Nested loops](https://www.digitalocean.com/community/tutorials/how-to-construct-for-loops-in-python-3#nested-for-loops) can be used to perform multiple iterations in our programs.

This time, we’ll look at an existing nested for loop construction and work our way towards a list comprehension.

Our code will create a new list that iterates over 2 lists and performs mathematical operations based on them. Here is our nested for loop code block:

my\_list = []

for x in [20, 40, 60]:

for y in [2, 4, 6]:

my\_list.append(x \* y)

print(my\_list)

Copy

When we run this code, we receive the following output:

Output

[40, 80, 120, 80, 160, 240, 120, 240, 360]

This code is multiplying the items in the first list by the items in the second list over each iteration.

To transform this into a list comprehension, we will condense each of the lines of code into one line, beginning with the x \* y operation. This will be followed by the outer for loop, then the inner for loop. We’ll add a print() statement below our list comprehension to confirm that the new list matches the list we created with our nested for loop block above:

my\_list = [x \* y for x in [20, 40, 60] for y in [2, 4, 6]]

print(my\_list)

Copy

Output

[40, 80, 120, 80, 160, 240, 120, 240, 360]

Our list comprehension takes the nested for loops and flattens them into one line of code while still creating the exact same list to assign to the my\_list variable.

List comprehensions provide us with a succinct way of making lists, enabling us to distill several lines of code into a single line. However, it is worth keeping in mind that the readability of our code should always take precedence, so when a list comprehension line becomes too long or unwieldy, it may be best to break it out into loops.

**Conclusion**

List comprehensions allow us to transform one list or other sequence into a new list. They provide a concise syntax for completing this task, limiting our lines of code.

List comprehensions follow the mathematical form of set-builder notation or set comprehension, so they may be particularly intuitive to programmers with a mathematical background.

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.