**List Comprehension**

**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#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.

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)**

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)**

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)**

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)**

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)**

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)**

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)**

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)

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)

Output:

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

**Generator Expression**

## Difference Between Iterable and Iterator

It will be easier to understand the concept of generators if you get the idea of iterables and iterators. Iterable is a “sequence” of data, you can iterate over using a loop. The easiest visible example of iterable can be a list of integers - [1, 2, 3, 4, 5, 6, 7]. It’s possible to iterate over other types of data like strings, dicts, tuples, sets, etc. Basically, any object that has iter() method can be used as an iterable. You can check it using hasattr() function in the interpreter.

>>> hasattr(str, '\_\_iter\_\_')

True

>>> hasattr(bool,'\_\_iter\_\_')

False

>>> simple\_list = [1,2,3]

>>> my\_iterator = iter(simple\_list)

>>> print(next(my\_iterator))

1

>>> print(next(my\_iterator))

2

## Generator Expressions

In Python, generators provide a convenient way to implement the iterator protocol. Generator is an iterable created using a function with a yield statement. The main feature of generator is evaluating the elements on demand. When you call a normal function with a return statement the function is terminated whenever it encounters a return statement. In a function with a yield statement the state of the function is “saved” from the last call and can be picked up the next time you call a generator function.

>>> gen\_exp = (x \*\* 2 for x in range(10) if x%2 == 0)

>>> for x in gen\_exp:

>>> print(x)

0

4

16

36

64

**Generators** are written just like a normal function but we use yield() instead of return() for returning a result. It is more powerful as a tool to implement iterators. It is easy and more convenient to implement because it offers the evaluation of elements on demand. Unlike regular functions which on encountering a return statement terminates entirely, generators use yield statement in which the state of the function is saved from the last call and can be picked up or resumed the next time we call a generator function.

## Type of data differs for list comprehensions and generator expressions

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

print(list\_comp)

gen\_exp = (x\*\*2 for x in range(10) if x%2 == 0)

print(gen\_exp)

[0, 4, 16, 36, 64]

<generator object <genexpr> at 0x012869F0>

## Generator takes less memory

from sys import getsizeof

my\_comp = [x\*5 for x in range(1000)]

my\_gen = (x\*5 for x in range(1000))

print(getsizeof(my\_comp))

print(getsizeof(my\_gen))

Output:

4516

48

**Lambda, Reduce, Map and Filter**

**The lambda operator** or **lambda function** is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created. Lambda functions are mainly used in combination with the functions filter(), map() and reduce(). The lambda feature was added to Python due to the demand from Lisp programmers.   
  
The general syntax of a lambda function is quite simple:

**lambda argument\_list: expression**

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.   
The following example of a lambda function returns the sum of its two arguments:

>>> f = lambda x, y : x + y

>>> f(1,1)

2

**The map() Function**

The advantage of the lambda operator can be seen when it is used in combination with the map() function.   
map() is a function with two arguments:

r = map(func, seq)

The first argument *func* is the name of a function and the second a sequence (e.g. a list) *seq*. *map()* applies the function *func* to all the elements of the sequence *seq*. It returns a new list with the elements changed by *func*

def fahrenheit(T):

return ((float(9)/5)\*T + 32)

def celsius(T):

return (float(5)/9)\*(T-32)

temp = (36.5, 37, 37.5,39)

F = map(fahrenheit, temp)

C = map(celsius, F)

In the example above we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius(). You can see this in the following interactive session:

>>> Celsius = [39.2, 36.5, 37.3, 37.8]

>>> Fahrenheit = map(lambda x: (float(9)/5)\*x + 32, Celsius)

>>> print(Fahrenheit)

[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]

>>> C = map(lambda x: (float(5)/9)\*(x-32), Fahrenheit)

>>> print(C)

[39.200000000000003, 36.5, 37.300000000000004, 37.799999999999997]

map() can be applied to more than one list. The lists have to have the same length. map() will apply its lambda function to the elements of the argument lists, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached:

>>> a = [1,2,3,4]

>>> b = [17,12,11,10]

>>> c = [-1,-4,5,9]

>>> map(lambda x,y:x+y, a,b)

[18, 14, 14, 14]

>>> map(lambda x,y,z:x+y+z, a,b,c)

[17, 10, 19, 23]

>>> map(lambda x,y,z:x+y-z, a,b,c)

[19, 18, 9, 5]

We can see in the example above that the parameter x gets its values from the list a, while y gets its values from b and z from list c.

**Filtering**

The function filter(function, list) offers an elegant way to filter out all the elements of a list, for which the function *function* returns True.   
The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False. This function will be applied to every element of the list *l*. Only if f returns True will the element of the list be included in the result list.

>>> fib = [0,1,1,2,3,5,8,13,21,34,55]

>>> result = filter(lambda x: x % 2, fib)

>>> print(result)

[1, 1, 3, 5, 13, 21, 55]

>>> result = filter(lambda x: x % 2 == 0, fib)

>>> print(result)

[0, 2, 8, 34]

**Reducing a List**

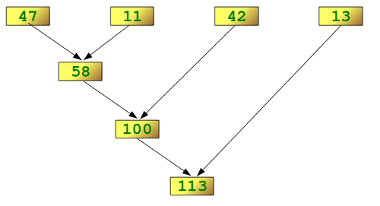
The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.   
  
If seq = [ s1, s2, s3, ... , sn ], calling reduce(func, seq) works like this:

* At first the first two elements of seq will be applied to func, i.e. func(s1,s2) The list on which reduce() works looks now like this: [ func(s1, s2), s3, ... , sn ]
* In the next step func will be applied on the previous result and the third element of the list, i.e. func(func(s1, s2),s3)  
  The list looks like this now: [ func(func(s1, s2),s3), ... , sn ]
* Continue like this until just one element is left and return this element as the result of reduce()

We illustrate this process in the following example:

>>> reduce(lambda x,y: x+y, [47,11,42,13])

113

The following diagram shows the intermediate steps of the calculation:   
 

**Examples of reduce()**

Determining the maximum of a list of numerical values by using reduce:

>>> f = lambda a,b: a if (a > b) else b

>>> reduce(f, [47,11,42,102,13])

102

>>>

Calculating the sum of the numbers from 1 to 100:

>>> reduce(lambda x, y: x+y, range(1,101))

5050