Python List Comprehension

A comprehension is a compact way of creating a Python data structure from iterators. With comprehensions, you can combine loops and conditional tests with a less verbose syntax.

Comprehension is considered more Pythonic and often useful in a variety of scenarios.

What is List Comprehension?

List comprehension sounds complex but it really isn't.

List comprehension is a way to build a new list by applying an expression to each item in an iterable.

It saves you having to write several lines of code, and keeps the readability of your code neat.

Basic Example

Suppose you want to create a list of all integer square numbers from 0 to 4. You could build that list by appending one item at a time to an empty list:

```
L = []
L.append(0)
L.append(1)
L.append(4)
L.append(9)
L.append(16)
print(L)
# Prints [0, 1, 4, 9, 16]
```

Or, you could just use an iterator and the range() function:

```
L = []
for x in range(5):
    L.append(x**2)
print(L)
# Prints [0, 1, 4, 9, 16]
```

Here both approaches produce the same result. However, a more Pythonic way to build a list is by using a list comprehension.

The general syntax for a list comprehension is:

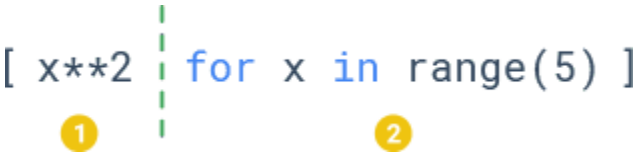
```
[expression for var in iterable]
```

Expression	*Var*	*Iterable*
It is evaluated once for each item in iterable	*It takes items from an iterable one by one*	*It's a collection of objects (like a list, tuple etc.)*

Here's how a list comprehension would build the above list:

```
L = [x**2 for x in range(5)]
print(L)
# Prints [0, 1, 4, 9, 16]
```

In the example above, list comprehension has two parts.



The first part collects the results of an expression on each iteration and uses them to fill out a new list.

The second part is exactly the same as the for loop, where you tell Python which iterable to work on. Every time the loop goes over the iterable, Python will assign each individual element to a variable x.

More Examples

Below are few examples of list comprehension.

Example 1

List comprehensions can iterate over any type of iterable such as lists, strings, files, ranges, and anything else that supports the iteration protocol.

Here's a simple list comprehension that uses string as an iterable.

```
L = [x*3 for x in 'RED']

print(L)
# Prints ['RRR', 'EEE', 'DDD']
```

Example 2

Following example applies `abs()` function to all the elements in a list.

```
# Convert list items to absolute values

vec = [-4, -2, 0, 2, 4]

L = [abs(x) for x in vec]

print(L)
# Prints [4, 2, 0, 2, 4]
```

Example 3

Following example calls a built-in method `strip()` on each element in a list.

```
# Remove whitespaces of list items

colors = ['  red', '  green ', 'blue  ']

L = [color.strip() for color in colors]

print(L)
# Prints ['red', 'green', 'blue']
```

Example 4

Following example creates a list of `(number, square)` tuples. Please note that, if a list comprehension is used to construct a list of tuples, the tuple values must be enclosed in parentheses.

```
L = [(x, x**2) for x in range(4)]

print(L)
# Prints [(0, 0), (1, 1), (2, 4), (3, 9)]
```

Here's why you should use list comprehension more often:

- List comprehensions are more concise to write and hence they turn out to be very useful in many contexts.
- Since a list comprehension is an expression, you can use it wherever you need an expression (e.g. as an argument to a function, in a return statement).
- List comprehensions run substantially faster than manual for loop statements (roughly twice as fast). It offers a major performance advantage especially for larger data sets.

List Comprehension with if Clause

A list comprehension may have an optional associated if clause to filter items out of the result.

Iterable's items are skipped for which the if clause is not true.

```
[expression for var in iterable if_clause]
```

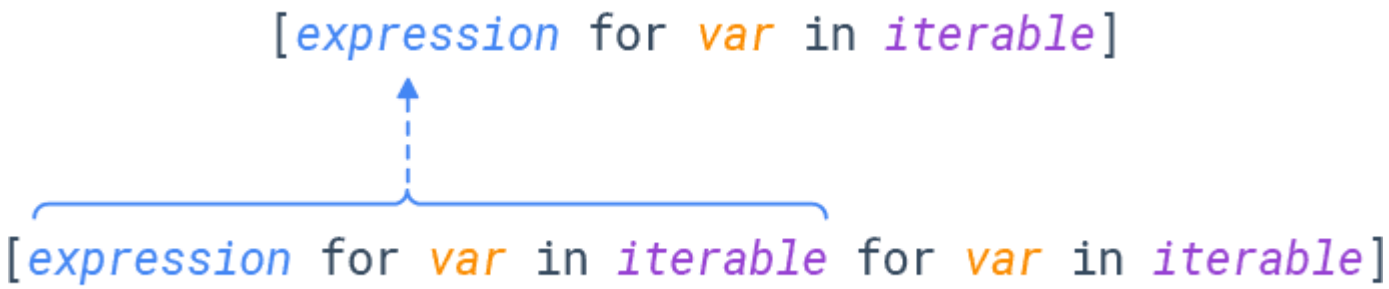
```
# Filter list to exclude negative numbers

vec = [-4, -2, 0, 2, 4]

L = [x for x in vec if x >= 0]

print(L)
# Prints [0, 2, 4]
```

This list comprehension is the same as a for loop that contains an [if statement]:

```
vec = [-4, -2, 0, 2, 4]
L = []
for x in vec:
    if x >= 0:
        L.append(x)
print(L)
# Prints [0, 2, 4]
```

Nested List Comprehensions

The initial expression in a list comprehension can be any expression, including another list comprehension.



For example, here's a simple list comprehension that flattens a nested list into a single list of items.

```
# With list comprehension
vector = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
L = [number for list in vector for number in list]
print(L)
# Prints [1, 2, 3, 4, 5, 6, 7, 8, 9]


# equivalent to the following plain, old nested loop:
vector = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
L = []
```

```
for list in vector:
    for number in list:
        L.append(number)
print(L)
# Prints [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Here's another list comprehension that transposes rows and columns.

```
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]
L = [[row[i] for row in matrix] for i in range(3)]
print(L)
# Prints [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

List Comprehension vs map() + lambda

When all you're doing is calling an already-defined function on each element, `map(f, L)` is a little faster than the corresponding list comprehension `[f(x) for x in L]`. Following example collects the ASCII codes of all characters in an entire string.

```
# With list comprehension
L = [ord(x) for x in 'foo']
print(L)
# Prints [102, 111, 111]


# With map() function
L = list(map(ord, 'foo'))
print(L)
# Prints [102, 111, 111]
```

However, when evaluating any other expression, `[some_expr for x in L]` is faster and clearer than `map(lambda x: some_expr, L)`, because the map incurs an extra function

call for each element. Following example creates a list of all integer square numbers.

```
# With list comprehension
L = [x ** 2 for x in range(5)]
print(L)
# Prints [0, 1, 4, 9, 16]


# With map() function
L = list(map((lambda x: x ** 2), range(5)))
print(L)
# Prints [0, 1, 4, 9, 16]
```

List Comprehension vs filter() + lambda

List comprehension with if clause can be thought of as analogous to the `filter()` function as they both skip an iterable's items for which the if clause is not true. Following example filters a list to exclude odd numbers.

```
# With list comprehension
L = [x for x in range(10) if x % 2 == 0]
print(L)
# Prints [0, 2, 4, 6, 8]


# With filter() function
L = list(filter((lambda x: x % 2 == 0), range(10)))
print(L)
# Prints [0, 2, 4, 6, 8]
```

As with `map()` function, `filter()` is slightly faster if you are using a built-in function.

List Comprehensions and Variable Scope

In Python 2, the iteration variables defined within a list comprehension remain defined even after the list comprehension is executed.

For example, in `[x for x in L]`, the iteration variable `x` overwrites any previously defined value of `x` and is set to the value of the last item, after the resulting list is created.

so, remember to use variable names that won't conflict with names of other local variables you have.

Fortunately, this is not the case in Python 3 where the iteration variable remains private, so you need not worry.