List Comprehensions All Versions

A list comprehension is a syntactical tool for creating lists in a natural and concise way, as illustrated in the following code to make a list of squares of the numbers 1 to 10: `[i ** 2 for i in range(1,11)]` The dummy `i` from an existing list `range` is used to make a new element pattern. It is used where a for loop would be necessary in less expressive languages.

Examples

Conditional List Comprehensions

Given a list comprehension you can append one or more `if` conditions to filter values.

```
[<expression> for <element> in <iterable> if <condition>]
```

For each `<element>` in `<iterable>` ; if `<condition>` evaluates to `True` , add `<expression>` (usually a function of `<element>`) to the returned list.

For example, this can be used to extract only even numbers from a sequence of integers:

```
[x for x in range(10) if x % 2 == 0]
# Out: [0, 2, 4, 6, 8]
```

Live demo

The above code is equivalent to:

```
even_numbers = []
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(l)
# Out: [0, 2, 4, 6, 8]
```

Also, a conditional list comprehension of the form `[e for x in y if c]` (where `e` and `c` are expressions in terms of `x`) is equivalent to `list(filter(lambda x: c, map(lambda x: e, y)))` .

Despite providing the same result, pay attention to the fact that the former example is almost 2x faster than the latter one. For those who are curious, this is a nice explanation of the reason why.

Note that this is quite different from the `... if ... else ...` conditional expression (sometimes known as a ternary expression) that you can use for the `<expression>` part of the list comprehension. Consider the following example:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Live demo

Here the conditional expression isn't a filter, but rather an operator determining the value to be used for the list items:

```
<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>
```

This becomes more obvious if you combine it with other operators:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Live demo

If you are using Python 2.7, `xrange` may be better than `range` for several reasons as described in the `xrange` documentation .

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

The above code is equivalent to:

```
numbers = []
for x in range(10):
    if x % 2 == 0:
        temp = x
    else:
```

```
        temp = -1
    numbers.append(2 * temp + 1)
print(l)
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

One can combine ternary expressions and `if` conditions. The ternary operator works on the filtered result:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

The same couldn't have been achieved just by ternary operator only:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', '*', 4, '*', 6, '*', 8, '*']
```

See also: *Filters* *, which often provide a sufficient alternative to conditional list comprehensions.* ***Note that filter() is discouraged in Python.***

List Comprehensions with Nested Loops

List Comprehensions can use nested `for` loops. You can code any number of nested for loops within a list comprehension, and each `for` loop may have an optional associated `if` test. When doing so, the order of the `for` constructs is the same order as when writing a series of nested `for` statements. The general structure of list comprehensions looks like this:

```
[ expression for target1 in iterable1 [if condition1]
             for target2 in iterable2 [if condition2]...
             for targetN in iterableN [if conditionN] ]
```

For example, the following code flattening a list of lists using multiple `for` statements:

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

can be equivalently written as a list comprehension with multiple `for` constructs:

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

Live Demo

In both the expanded form and the list comprehension, the outer loop (first for statement) comes first.

In addition to being more compact, the nested comprehension is also significantly faster.

```
In [1]: data = [[1,2],[3,4],[5,6]]
In [2]: def f():
   ...:     output=[]
   ...:     for each_list in data:
   ...:         for element in each_list:
   ...:             output.append(element)
   ...:     return output
In [3]: timeit f()
1000000 loops, best of 3: 1.37 µs per loop
In [4]: timeit [inner for outer in data for inner in outer]
1000000 loops, best of 3: 632 ns per loop
```

The overhead for the function call above is about *140ns* .

Inline `if` s are nested similarly, and may occur in any position after the first `for` :

```
data = [[1], [2, 3], [4, 5]]
output = [element for each_list in data
                if len(each_list) == 2
                for element in each_list
                if element != 5]
print(output)
# Out: [2, 3, 4]
```

Live Demo

For the sake of readability, however, you should consider using traditional *for-loops* . This is especially true when nesting is more than 2 levels deep, and/or the logic of the comprehension is too complex. multiple nested loop list comprehension could be error prone or it gives unexpected result.

Refactoring filter and map to list comprehensions

The `filter` or `map` functions should often be replaced by list comprehensions . Guido Van Rossum describes this well in an open letter in 2005 :

> `filter(P, S)` is almost always written clearer as `[x for x in S if P(x)]` , and this has the huge advantage that the most common usages involve predicates that are comparisons, e.g. `x==42` , and defining a lambda for that just requires much more effort for the reader (plus the lambda is slower than the list comprehension). Even more so for `map(F, S)` which becomes `[F(x) for x in S]` . Of course, in many cases you'd be able to use generator expressions instead.

The following lines of code are considered " *not pythonic* " and will raise errors in many python linters.

```
filter(lambda x: x % 2 == 0, range(10)) # even numbers < 10
map(lambda x: 2*x, range(10)) # multiply each number by two
reduce(lambda x,y: x+y, range(10)) # sum of all elements in list
```

Taking what we have learned from the previous quote, we can break down these `filter` and `map` expressions into their equivalent *list comprehensions* ; also removing the *lambda* functions from each - making the code more readable in the process.

```
# Filter:
# P(x) = x % 2 == 0
# S = range(10)
[x for x in range(10) if x % 2 == 0]

# Map
# F(x) = 2*x
# S = range(10)
[2*x for x in range(10)]
```

Readability becomes even more apparent when dealing with chaining functions. Where due to readability, the results of one map or filter function should be passed as a result to the next; with simple cases, these can be replaced with a single list comprehension. Further, we can easily tell from the list comprehension what the outcome of our process is, where there is more cognitive load when reasoning about the chained Map & Filter process.

```
# Map & Filter
filtered = filter(lambda x: x % 2 == 0, range(10))
results = map(lambda x: 2*x, filtered)

# List comprehension
results = [2*x for x in range(10) if x % 2 == 0]
```

Refactoring - Quick Reference

- **Map**

  ```
  map(F, S) == [F(x) for x in S]
  ```

- **Filter**

  ```
  filter(P, S) == [x for x in S if P(x)]
  ```

where `F` and `P` are functions which respectively transform input values and return a `bool`

Nested List Comprehensions

Nested list comprehensions, unlike list comprehensions with nested loops, are List comprehensions within a list comprehension. The initial expression can be any arbitrary expression, including another list comprehension.

```
#List Comprehension with nested loop
[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
#Out: [4, 5, 6, 5, 6, 7, 6, 7, 8]
```

```
#Out: [4, 5, 6, 5, 6, 7, 6, 7, 8]

#Nested List Comprehension
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]]
#Out: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

The Nested example is equivalent to

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

One example where a nested comprehension can be used it to transpose a matrix.

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

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

Like nested `for` loops, there is not limit to how deep comprehensions can be nested.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[['1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

Iterate two or more list simultaneously within list comprehension

For iterating more than two lists simultaneously within *list comprehension* , one may use zip() as:

```
>>> list_1 = [1, 2, 3 , 4]
>>> list_2 = ['a', 'b', 'c', 'd']
>>> list_3 = ['6', '7', '8', '9']

# Two lists
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# Three lists
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]

# so on ...
```

Syntax

```
[i for i in range(10)] # basic list comprehension
[i for i in xrange(10)] # basic list comprehension with generator object in python 2.x
[i for i in range(20) if i % 2 == 0] # with filter
[x + y for x in [1, 2, 3] for y in [3, 4, 5]] # nested loops
[i if i > 6 else 0 for i in range(10)] # ternary expression
[i for i in range(20) if i % 2 == 0] # with filter and ternary expression
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]] # nested list comprehension
```

Parameters

Remarks

List comprehensions were outlined in PEP 202 and introduced in Python 2.0.