Comprehensions

All Versions

List comprehensions in Python are concise, syntactic constructs. They can be utilized to generate lists from other lists by applying functions to each element in the list. The following section explains and demonstrates the use of these expressions.

Examples

List Comprehensions

A list comprehension creates a new `list` by applying an expression to each element of an iterable . The most basic form is:

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

There's also an optional 'if' condition:

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

Each `<element>` in the `<iterable>` is plugged in to the `<expression>` if the (optional) `<condition>` evaluates to true . All results are returned at once in the new list. Generator expressions are evaluated lazily, but list comprehensions evaluate the entire iterator immediately - consuming memory proportional to the iterator's length.

To create a `list` of squared integers:

```
squares = [x * x for x in (1, 2, 3, 4)]
# squares: [1, 4, 9, 16]
```

The `for` expression sets `x` to each value in turn from `(1, 2, 3, 4)` . The result of the expression `x * x` is appended to an internal `list` . The internal `list` is assigned to the variable `squares` when completed.

Besides a speed increase (as explained here), a list comprehension is roughly equivalent to the following for-loop:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
# squares: [1, 4, 9, 16]
```

The expression applied to each element can be as complex as needed:

```
# Get a list of uppercase characters from a string
[s.upper() for s in "Hello World"]
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']

# Strip off any commas from the end of strings in a list
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]
# ['these', 'words', 'mostly', 'have,commas']

# Organize letters in words more reasonably - in an alphabetical order
sentence = "Beautiful is better than ugly"
["".join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

---

else

`else` can be used in List comprehension constructs, but be careful regarding the syntax. The if/else clauses should be used before `for` loop, not after:

```
# create a list of characters in apple, replacing non vowels with '*'
# Ex - 'apple' --> ['a', '*', '*', '*' ,'e']

[x for x in 'apple' if x in 'aeiou' else '*']
#SyntaxError: invalid syntax

# When using if/else together use them before the loop
[x if x in 'aeiou' else '*' for x in 'apple']
#['a', '*', '*', '*', 'e']
```

Note this uses a different language construct, a conditional expression , which itself is not part of the comprehension syntax . Whereas the `if` after the `for…in` *is* a part of list comprehensions and used to *filter* elements from the source iterable.

Double Iteration

Order of double iteration `[... for x in ... for y in ...]` is either natural or counter-intuitive. The rule of thumb is to

Order of double iteration [... for x in ... for y in ...] is either natural or counter-intuitive. The rule of thumb is to follow an equivalent `for` loop:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)
```

This becomes:

```
[str(x)
    for i in range(3)
        for x in foo(i)
]
```

This can be compressed into one line as `[str(x) for i in range(3) for x in foo(i)]`

In-place Mutation and Other Side Effects

Before using list comprehension, understand the difference between functions called for their side effects (*mutating* , or *in-place* functions) which usually return `None` , and functions that return an interesting value.

Many functions (especially *pure* functions) simply take an object and return some object. An *in-place* function modifies the existing object, which is called a *side effect* . Other examples include input and output operations such as printing.

`list.sort()` sorts a list *in-place* (meaning that it modifies the original list) and returns the value `None` . Therefore it won't work as expected in a list comprehension:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

Instead, `sorted()` returns a sorted `list` rather than sorting in-place:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

Using comprehensions for side-effects is possible, such as I/O or in-place functions. Yet a for loop is usually more readable. While this works in Python 3:

```
[print(x) for x in (1, 2, 3)]
```

Instead use:

```
for x in (1, 2, 3):
    print(x)
```

In some situations, side effect functions *are* suitable for list comprehension. `random.randrange()` has the side effect of changing the state of the random number generator, but it also returns an interesting value. Additionally, `next()` can be called on an iterator.

The following random value generator is not pure, yet makes sense as the random generator is reset every time the expression is evaluated:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Avoid repetitive and expensive operations using conditional clause

Consider the below list comprehension:

```
>>> def f(x):
...     import time
...     time.sleep(.1)       # Simulate expensive function
...     return x**2

>>> [f(x) for x in range(1000) if f(x) > 10]
[16, 25, 36, ...]
```

This results in two calls to `f(x)` for 1,000 values of `x` : one call for generating the value and the other for checking the `if` condition. If `f(x)` is a particularly expensive operation, this can have significant performance implications. Worse, if calling `f()` has side effects, it can have surprising results.

Instead, you should evaluate the expensive operation only once for each value of `x` by generating an intermediate iterable (generator expression) as follows:

intermediate iterable (generator expression) as follows:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

Or, using the builtin map equivalent:

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Another way that could result in a more readable code is to put the partial result (`v` in the previous example) in an iterable (such as a list or a tuple) and then iterate over it. Since `v` will be the only element in the iterable, the result is that we now have a reference to the output of our slow function computed only once:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

However, in practice, the logic of code can be more complicated and it's important to keep it readable. In general, a separate generator function is recommended over a complex one-liner:

```
>>> def process_prime_numbers(iterable):
...     for x in iterable:
...         if is_prime(x):
...             yield f(x)
...
>>> [x for x in process_prime_numbers(range(1000)) if x > 10]
[11, 13, 17, 19, ...]
```

Another way to prevent computing `f(x)` multiple times is to use the `@functools.lru_cache()` (Python 3.2+) decorator on `f(x)` . This way since the output of `f` for the input `x` has already been computed once, the second function invocation of the original list comprehension will be as fast as a dictionary lookup. This approach uses memoization to improve efficiency, which is comparable to using generator expressions.

Say you have to flatten a list

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

Some of the methods could be:

```
reduce(lambda x, y: x+y, l)

sum(l, [])

list(itertools.chain(*l))
```

However list comprehension would provide the best time complexity.

```
[item for sublist in l for item in sublist]
```

The shortcuts based on + (including the implied use in sum) are, of necessity, O(L^2) when there are L sublists -- as the intermediate result list keeps getting longer, at each step a new intermediate result list object gets allocated, and all the items in the previous intermediate result must be copied over (as well as a few new ones added at the end). So (for simplicity and without actual loss of generality) say you have L sublists of I items each: the first I items are copied back and forth L-1 times, the second I items L-2 times, and so on; total number of copies is I times the sum of x for x from 1 to L excluded, i.e., I * (L**2)/2.

The list comprehension just generates one list, once, and copies each item over (from its original place of residence to the result list) also exactly once.

Dictionary Comprehensions

A dictionary comprehension is similar to a list comprehension except that it produces a dictionary object instead of a list.

A basic example:

Python 2.x ≥2.7

```
{x: x * x for x in (1, 2, 3, 4)}
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

which is just another way of writing:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

As with a list comprehension, we can use a conditional statement inside the dict comprehension to produce only the dict elements meeting some criterion.

Python 2.x ≥2.7

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# Out: {'Exchange': 8, 'Overflow': 8}
```

Or, rewritten using a generator expression.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

Starting with a dictionary and using dictionary comprehension as a key-value pair filter

Python 2.x ≥2.7

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# Out: {'x': 1}
```

Switching key and value of dictionary (invert dictionary)

If you have a dict containing simple *hashable* values (duplicate values may have unexpected results):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

and you wanted to swap the keys and values you can take several approaches depending on your coding style:

- `swapped = {v: k for k, v in my_dict.items()}`
- `swapped = dict((v, k) for k, v in my_dict.iteritems())`
- `swapped = dict(zip(my_dict.values(), my_dict))`
- `swapped = dict(zip(my_dict.values(), my_dict.keys()))`
- `swapped = dict(map(reversed, my_dict.items()))`

```
print(swapped)
# Out: {a: 1, b: 2, c: 3}
```

Python 2.x ≥2.3

If your dictionary is large, consider *importing itertools* and utilize `izip` or `imap` .

Merging Dictionaries

Combine dictionaries and optionally override old values with a nested dictionary comprehension.

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

However, dictionary unpacking (PEP 448) may be a preferred.

Python 3.x ≥3.5

```
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Note : dictionary comprehensions were added in Python 2.7+, unlike list comprehensions, which were added in 2.0. Before version 2.7, generator expression and the `dict()` function can be used to simulate the behavior of dictionary comprehensions.

Generator Expressions

Generator expressions are very similar to list comprehensions. The main difference is that it does not create a full set of results at once; it creates a generator object which can then be iterated over.

For instance, see the difference in the following code:

```
# list comprehension
[x**2 for x in range(10)]
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python 2.x ≥2.4

```
# generator comprehension
(x**2 for x in xrange(10))
# Output: <generator object <genexpr> at 0x11b4b7c80>
```

These are two very different objects:

- the list comprehension returns a `list` object whereas the generator comprehension returns a `generator` .
- `generator` objects cannot be indexed and makes use of the `next` function to get items in order.

Note : We use `xrange` since it too creates a generator object. If we would use range, a list would be created. Also, `xrange` exists only in later version of python 2. In python 3, `range` just returns a generator. For more information, see the *Differences between range and xrange functions* example .

Python 2.x ≥2.4

```
g = (x**2 for x in xrange(10))
print(g[0])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object has no attribute '__getitem__'
```

```
g.next()  # 0
g.next()  # 1
g.next()  # 4
...
g.next()  # 81

g.next()  # Throws StopIteration Exception
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Python 3.x ≥3.0

> NOTE: The function `g.next()` should be substituted by `next(g)` and `xrange` with `range` since `Iterator.next()` and `xrange()` do not exist in Python 3.

Although both of these can be iterated in a similar way:

```
for i in [x**2 for x in range(10)]:
    print(i)

"""
Out:
0
1
4
...
81
"""
```

Python 2.x ≥2.4

```
for i in (x**2 for x in xrange(10)):
    print(i)

"""
Out:
0
1
4
.
.
```

```
.
81
"""
```

Use cases

Generator expressions are lazily evaluated, which means that they generate and return each value only when the generator is iterated. This is often useful when iterating through large datasets, avoiding the need to create a duplicate of the dataset in memory:

```
for square in (x**2 for x in range(1000000)):
    #do something
```

Another common use case is to avoid iterating over an entire iterable if doing so is not necessary. In this example, an item is retrieved from a remote API with each iteration of `get_objects()`. Thousands of objects may exist, must be retrieved one-by-one, and we only need to know if an object matching a pattern exists. By using a generator expression, when we encounter an object matching the pattern.

```
def get_objects():
    """Gets objects from an API on by one"""
    while True:
        yield get_next_item()

def object_matches_pattern(obj):
    # perform potentially complex calculation
    return matches_pattern

def right_item_exists():
    items = (object_matched_pattern(each) for each in get_objects())
    for item in items:
        if item.is_the_right_one:


            return True
    return False
```

Set Comprehensions

Set comprehension is similar to list and dictionary comprehension, but it produces a set, which is an unordered collection of unique elements.

Python 2.x ≥2.7

```
# A set containing every value in range(5):
{x for x in range(5)}
# Out: {0, 1, 2, 3, 4}

# A set of even numbers between 1 and 10:
{x for x in range(1, 11) if x % 2 == 0}
# Out: {2, 4, 6, 8, 10}

# Unique alphabetic characters in a string of text:
text = "When in the Course of human events it becomes necessary for one people..."
{ch.lower() for ch in text if ch.isalpha()}
# Out: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',
#           'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

Live Demo

Keep in mind that sets are unordered. This means that the order of the results in the set may differ from the one presented in the above examples.

Note : Set comprehension is available since python 2.7+, unlike list comprehensions, which were added in 2.0. In Python 2.2 to Python 2.6, the `set()` function can be used with a generator expression to produce the same result:

Python 2.x ≥2.2

```
set(x for x in range(5))
# Out: {0, 1, 2, 3, 4}
```

Comprehensions involving tuples

The `for` clause of a list comprehension can specify more than one variable:

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [3, 7, 11]

[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]
# Out: [3, 7, 11]
```

This is just like regular `for` loops:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Note however, if the expression that begins the comprehension is a tuple then it must be parenthesized:

```
[x, y for x, y in [(1, 2), (3, 4), (5, 6)]]
# SyntaxError: invalid syntax

[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [(1, 2), (3, 4), (5, 6)]
```

Counting Occurrences Using Comprehension

When we want to count the number of items in an iterable, that meet some condition, we can use comprehension to produce an idiomatic syntax:

```
# Count the numbers in `range(1000)` that are even and contain the digit `9`:
print (sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
# Out: 95
```

The basic concept can be summarized as:

1. Iterate over the elements in `range(1000)` .
2. Concatenate all the needed `if` conditions.
3. Use `1` as *expression* to return a 1 for each item that meets the conditions.
4. Sum up all the `1` s to determine number of items that meet the conditions.

Note : Here we are not collecting the `1` s in a list (note the absence of square brackets), but we are passing the ones directly to the `sum` function that is summing them up. This is called a *generator expression* , which is similar to a Comprehension.

Changing Types in a List

Quantitative data is often read in as strings that must be converted to numeric types before processing. The types of all list items can be converted with either a List Comprehension or the `map()` function .

```
# Convert a list of strings to integers.
items = ["1","2","3","4"]
[int(item) for item in items]
# Out: [1, 2, 3, 4]

# Convert a list of strings to float.
items = ["1","2","3","4"]
map(float, items)
# Out:[1.0, 2.0, 3.0, 4.0]
```

Syntax

```
[x + 1 for x in (1, 2, 3)] # list comprehension, gives [2, 3, 4]
```

```
(x + 1 for x in (1, 2, 3)) # generator expression, will yield 2, then 3, then 4
```

```
[x for x in (1, 2, 3) if x % 2 == 0] # list comprehension with filter, gives [2]
```

```
[x + 1 if x % 2 == 0 else x for x in (1, 2, 3)] # list comprehension with ternary
```

```
[x + 1 if x % 2 == 0 else x for x in range(-3,4) if x > 0] # list comprehension with ternary and filtering
```

```
{x for x in (1, 2, 2, 3)} # set comprehension, gives {1, 2, 3}
```

```
{k: v for k, v in [('a', 1), ('b', 2)]} # dict comprehension, gives {'a': 1, 'b': 2} (python 2.7+ and 3.0+ only)
```

```
[x + y for x in [1, 2] for y in [10, 20]] # Nested loops, gives [11, 21, 12, 22]
```

```
[x + y for x in [1, 2, 3] if x > 2 for y in [3, 4, 5]] # Condition checked at 1st for loop
```

```
[x + y for x in [1, 2, 3] for y in [3, 4, 5] if x > 2] # Condition checked at 2nd for loop
```

```
[x for x in xrange(10) if x % 2 == 0] # Condition checked if looped numbers are odd numbers
```

Parameters

Remarks

Comprehensions are syntactical constructs which define data structures or expressions unique to a particular language. Proper use of comprehensions reinterpret these into easily-understood expressions. As expressions, they can be used:

- in the right hand side of assignments
- as arguments to function calls
- in the body of a lambda function
- as standalone statements. (For example: `[print(x) for x in range(10)]`)