COMPREHENSIONS

LIST COMPREHENSIONS

[endresult of whatever is done with each element for i in collection]

A list comprehension provides a compact way of **mapping a list into another list** by applying a function to each of the elements of the list.

```
>>> l = [1, 9, 8, 4]
>>> [elem * 2 for elem in l]
[2, 18, 16, 8]
>>> l
[1, 9, 8, 4]
>>> l = [elem * 2 for elem in l]
>>> l
[2, 18, 16, 8]
```

- To make sense of this, look at it from right to left. a_list is the list you're mapping. The Python interpreter loops through a_list one element at a time, temporarily assigning the value of each element to the variable elem. Python then applies the function elem * 2 and appends that result to the returned list.

- A list comprehension creates a new list; it does not change the original list.

- It is safe to assign the result of a list comprehension to the variable that you're mapping. Python constructs the new list in memory, and when the list comprehension is complete, it assigns the result to the original variable.

IF:

List comprehensions can also filter items, producing a result that can be smaller than the original list.

To filter a list, you can include an **if** clause at the end of the list comprehension. The expression after the **if** keyword will be evaluated for each item in the list. If the expression evaluates to True, the item will be included in the output.

```
>>> l = [1, 9, 8, 4]
>>> l = [elem * 2 for elem in l if elem % 2]
>>> l
[2, 18]
```

Nested loops: for

List comprehensions can become even more complex if we need them to—for instance, they may contain *nested loops*, coded as a series of for clauses. In fact, their full syntax allows for any number of for clauses, each of which can have an optional associated if clause.

For example, the following builds a list of the concatenation of x + y for every x in one string and every y in another. It effectively collects all the *ordered combinations* of the characters in two strings:

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Again, one way to understand this expression is to convert it to statement form by indenting its parts. The following is an equivalent, but likely slower, alternative way to achieve the same effect:

```
>>> res = []
>>> for x in 'abc':
        for y in 'lmn':
            res.append(x + y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Beyond this complexity level, though, list comprehension expressions can often become too compact for their own good. In general, they are intended for simple types of iterations; for more involved work, a simpler for statement structure will probably be easier to understand and modify in the future. As usual in programming, if something is difficult for you to understand, it's probably not a good idea.

DICTIONARY COMPREHENSIONS

A dictionary comprehension is like a list comprehension, but it **constructs a dictionary** instead of a list.

They run an implied loop, collecting the key/value results of expressions on each iteration and using them to fill out a new dictionary.

A loop variable allows the comprehension to use loop iteration values along the way.

Comprehensions actually require more code in this case, but they are also more general than this example implies—we can use them to map a single stream of values to dictionaries as well, and keys can be computed with expressions just like values:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}                    # Or: range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c * 4 for c in 'SPAM'}                           # Loop over any iterable
>>> D
{'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM'}

>>> d = {c: ord(c) for c in 'SPAM'}
>>> d
{'S': 83, 'P': 80, 'A': 65, 'M': 77}
>>>

>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
>>> D
{'eggs': 'EGGS!', 'spam': 'SPAM!', 'ham': 'HAM!'}
```

Dictionary comprehensions are also useful for initializing dictionaries from keys lists, in much the same way as the fromkeys method we met at the end of the preceding section:

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)          # Initialize dict from keys
>>> D
{'b': 0, 'c': 0, 'a': 0}

>>> D = {k:0 for k in ['a', 'b', 'c']}             # Same, but with a comprehension
>>> D
{'b': 0, 'c': 0, 'a': 0}

>>> D = dict.fromkeys('spam')                      # Other iterables, default value
>>> D
{'s': None, 'p': None, 'a': None, 'm': None}

>>> D = {k: None for k in 'spam'}
>>> D
{'s': None, 'p': None, 'a': None, 'm': None}
```

Like related tools, dictionary comprehensions support additional syntax not shown here, including nested loops and if clauses.

IF COMPREHENSION

trueStatement if condition else falseStatement

example:
When reporting the number of characters in a string, instead of printing “0 character(s)”, “1 character (s)” or 10 character(s), we could use a couple of conditional expressions to print “NO characters”, “1 character” or “10 characters”

```
>>> s='qwerty'
>>> print('{} character{}'.format((len(s) if len(s)>0 else 'No'),(" if len(s)==1 else 's')))
6 characters
>>> s=''
>>> print('{} character{}'.format((len(s) if len(s)>0 else 'No'),(" if len(s)==1 else 's')))
No characters
>>> s='a'
>>> print('{} character{}'.format((len(s) if len(s)>0 else 'No'),(" if len(s)==1 else 's')))
1 character
>>>
>>> s=input('string ??')
string ??qwerty
>>> s
'qwerty'
>>> print('{} character{}'.format((len(s) if len(s)>0 else 'No'),(" if len(s)==1 else 's')))
6 characters
>>>
>>>
>>>

>>> s=input('string ??')
string ??
>>> print('{} character{}'.format((len(s) if len(s)>0 else 'No'),(" if len(s)==1 else 's')))
No characters
>>>
>>> s=input('string ??')
string ??1
>>> print('{} character{}'.format((len(s) if len(s)>0 else 'No'),(" if len(s)==1 else 's')))
1 character
>>>
>>>
```

This will print “no characters”, “1 character”, “6 characters”, etc, which gives a much more professional impression.