Python’s List Comprehensions

Reading / References

- List comprehensions in Python’s docs[1] for processing lists
- Visual explanation of list comprehensions[2]
- Even more examples of list comprehensions[3]

Notes

As list comprehensions are an incredibly powerful way of processing lists in Python, they deserve a special mention. This short session aims to do just that.

All list comprehensions have one thing in common: They start with a list of items, and return a new list based on those items.

Simple list comprehensions

The simplest list comprehensions involve doing something simple with each element in the list. For instance if we have words, we could use a list comprehension to make them all uppercase:

```
words = ["The", "big", "bad", "Wolf"]
uppers = [ word.upper() for word in words ]
```

Don’t confuse the brackets in the second line with the literal list construction of the first line. The brackets on the second line form a *list comprehension*, which in its simplest form has the following shape (where the angle-bracketed elements are variable):

```
[<expression> for <x> in <list>]
```

This syntax expresses that a new list should be created by evaluating the `expression` once for each possible value of `x` taken from the `list`.

Every list comprehension can be done via a `for−in` loop, just not as elegantly. So the above list comprehension could be written as:

```
uppers = []
for word in words:
    uppers.append(word.upper())
```

Whenever you see a list comprehension, it essentially corresponds to some analogous “for” expression.

Here is a variation that replaces each word with a pair of the word in capitals and its length:

[1] https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

[2] http://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/

[3] http://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html

```
words = ["The", "big", "bad", "Wolf"]
pairs = [
    [word.upper(), len(word)]
    for word in words
]
# Reading back the lengths from those pairs:
lengths = [ p[1] for p in pairs ]
```

Practice problem: Work the same thing out via a “for” loop.

List comprehensions are often just the first step in further processing. For example, we could use a list comprehension to get the lengths of the words in the list, then use the sum function to add up those lengths:

```
lengths = [len(word) for word in words]
totalLength = sum(lengths)
```

Filters

One neat feature of list comprehensions is that they allow you to filter the results in a suitable way. So imagine we wanted to collect all numbers from 1 to 20 that are odd, and square them. We could write:

```
numbers = []
for n in range(1, 21):
    if n % 2 == 1:
        numbers.append(n * n)
```

Instead we could do this via a list comprehension:

```
numbers = [ n * n for n in range(1, 21) if n % 2 == 1 ]
# Or expanded:
numbers = [
    n * n
    for n in range(1, 21)
    if n % 2 == 1
]
```

Practice

1. Write a list comprehension that returns all numbers from 1 to 100 that produce a remainder of 3 when divided by 13.
2. Write a list comprehension that is given a list of words and appends at the end of each word a space followed by the length. So the word "art" will become "art 3".
3. A string can be used as the “list” in a list comprehension, and it will then be treated as a list of its characters. Use this idea to start from a string and return a list of the ASCII/UTF8 codes for the characters in the string.

Some more intricate examples

We will look at two more complicated examples. The first involves nested loops. Imagine for instance that we had a list of lists of numbers, and we wanted to flatten it all into a single list. One way to do this would be via a nested loop, the other would be via a list comprehension:

```
nums = [[1, 2, 3], [4, 5, 6]]
# Via nested loop
result = []
for row in nums:
    for n in row:
        result.append(n)
# Via list comprehension:
result = [
    n
    for row in nums
    for n in row
]
```

So each subsequent for−in generator will happen within each case of the previous generators. So for every row in the nums list and every number n in that row, we read that number.

Here is a more complicated example for finding all "Pythagorian triples" up to 100. A Pythagorian triple is three integers x, y, z such the squares of x and y add up to the squares of z. Here is how that might look via a list comprehension:

```
triples = [
    [x, y, z]
    for x in range(1, 101)
    for y in range(1, 101)
    for z in range(1, 101)
    if x*x + y*y == z*z
]
```

Dictionary comprehensions

A list comprehension can actually produce a dictionary instead. Here is an example where for each string in the list of words earlier we create a key in the dictionary, whose value is the length:

```
{
    word: len(word)
    for word in words
}
```

Practice

1. Start with a list of strings, and produce instead a single list containing all the individual characters.

2. You provided a JSON representation of students earlier. It looked in general like so:

   ```
   {
       "login": "...",
       "first": "...",
       "last": "...",
       "courses": [
           {
               "dept": "CS",
               "name": "...",
               ...
           },
           ...
       ]
   }
   ```

   Starting with a list of such objects as input, use a list comprehension to create a list that contains pairs (login, number) for each student and the number of courses the student has.

3. Instead of a list comprehension, now use a dictionary comprehension where the logins will be the dictionary keys and the number of courses will be the corresponding values.

4. With the same input as in problem 2, use a list comprehension to create a list of pairs (login, coursename) for each course a student is enrolled in.

5. Starting with two lists of numbers, use a list comprehension to produce all possible sums of numbers, one from each list.

6. Starting with a list of numbers, use a list comprehension to produce a list of strings, each string consisting of a number of asterisks equal to the corresponding number (multiplying a string with a number returns that string replicated that many times).