Python's List Comprehensions

Reading / References

- List comprehensions in Python's docs[1] for processing lists
- Some list comprehension examples[2]
- Visual explanation of list comprehensions[3]
- Even more examples of list comprehensions[4]

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 ]
```

Every list comprehension can be done via a for 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:

```
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.

[1]https://docs.python.org/2/tutorial/datastructures.html#tut-listcomps
[2]http://www.secnetix.de/olli/Python/list_comprehensions.hawk
[3]http://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/
[4]http://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html

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 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
]
```

Some more intricate examples

We will look at two more complicated examples. The one 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 ]
```

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
}
```

If tweets is the list of tweets we are working with, here is a complex comprehension with a list and objects in it:

```
[
    {
        "text": tweet['text'],
        "author": tweet['user']['screen_name']
    }
    for tweet in tweets
]
```

This results in a list of dictionaries, one for each tweet, containing the text and author information.