Comprehensions

One very nice feature in Python is a "comprehension". A comprehension lets you write down a list, set, or dictionary by specifying how to construct its value. Suppose that you wanted to write a list with all of the integers from 0 to 1000. Writing the literal list would be long and tedious. Instead, we can write a comprehension which specifies how to create the sequence:

```
myList = [i for i in range(0,1000)]
```

The syntax for this list comprehension looks a bit strange, especially as you have to somewhat read it backwards to understand what is going on. First, let us note that the way we can tell that we are using a comprehension is that we have the keyword for in the middle of what would otherwise look like a list literal:

```
myList = [i for i in range(0,1000)]
```

Next, let us look at the last part:

```
myList = [i for i in range(0,1000)]
```

This last part tells us the underlying sequence of data that we are going to use to generate the list elements. Here, we are going to bind the variable i to each element in range(0,1000). You are already familiar with range, and hopefully recall that range(0,1000) will produce the integers from 0 (inclusive) to 1000 (exclusive). You may recall that you can leave off the start whenever it is 0 (and just write range(1000)). However, I like to be explicit.

One we know what values i is going to take, we can look at how it gets used in the first part of the comprehension expression:

```
myList = [i for i in range(0,1000)]
```

This first part of the comprehension is an expression which specifies what value to put in the list. This expression is re-evaluated for each value that i takes on. Here, the expression is just i itself, so when i is 0, we put 0 in the list. When i is 1, we put 1 in the list, and so on. We can use more complex expressions if we want, and of course, could use other variable names than i. For example, we could write

```
otherList = [2 * j + 3 for j in range(4, 7)]
```

This comprehension works the same as before. Here, we have named the variable j, and it takes the values 4, 5, and 6 (remember that we exclude 7). For each of these values we compute 2 * j + 3 and put it in the resulting list. Accordingly we get 2 * 4 + 3 = 11, 2 * 5 + 3 = 13, and 2 * 6 + 3 = 15. So the result is that otherList has the value [11, 13, 15].

You may recall that we have previously discussed the idea of composability: that we programming constructs work the same no matter what other constructs we put them inside of. While we have been showing list comprehensions assigned to variables, they do not need to be. We could write them anywhere that we can write another list. We could even use list comprehensions inside of other list comprehensions!

Consider the following:

```
x = [[i + j for i in range (0,4)] for j in range (0,3)]
```

This comprehension produces a list of lists, namely x is `[[0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5]]`.

We can see this by thinking through exactly the same rules we discussed.

```
x = [[i + j for i in range (0,4)] for j in range (0,3)]
```

Here we need to evaluate the expression on the left for j being 0, 1, and 2. Composition says it does not matter how complicated the expression on the left is, we evaluate it for each value (0, 1, and 2) because those are the rules of a comprehension. What expression is that?

```
x = [[i + j for i in range (0,4)] for j in range (0,3)]
```

The expression is a list comprehension.And we evaluate that list comprehension the same as we always do.When j is 0, that evaluates like

```
[i + 0 for i in range (0,4)] = [0, 1, 2, 3]
```

When j is 1, that evaluates like

```
[i + 1 for i in range (0,4)] = [1, 2, 3, 4]
```

And finally when j is 2, that evaluates like

```
[i + 2 for i in range (0,4)] = [2, 3, 4, 5]
```

Those three lists ([0, 1, 2, 3], [1, 2, 3, 4], and [2, 3, 4, 5]) are the three items that make up the value of the whole big comprehension. You can also use comprehensions to create sets and dictionaries in much the same way. For example,

```
s = { i for i in range(0,100)}
```

Makes `s` be the set of integers from 0 (inclusive) to 100 (exclusive). You can tell that this comprehension makes a set instead of a list because it has curly braces ({}) like a set literal instead of square brackets ([]) like a list literal.

You can also make a dictionary comprehension, such as

```
d = { i : 4*i+ 7 for i in range(0,4) }
```

This dictionary comprehension would have the same effect as writing

```
d = {0: 7, 1: 11, 2: 15, 3: 19}
```

For the comprehension, we are again making `i` take each value in `range(0,4)`. For each of those values we evaluate the key value pairing `i : 4*i+7`. The expression before the colon is the key, and the one after the colon is the value.

We'll note that you often want something more complex for your key/value pairs—you typically do not want to just map ints to ints. One thing you might want to do is build your dictionary from some data you already have. For example, if you have a list of keys and a list of values (of the same length), such that keys[i] should have value values[i] you could do:

```
{ keys[i] : values[i] for i in range(0, len(keys) }
```

Another common way to make dictionary comprehensions is if you have a list (or set) of keys, and some function that computes the values you want. For example

```
{ k : f(k) : for k in somelist }
```

The last thing we will note about comprehensions is that they are syntactic sugar. Recall that syntactic sugar just means that they provide nicer ways to do things rather than ways to do new things. Anything you want to do, you could do without ever writing a comprehension (you could instead write a loop to add each element, for example). However, use of comprehensions not only makes your code easier to write and read, but is also considered "Pythonic". If you use comprehensions, it tells other Python programmers that you know Python-specific features. If you do not use them when they are a good choice, Python programmers looking at your code will wonder why you did not use them.