1.7 Building Up Data with Comprehensions

To wrap up our introduction to data in Python, we're going to learn about one last kind of expression that allows to build up and transform large collections of data in Python.

From set builder notation…

Recall *set builder notation*, which is a concise way of defining a mathematical set by specifying the values of the elements in terms of a larger domain. For example, suppose we have a set $S = \{1, 2, 3, 4, 5\}$. We can express a set of squares of the elements of S as follows:

$$\{x^2 \mid x \in S\}.$$

We can read the $\mid$ symbol as "where" and the $\in$ symbol as "in", so that the above expression reads as "the set of values x^2, where x is in the set S". More formally, this form of set builder notation has three parts: the *variable* x, the *domain* of that variable S, and the *build expression* x^2.[1]

[1] You might be familiar with other syntaxes for set builder notation from your studies in mathematics, but this is the form we'll use in this course because of how it translates into programming, as we'll see next.

…to set comprehensions!

It turns out that this form of set builder notation translates naturally to Python. To see how this works, let's go into the Python console and create a variable that refers to a set of numbers:

```
>>> numbers = {1, 2, 3, 4, 5}
```

Now, we introduce a new kind of Python expression called a **set comprehension**, which has the following syntax:[2]

[2] Careful with this: even though set comprehensions also use curly braces, they are *not* the same as set literals. In this case, we aren't writing the individual set elements separated by commas.

```
{ <expression> for <variable> in <collection> }
```

Evaluating a set comprehension is done by taking the `<expression>` and evaluating it once for each value in `<collection>` assigned to the `<variable>`. This is exactly analogous to set builder notation, except using `for` instead of $\mid$ and `in` instead of $\in$. Here's how we can repeat our initial example in Python using a set comprehension:

```
>>> {x ** 2 for x in numbers}
{1, 4, 9, 16, 25}
```

Pretty cool, eh? If you aren't sure exactly what happened here, it's useful to write out the expanded form of the set comprehension:

```
    {x ** 2 for x in numbers}
 == {1 ** 2, 2 ** 2, 3 ** 2, 4 ** 2, 5 ** 2}  # Replacing x with 1, 2, 3, 4, and 5.
```

It goes even further—we can use set comprehensions with a list collection instead of a set.

```
>>> {x ** 2 for x in [1, 2, 3, 4, 5]}
{1, 4, 9, 16, 25}
```

In fact, as we'll see later in this course, set comprehensions can be used with any "collection" data type in Python, not just sets and lists.

List and dictionary comprehensions

Even though set comprehensions draw their inspiration from set builder notation in mathematics, Python has extended them to other data types beyond `set`.

A **list comprehension** expression is very similar to a set comprehension, except its syntax uses square brackets instead of curly braces, and it produces a `list` instead of a `set`:

```
[ <expression> for <variable> in <collection> ]
```

Once again, `<collection>` can be a set or a list:

```
>>> [x + 4 for x in {10, 20, 30}]
[14, 24, 34]
>>> [x * 3 for x in [100, 200, 300]]
[300, 600, 900]
```

One word of warning: because sets are unordered but lists are ordered, you should *not* assume a particular ordering of the elements when a list comprehension generates elements from a set—the results can be unexpected!

```
>>> [x for x in {20, 10, 30}]
[10, 20, 30]
```

A **dictionary comprehension** expression is again similar to a set comprehension, but produces a `dict` by specifying both an expression to generate keys and an expression to generate their associated values:[3]

[3] Note that both dictionary and set comprehensions use outer curly braces. How do you tell them apart? A dictionary comprehension has a colon `:` on the left side of the `for`, which is used to separate two different expressions.

```
{ <key_expr>: <value_expr> for <variable> in <collection> }
```

Out of all three comprehension types, dictionary comprehensions are the most complex, because the left-hand side (before the `for`) consists of two expressions instead of one. Here is an example of a dictionary comprehension that creates a "table of values" for the function $f(x) = x^2 + 1$.

```
>>> {x : x ** 2 + 1 for x in {1, 2, 3, 4, 5}}
{1: 2, 2: 5, 3: 10, 4: 17, 5: 26}
```

And here is an example of a dictionary comprehension that creates keys of the form `'Mario is <word>'` and maps them to associated values of the form `'David is not <word>'`:

```
>>> words = {'cool', 'great', '✨'}
>>> {'Mario is ' + word : 'David is not ' + word for word in words}
{'Mario is cool': 'David is not cool', 'Mario is ✨': 'David is not ✨', 'Mario is great': 'Dav
```

Comprehensions over a range of numbers

In an above example, we wrote `for x in {1, 2, 3, 4, 5}` as part of a comprehension. You might have wondered, "Wow it's going to be pretty tedious to write these comprehensions if we have to write out each number in a range explicitly!" And indeed, in Python there is a much more concise way of expressing ranges of numbers.

In Python, given two integers `start` and `end`, we use the syntax[4] `range(start, end)` to produce a collection of the numbers in the sequence `start`, `start + 1`, `start + 2`, …, `end - 1`. For example, `range(1, 6)` represents the sequence of numbers `1, 2, 3, 4, 5`. Note that these ranges *include* the start number but *exclude* the end number.[5]

[4] Like `set()` from the previous chapter, `range(..., ...)` is a new kind of expression known as a *function call*, which we'll study in much more detail in the next chapter.

[5] In mathematics, you might have learned the terms *open* and *closed* intervals, which exclude and include their endpoints, respectively. In Python, we can say that `range`s are *half-open*.

So, here is another way we could have written our "table of values" expression:

```
>>> {x : x ** 2 + 1 for x in range(1, 6)}
{1: 2, 2: 5, 3: 10, 4: 17, 5: 26}
```

Of course, the power of `range` is that we can use it to express a very large sequence of numbers without writing them all down explicitly:

```
>>> {x : x ** 2 + 1 for x in range(1, 60)}
{1: 2, 2: 5, 3: 10, 4: 17, 5: 26, 6: 37, 7: 50, 8: 65, 9: 82, 10: 101, 11: 122, 12: 145, 13: 17
```

Combining collections with multiple variables

Our last example in this section will be to illustrate how we can combine two collections together using comprehension expressions with multiple variables.

First, let's introduce one new set operation. Let A and B be sets. We define the **Cartesian product** of A and B, denoted $A \times B$, to be the set consisting of all *pairs* (a, b) where a is an element of A and b is an element of B. We can use set builder notation to express this definition mathematically:

$$A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}.$$

This form of set builder notation is similar to what we saw at the start of this chapter, except now there are two variables and two domains. In this expression, the expression (x, y) is evaluated once for *every possible combination* of elements x from A and elements y from B. For example, if $A = \{1, 2, 3\}$ and $B = \{10, 20, 30\}$, then

$$A \times B = \{(1, 10), (1, 20), (1, 30), (2, 10), (2, 20), (2, 30), (3, 10), (3, 20), (3, 30)\}.$$

In Python, set, list, and dictionary comprehensions can have multiple variables as well, and the same principle of "all possible combinations" applies. We can specify additional variables in a comprehension by adding extra `for <variable> in <collection>` clauses to the comprehension. For example, if we define the following sets:

```
>>> nums1 = {1, 2, 3}
>>> nums2 = {10, 20, 30}
```

then we can calculate their Cartesian product using the following set comprehension:[6]

[6] Remember, order does not matter in sets! Don't worry about the order in the output.

```
>>> {(x, y) for x in nums1 for y in nums2}
{(3, 30), (2, 20), (2, 10), (1, 30), (3, 20), (1, 20), (3, 10), (1, 10), (2, 30)}
```

In general, if we have a comprehension with clauses `for v1 in collection1`, `for v2 in collection2`, etc., then the comprehension's build expression is evaluated *once for each combination of values for the variables*. This illustrates yet another pretty impressive power of Python: the ability to combine different collections of data together in a short amount of code.

Note: Python `tuple`s

One thing you might have noticed about this Cartesian product example is that in our Python comprehension we wrote `(x, y)`. Even though this notation is familiar from mathematics, we haven't formally introduced this type of syntax in Python.

It turns out that Python has two data types that can be used to represent sequences: `list`, whose literals are written using square brackets, and `tuple`, whose literals are written using parentheses. For example, `[3, 30]` is a `list` containing the elements `3` and `30`, and `(3, 30)` is a `tuple` containing the same elements.

Right now you can treat `list`s and `tuple`s as mostly equivalent, and we'll mainly use `list`s for the first few chapters of these notes. Later, we'll explore some key differences between these two data types.