Sequence Iteration in Python

Mapping is itself an instance of a general pattern of computation: iterating over all elements in a sequence. To map a function over a sequence, we do not just select a particular element, but each element in turn. This pattern is so common that Python has an additional control statement to process sequential data: the `for` statement.

Consider the problem of counting how many times a value appears in a sequence. We can implement a function to compute this count using a `while` loop.

```
>>> def count(s, value):
        """Count the number of occurrences of value in
sequence s."""
        total, index = 0, 0
        while index < len(s):
            if s[index] == value:
                total = total + 1
            index = index + 1
        return total
>>> count(digits, 8)
2
```

The Python `for` statement can simplify this function body by iterating over the element values directly, without introducing the name`index` at all. `For` example (pun intended), we can write:

```
>>> def count(s, value):
        """Count the number of occurrences of value in
sequence s."""
        total = 0
        for elem in s:
            if elem == value:
                total = total + 1
        return total
>>> count(digits, 8)
2
```

A `for` statement consists of a single clause with the form:

```
for <name> in <expression>:
    <suite>
```

A `for` statement is executed by the following procedure:

1. Evaluate the header `<expression>`, which must yield an iterable value.
2. For each element value in that sequence, in order:
   A. Bind `<name>` to that value in the local environment.
   B. Execute the `<suite>`.

Step 1 refers to an iterable value. Sequences are iterable, and their elements are considered in their sequential order. Python does include other iterable types, but we will focus on sequences for now; the general definition of the term "iterable" appears in the section on iterators in Chapter 4.

An important consequence of this evaluation procedure is that `<name>` will be bound to the last element of the sequence after the`for` statement is executed. The `for` loop introduces yet another way in which the local environment can be updated by a statement.

Sequence unpacking. A common pattern in programs is to have a sequence of elements that are themselves sequences, but all of a fixed length. `For` statements may include multiple names in their header to "unpack" each element sequence into its respective elements. For example, we may have a sequence of pairs (that is, two-element tuples),

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))
```

and wish to find the number of pairs that have the same first and second element.

```
>>> same_count = 0
```

The following `for` statement with two names in its header will bind each name `x` and `y` to the first and second elements in each pair, respectively.

```
>>> for x, y in pairs:
        if x == y:
            same_count = same_count + 1
>>> same_count
2
```

This pattern of binding multiple names to multiple values in a fixed-length sequence is called *sequence unpacking*; it is the same pattern that we see in assignment statements that bind multiple names to multiple values.

Ranges. A `range` is another built-in type of sequence in Python, which represents a range of integers. Ranges are created with the `range` function, which takes two integer arguments: the first number and one beyond the last number in the desired range.

```
>>> range(1, 10)  # Includes 1, but not 10
range(1, 10)
```

Calling the `tuple` constructor on a range will create a tuple with the same elements as the range, so that the elements can be easily inspected.

```
>>> tuple(range(5, 8))
(5, 6, 7)
```

If only one argument is given, it is interpreted as one beyond the last value for a range that starts at 0.

```
>>> tuple(range(4))
(0, 1, 2, 3)
```

Ranges commonly appear as the expression in a `for` header to specify the number of times that the suite should be executed:

```
>>> total = 0
>>> for k in range(5, 8):
        total = total + k
```

```
>>> total
18
```

A common convention is to use a single underscore character for the name in the `for` header if the name is unused in the suite:

```
>>> for _ in range(3):
        print('Go Bears!')

Go Bears!
Go Bears!
Go Bears!
```

Note that an underscore is just another name in the environment as far as the interpreter is concerned, but has a conventional meaning among programmers that indicates the name will not appear in any expressions.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#sequence-iteration