Sequence Abstraction and Nested Pairs in Python

Sequence Abstraction

We have now introduced two types of native data types that implement the sequence abstraction: tuples and ranges. Both satisfy the conditions with which we began this section: length and element selection. Python includes two more behaviors of sequence types that extend the sequence abstraction.

Membership. A value can be tested for membership in a sequence. Python has two operators `in` and `not in` that evaluate to `True` or `False` depending on whether an element appears in a sequence.

```
>>> digits
(1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

All sequences also have methods called `index` and `count`, which return the index of (or count of) a value in a sequence.

Slicing. Sequences contain smaller sequences within them. We observed this property when developing our nested pairs implementation, which decomposed a sequence into its first element and the rest. A *slice* of a sequence is any span of the original sequence, designated by a pair of integers. As with the `range` constructor, the first integer indicates the starting index of the slice and the second indicates one beyond the ending index.

In Python, sequence slicing is expressed similarly to element selection, using square brackets. A colon separates the starting and ending indices. Any bound that is omitted

is assumed to be an extreme value: 0 for the starting index, and the length of the sequence for the ending index.

```
>>> digits[0:2]
(1, 8)
>>> digits[1:]
(8, 2, 8)
```

Enumerating these additional behaviors of the Python sequence abstraction gives us an opportunity to reflect upon what constitutes a useful data abstraction in general. The richness of an abstraction (that is, how many behaviors it includes) has consequences. For users of an abstraction, additional behaviors can be helpful. On the other hand, satisfying the requirements of a rich abstraction with a new data type can be challenging. To ensure that our implementation of recursive lists supported these additional behaviors would require some work. Another negative consequence of rich abstractions is that they take longer for users to learn.

Sequences have a rich abstraction because they are so ubiquitous in computing that learning a few complex behaviors is justified. In general, most user-defined abstractions should be kept as simple as possible.

Nested Pairs

For rational numbers, we paired together two integer objects using a two-element tuple, then showed that we could implement pairs just as well using functions. In that case, the elements of each pair we constructed were integers. However, like expressions, tuples can nest. Either element of a pair can itself be a pair, a property that holds true for either method of implementing a pair that we have seen: as a tuple or as a dispatch function.

We visualize pairs (two-element tuples) in environment diagrams using *box-and-pointer* notation. Pairs are depicted as boxes with two parts: the left part contains (an arrow to) the first element of the pair and the right part contains the second. Simple values such as numbers, strings, boolean values, and `None` appear within the box. Composite values, such as function values and other pairs, are connected by a pointer.

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

We can use recursion to process an arbitrary nesting of pairs. For example, let's write a function to compute the sum of all integer elements in a nesting of pairs and integers.

```
def sum_elems(elem):
    if type(elem) == int:
        return elem
    else:
        return (sum_elems(elem[0]) +
                sum_elems(elem[1]))

pairs = ((1, 2), (3, 4))
total = sum_elems(pairs)
```

The `sum_elems` function computes the sum of integer elements in a nested pair by recursively computing the sums of its first and second elements and adding the results. The base case is when an element is an integer, in which case the sum is the integer itself.

Our ability to use tuples as the elements of other tuples provides a new means of combination in our programming language. We call the ability for tuples to nest in this way a *closure property* of the tuple data type. In general, a method for combining data values satisfies the closure property if the result of combination can itself be combined using the same method. Closure is the key to power in any means of combination because it permits us to create hierarchical structures --- structures made up of parts, which themselves are made up of parts, and so on.

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