Sequences in Python

A sequence is an ordered collection of data values. Unlike a pair, which has exactly two elements, a sequence can have an arbitrary (but finite) number of ordered elements.

The sequence is a powerful, fundamental abstraction in computer science. For example, if we have sequences, we can list every university in the world, or every student in every university. The sequence abstraction enables the thousands of data-driven programs that impact our lives every day.

A sequence is not a particular abstract data type, but instead a collection of behaviors that different types share. That is, there are many kinds of sequences, but they all share certain properties. In particular,

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Unlike an abstract data type, we have not stated how to construct a sequence. The sequence abstraction is a collection of behaviors that does not fully specify a type (i.e., with constructors and selectors), but may be shared among several types. Sequences provide a layer of abstraction that may hide the details of exactly which sequence type is being manipulated by a particular program.

In this section, we introduce built-in Python types that implement the sequence abstraction. We then develop our own abstract data type that can implement the same abstraction.

Tuples

In fact, the `tuple` type that we introduced to form primitive pairs is itself a full sequence type. Tuples provide substantially more functionality than the pair abstract data type that we implemented functionally.

Tuples can have arbitrary length, and they exhibit the two principal behaviors of the sequence abstraction: length and element selection. Below, `digits` is a tuple with four elements.

```
>>> digits = (1, 8, 2, 8)
>>> len(digits)
4
>>> digits[3]
8
```

Additionally, tuples can be added together and multiplied by integers. For tuples, addition and multiplication do not add or multiply elements, but instead combine and replicate the tuples themselves. That is, the `add` function in the `operator` module (and the `+`operator) returns a new tuple that is the conjunction of the added arguments. The `mul` function in `operator` (and the `*` operator) can take an integer `k` and a tuple and return a new tuple that consists of `k` copies of the tuple argument.

```
>>> (2, 7) + digits * 2
(2, 7, 1, 8, 2, 8, 1, 8, 2, 8)
```

Mapping. A powerful method of transforming one tuple into another is by applying a function to each element and collecting the results. This general form of computation is called *mapping* a function over a sequence, and corresponds to the built-in function`map`. The result of `map` is an object that is not itself a sequence, but can be converted into a sequence by calling `tuple`, the constructor function for tuples.

```
>>> alternates = (-1, 2, -3, 4, -5)
>>> tuple(map(abs, alternates))
(1, 2, 3, 4, 5)
```

The `map` function is important because it relies on the sequence abstraction: we do not need to be concerned about the structure of the underlying tuple; only that we can access each one of its elements individually in order to pass it as an argument to the mapped function (`abs`, in this case).

Multiple assignment and return values. In Chapter 1, we saw that Python allows multiple names to be assigned in a single statement.

```
>>> from math import pi
>>> radius = 10
>>> area, circumference = pi * radius * radius, 2 * pi * radius
>>> area
314.1592653589793
>>> circumference
62.83185307179586
```

We can also return multiple values from a function.

```
>>> def divide_exact(n, d):
        return n // d, n % d
>>> quotient, remainder = divide_exact(10, 3)
>>> quotient
3
>>> remainder
1
```

Python actually uses tuples to represent multiple values separated by commas. This is called *tuple packing*.

```
>>> digits = 1, 8, 2, 8
>>> digits
(1, 8, 2, 8)
>>> divide_exact(10, 3)
(3, 1)
```

Using a tuple to assign to multiple names is called, as one might expect, *tuple unpacking*. The names may or may not be enclosed by parentheses.

```
>>> d0, d1, d2, d3 = digits
>>> d2
2
>>> (quotient, remainder) = divide_exact(10, 3)
>>> quotient
3
>>> remainder
1
```

Multiple assignment is just the combination of tuple packing and unpacking.

Arbitrary argument lists. Tuples can be used to define a function that takes in an arbitrary number of arguments, such as the built-in `print` function. We precede a parameter name with a `*` to indicate that an arbitrary number of arguments can be passed in for that parameter. Python automatically packs those arguments into a tuple and binds the parameter name to that tuple..

```
>>> def add_all(*args):
        """Compute the sum of all arguments."""
        total, index = 0, 0
        while index < len(args):
            total = total + args[index]
            index = index + 1
        return total
>>> add_all(1, 3, 2)
6
```

In addition, we can use the `*` operator to unpack a tuple to pass its elements as separate arguments to a function call.

```
>>> pow(*(2, 3))
8
```

As can be seen here, tuples are used to provide many of the features that we have been using in Python.

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