Conventional Interfaces in Python

In working with compound data, we've stressed how data abstraction permits us to design programs without becoming enmeshed in the details of data representations, and how abstraction preserves for us the flexibility to experiment with alternative representations. In this section, we introduce another powerful design principle for working with data structures --- the use of*conventional interfaces*.

A conventional interface is a data format that is shared across many modular components, which can be mixed and matched to perform data processing. For example, if we have several functions that all take a sequence as an argument and return a sequence as a value, then we can apply each to the output of the next in any order we choose. In this way, we can create a complex process by chaining together a pipeline of functions, each of which is simple and focused.

This section has a dual purpose: to introduce the idea of organizing a program around a conventional interface, and to demonstrate examples of modular sequence processing.

Consider these two problems, which appear at first to be related only in their use of sequences:

1. Sum the even members of the first `n` Fibonacci numbers.
2. List the letters in the acronym for a name, which includes the first letter of each capitalized word.

These problems are related because they can be decomposed into simple operations that take sequences as input and yield sequences as output. Moreover, those operations are instances of general methods of computation over sequences. Let's consider the first problem. It can be decomposed into the following steps:

```
 enumerate     map    filter  accumulate
-----------    ---    ------  ----------
naturals(n)    fib    iseven     sum
```

The `fib` function below computes Fibonacci numbers (now updated from the definition in Chapter 1 with a `for` statement),

```
>>> def fib(k):
        """Compute the kth Fibonacci number."""
        prev, curr = 1, 0  # curr is the first Fibonacci
number.
        for _ in range(k - 1):
            prev, curr = curr, prev + curr
        return curr
```

and a predicate `iseven` can be defined using the integer remainder operator, `%`.

```
>>> def iseven(n):
        return n % 2 == 0
```

The functions `map` and `filter` are operations on sequences. We have already encountered `map`, which applies a function to each element in a sequence and collects the results. The `filter` function takes a sequence and returns those elements of a sequence for which a predicate is true. Both of these functions return intermediate objects, `map` and `filter` objects, which are iterable objects that can be converted into tuples or summed.

```
>>> nums = (5, 6, -7, -8, 9)
>>> tuple(filter(iseven, nums))
(6, -8)
>>> sum(map(abs, nums))
35
```

Now we can implement `even_fib`, the solution to our first problem, in terms of `map`, `filter`, and `sum`.

```
>>> def sum_even_fibs(n):
        """Sum the even members of the first n Fibonacci
numbers."""
        return   sum(filter(iseven,   map(fib,   range(1,
n+1))))
>>> sum_even_fibs(20)
3382
```

Now, let's consider the second problem. It can also be decomposed as a pipeline of sequence operations that include `map` and`filter`:

```
enumerate  filter   map   accumulate
---------  ------  -----  ----------
  words    iscap   first    tuple
```

The words in a string can be enumerated via the `split` method of a string object, which by default splits on spaces.

```
>>> tuple('Spaces between words'.split())
('Spaces', 'between', 'words')
```

The first letter of a word can be retrieved using the selection operator, and a predicate that determines if a word is capitalized can be defined using the built-in predicate `isupper`.

```
>>> def first(s):
        return s[0]
>>> def iscap(s):
        return len(s) > 0 and s[0].isupper()
```

At this point, our acronym function can be defined via `map` and `filter`.

```
>>> def acronym(name):
        """Return a tuple of the letters that form the
acronym for name."""
        return       tuple(map(first,       filter(iscap,
name.split())))
>>>    acronym('University    of    California    Berkeley
Undergraduate Graphics Group')
('U', 'C', 'B', 'U', 'G', 'G')
```

These similar solutions to rather different problems show how to combine general components that operate on the conventional interface of a sequence using the general computational patterns of mapping, filtering, and accumulation. The sequence abstraction allows us to specify these solutions concisely.

Expressing programs as sequence operations helps us design programs that are modular. That is, our designs are constructed by combining relatively independent pieces, each of which transforms a sequence. In general, we can encourage modular design by providing a library of standard components together with a conventional interface for connecting the components in flexible ways.

Generator expressions. The Python language includes a second approach to processing sequences, called *generator expressions*. which provide similar functionality to `map` and `filter`, but may require fewer function definitions.

Generator expressions combine the ideas of filtering and mapping together into a single expression type with the following form:

```
<map expression> for <name> in <sequence expression> if
<filter expression>
```

To evaluate a generator expression, Python evaluates the `<sequence expression>`, which must return an iterable value. Then, for each element in order, the element value is bound to `<name>`, the filter expression is evaluated, and if it yields a true value, the map expression is evaluated.

The result value of evaluating a generator expression is itself an iterable value. Accumulation functions like `tuple`, `sum`, `max`, and`min` can take this returned object as an argument.

```
>>> def acronym(name):
        return tuple(w[0] for w in name.split() if
iscap(w))
>>> def sum_even_fibs(n):
        return sum(fib(k) for k in range(1, n+1) if
fib(k) % 2 == 0)
```

Generator expressions are specialized syntax that utilizes the conventional interface of iterable values, such as sequences. These expressions subsume most of the functionality of `map` and `filter`, but avoid actually creating the function values that are

applied (or, incidentally, creating the environment frames required to apply those functions).

Reduce. In our examples we used specific functions to accumulate results, either `tuple` or `sum`. Functional programming languages (including Python) include general higher-order accumulators that go by various names. Python includes `reduce` in the`functools` module, which applies a two-argument function cumulatively to the elements of a sequence from left to right, to reduce a sequence to a value. The following expression computes 5 factorial.

```
>>> from operator import mul
>>> from functools import reduce
>>> reduce(mul, (1, 2, 3, 4, 5))
120
```

Using this more general form of accumulation, we can also compute the product of even Fibonacci numbers, in addition to the sum, using sequences as a conventional interface.

```
>>> def product_even_fibs(n):
        """Return the product of the first n even
Fibonacci numbers, except 0."""
        return   reduce(mul,   filter(iseven,   map(fib,
range(2, n+1))))
>>> product_even_fibs(20)
123476336640
```

The combination of higher order procedures corresponding to `map`, `filter`, and `reduce` will appear again in Chapter 4, when we consider methods for distributing computation across multiple computers.

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