map, filter, and reduce

Python provides several functions which enable a **functional approach** to programming. These functions are all convenience features in that they can be written in Python fairly easily.

Functional programming is all about **expressions**. We may say that the Functional programming is an **expression oriented programming**.

Expression oriented functions of Python provides are:

- map(aFunction, aSequence)
- filter(aFunction, aSequence)
- reduce(aFunction, aSequence)
- lambda
- list comprehension

map

One of the common things we do with list and other sequences is applying an operation to each item and collect the result. For example, updating all the items in a list can be done easily with a **for** loop:

```
>>> items = [1, 2, 3, 4, 5]
>>> squared = []
>>> for x in items:
        squared.append(x ** 2)


>>> squared
[1, 4, 9, 16, 25]
>>>
```

Since this is such a common operation, actually, we have a built-in feature that does most of the work for us. The **map(aFunction, aSequence)** function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results.

```
>>> items = [1, 2, 3, 4, 5]
>>>
>>> def sqr(x): return x ** 2

>>> list(map(sqr, items))
```

```
[1, 4, 9, 16, 25]
>>>
```

We passed in a user-defined function applied to each item in the list. **map** calls **sqr** on each list item and collects all the return values into a new list.

Because **map** expects a function to be passed in, it also happens to be one of the places where **lambda** routinely appears:

```
>>> list(map((lambda x: x **2), items))
[1, 4, 9, 16, 25]
>>>
```

In the short example above, the **lambda** function squares each item in the items list.

As shown earlier, map is defined like this:

```
map(aFunction, aSequence)
```

While we still use **lamda** as a **aFunction**, we can have a **list of functions** as **aSequence**:

```
def square(x):
        return (x**2)
def cube(x):
        return (x**3)

funcs = [square, cube]
for r in range(5):
    value = map(lambda x: x(r), funcs)
    print value
```

Output:

```
[0, 0]
[1, 1]
[4, 8]
[9, 27]
[16, 64]
```

Because using **map** is equivalent to **for** loops, with an extra code we can always write a general mapping utility:

```
>>> def mymap(aFunc, aSeq):
```

```
        result = []
        for x in aSeq: result.append(aFunc(x))
        return result

>>> list(map(sqr, [1, 2, 3]))
[1, 4, 9]
>>> mymap(sqr, [1, 2, 3])
[1, 4, 9]
>>>
```

Since it's a built-in, **map** is always available and always works the same way. It also has some performance benefit because it is usually faster than a manually coded **for** loop. On top of those, **map** can be used in more advance way. For example, given multiple sequence arguments, it sends items taken form sequences in parallel as distinct arguments to the function:

```
>>> pow(3,5)
243
>>> pow(2,10)
1024
>>> pow(3,11)
177147
>>> pow(4,12)
16777216
>>>
>>> list(map(pow,[2, 3, 4], [10, 11, 12]))
[1024, 177147, 16777216]
>>>
```

As in the example above, with multiple sequences, **map()** expects an N-argument function for N sequences. In the example, **pow** function takes two arguments on each call.

The **map** call is similar to the **list comprehension expression**. But **map** applies a **function call** to each item instead of an **arbitrary expression**. Because of this limitation, it is somewhat less general tool. In some cases, however, **map** may be faster to run than a list comprehension such as when mapping a built-in function. And **map** requires less coding.

If **function** is **None**, the **identity** function is assumed; if there are multiple arguments, **map()** returns a list consisting of **tuples** containing the corresponding items from all iterables (a kind of transpose operation). The iterable arguments may be a sequence or any iterable object; the result is always a list:

```
>>> m = [1,2,3]
>>> n = [1,4,9]
>>> new_tuple = map(None, m, n)
>>> new_tuple
[(1, 1), (2, 4), (3, 9)]
```

filter and reduce

As the name suggests **filter** extracts each element in the sequence for which the function returns **True**. The **reduce** function is a little less obvious in its intent. This function reduces a list to a single value by combining elements via a supplied function. The **map** function is the simplest one among Python built-ins used for **functional programming**.

These tools apply functions to sequences and other iterables. The **filter** filters out items based on a test function which is a **filter** and apply functions to pairs of item and running result which is **reduce**.

Because they return iterables, **range** and **filter** both require **list** calls to display all their results in Python 3.0.

As an example, the following **filter** call picks out items in a sequence that are less than zero:

```
>>> list(range(-5,5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>>
>>> list( filter((lambda x: x < 0), range(-5,5)))
[-5, -4, -3, -2, -1]
>>>
```

Items in the sequence or iterable for which the function returns a true, the result are added to the result list. Like **map**, this function is roughly equivalent to a **for** loop, but it is built-in and fast:

```
>>>
>>> result = []
>>> for x in range(-5, 5):
        if x < 0:
                result.append(x)


>>> result
[-5, -4, -3, -2, -1]
>>>
```

The **reduce** is in the **functools** in Python 3.0. It is more complex. It accepts an iterator to process, but it's not an iterator itself. It returns a single result:

```
>>>
>>> from functools import reduce
>>> reduce( (lambda x, y: x * y), [1, 2, 3, 4] )
24
>>> reduce( (lambda x, y: x / y), [1, 2, 3, 4] )
0.041666666666666664
>>>
```

At each step, **reduce** passes the current product or division, along with the next item from the list, to the passed-in **lambda** function. By default, the first item in the sequence initialized the starting value.

Here's the **for** loop version of the first of these calls, with the multiplication hardcoded inside the loop:

```
>>> L = [1, 2, 3, 4]
>>> result = L[0]
>>> for x in L[1:]:
        result = result * x


>>> result
24
>>>
```

Let's make our own version of **reduce**.

```
>>> def myreduce(fnc, seq):
        tally = seq[0]
        for next in seq[1:]:
                tally = fnc(tally, next)
        return tally

>>> myreduce( (lambda x, y: x * y), [1, 2, 3, 4])
24
>>> myreduce( (lambda x, y: x / y), [1, 2, 3, 4])
0.041666666666666664
>>>
```

We can concatenate a list of strings to make a sentence. Using the [Dijkstra's famous quote] on bug:

```
import functools
>>> L = ['Testing ', 'shows ', 'the ', 'presence', ', ','not ', 'the ',
'absence ', 'of ', 'bugs']
>>> functools.reduce( (lambda x,y:x+y), L)
'Testing shows the presence, not the absence of bugs'
>>>
```

We can get the same result by using **join** :

```
>>> ''.join(L)
'Testing shows the presence, not the absence of bugs'
```

We can also use **operator** to produce the same result:

```
>>> import functools, operator
>>> functools.reduce(operator.add, L)
'Testing shows the presence, not the absence of bugs'
```

>>>

The built-in **reduce** also allows an optional third argument placed before the items in the sequence to serve as a default result when the sequence is empty.