Functional Programming: map, filter, reduce

"Functional programming" is a useful programming style, that stands in contrast to "imperative" programming and "object oriented" programming in important ways. The key idea is to decompose computation into functions that simply produce output values in response to input values, without local state and mutation of state. The composition of functions such as `f(g(x, h(y, z))))` matters, but the order of separate function calls does not matter. For example, `f(3, 55)` will always return the same value in a purely functional program, since there is no change in state under the covers (i.e., in global or local variables, or instance/class variables in OOP) that might affect that functional mapping.

Note that python is multilingual, in the sense that it supports all of (or a combination of) functional, imperative, and object oriented programming styles. Indeed, the idea of "comprehensions", and the generator or iterator mechanism, can be viewed as based on functional concepts -- though these are not strictly functional, in that mutation is possible within such constructs.

A popular or classic group of functions typical in functional programming are *map*, *filter*, and *reduce*. Python has `map` and `filter` as built-ins; as of Python 3, `reduce` as been moved to `functools.reduce`. Here we'll consider our own implementations of these functions, in both python and in Scheme (Carlae). In 6.009 Lab8, you'll implement built-in versions of these to include your Carlae language.

You can learn more about functional programming in python at https://docs.python.org/3/howto/functional.html (https://docs.python.org/3/howto/functional.html)

1. map

Python -- our own `map_list`

In [ ]:
```
def map_list(f, L):
    return [f(x) for x in L]

map_list(lambda z: z*z, [1, 2, 3])
```

In [ ]:
```
def sq(z):
    return z*z
map_list(sq, range(10))
```

The `map_list` above is kind of cheating; it uses list comprehension to do the mapping. What if we didn't have list comprehensions at our disposal? We could write other iterative or recursive versions. Below is a recursive implementation, which turns out to be close to how we would write the function in Scheme:

In [ ]:
```
def map_list(f, L):
    if len(L) == 0:
        return []
    return [f(L[0])] + map_list(f, L[1:])

map_list(lambda z: z*z, range(10))
```

Scheme versions of `map_list`

In [ ]:
```
;; Scheme/Carlae
(define (map_list f L)
    (if (=? (length L) 0)
        (list)
        (concat (list (f (car L))) (map_list f (cdr L)))))

(define (sq x) (* x x))
(map_list sq (list 1 2 3 4 5 6))
; out> (1 4 9 16 25 36)
```

In [ ]:
```
;; Scheme/Carlae; a bit more "Scheme-ish"
(define (map_list f L)
    (if (=? (length L) 0)
        (list)
        (cons (f (car L)) (map_list f (cdr L)))))
```

Python -- our own map_list, but now as a generator

Python generators empower the functional programming style very nicely, specifically by deferring computations until the value is actually needed. This is sometimes referred to as "lazy evaluation", and is a facility provided by some languages (e.g., with `delay` and `force` in Scheme).

In [ ]:
```
def map_iter(f, L):
    return (f(x) for x in L)

map_iter(lambda z: z*z, range(10))
```

In [ ]:
```
list(map_iter(lambda z: z*z, range(10)))
```

In [ ]:
```
def map_iter(f, L):
    for item in L:
        yield f(item)

map_iter(lambda z: z*z, range(10))
```

In [ ]:
```
list(map_iter(lambda z: z*z, range(10)))
```

Python's built-in map is an iterator

In [ ]:
```
map(lambda z: z*z, range(10))
```

In [ ]:
```
list(map(lambda z: z*z, range(10)))
```

Note that python's built-in `map` is more flexible than our `map_iter`, in that it can take multiple arguments corresponding to multiple-argument functions:

In [ ]:
```
def foo(x, y, z):
    return (x, y, z)

list(map(foo, range(5), range(50, 55), ['a', 'b', 'c', 'd', 'e']))
list(map(foo, range(5), range(50, 55), ['a', 'b']))
```

Carlae's built-in map creates output lists

That is to say, in our Carlae language, we do not implement the delayed or lazy evaluation model. Rather, our Carlae `map`, `filter`, and `reduce` will operate on `lists`. You will implement `map` and these other functions as built-ins for Carlae in lab8.

2. Filter

A second classic function in the "map, filter, reduce" paradigm is `filter`.

Python version: `filter_list`

In [ ]:
```
def filter_list(pred, L):
    if len(L) == 0:
        return []
    elif pred(L[0]):
        return [L[0]] + filter_list(pred, L[1:])
    else:
        return filter_list(pred, L[1:])

def not_seven(z):
    return not z == 7

filter_list(not_seven, [1, 2, 7, 2, 7, 14, 7])
#filter_list(not_seven, [])
```

Python version: `filter_iter`

In [ ]:
```
def filter_iter(pred, L):
    for item in L:
        if pred(item):
            yield item

def not_seven(z):
    return not z == 7

filter_iter(not_seven, [1, 2, 7, 2, 7, 14, 7])
list(filter_iter(not_seven, [1, 2, 7, 2, 7, 14, 7]))
#list(filter_iter(not_seven, []))
```

Scheme version: `filter_list`

In [ ]:
```
;; Scheme/Carlae
(define (filter_list pred L)
    (if (=? (length L) 0)
        (list)
        (if (pred (car L))
            (cons (car L) (filter_list pred (cdr L)))
            (filter_list pred (cdr L)))))

(define (not_seven x) (not (=? x 7)))
(filter_list not_seven (list 1 7 3 4 7))
; out> (1 3 4)
```

3. Reduce

Finally, the last function we'll look at in the "map, filter, reduce" paradigm is `reduce`.

Scheme version: `reduce_list`

```
In [ ]:
```

```scheme
;; Scheme/Carlae
(define (reduce_list f L init)
    (if (=? (length L) 0)
        init
        (reduce_list f (cdr L) (f init (car L)))))

(reduce_list + (list 1 2 3) 0)
; out> 6
```

Python reduce -- in functools

```
In [ ]:
```

```python
from functools import reduce

def add(x, y):
    return x + y
reduce(add, [1, 2, 3], 0)
```

Example: sum of the even squares from 1 to 1,000,000:

```
In [ ]:
```

```python
reduce(add, filter(lambda x: x%2==0, map(sq, range(1, 1000001))), 0)
```

```
In [ ]:
```

```python
reduce(add, filter_list(lambda x: x%2==0, map_list(sq, range(1, 1000001))), 0)
```

So one of the problems with our python recursive implementations of filter_list and map_list is that the recursion depth can grow very large, and may exceed the maximum recursion depth. This is one of the reasons for implementing these as generators and iterators instead:

```
In [ ]:
```

```python
reduce(add, filter_iter(lambda x: x%2==0, map_iter(sq, range(1, 1000001))), 0)
```

Example: *all* even squares:

```
In [ ]:
```

```python
def all_ints(start):
    while True:
        yield start
        start += 1

def even(x): return x%2 == 0

all_even_squares = filter_iter(even, map_iter(sq, all_ints(0)))
print("all even squares:", all_even_squares)
print("first 10 even squares:", [next(all_even_squares) for i in range(10)])
```