CIS 343 - Structure of Programming Languages

Nathan Bowman

Based on slides by Ira Woodring

Functional Programming Tools

There are several useful routines that are built in to many functional languages in some form

- map
- fold (reduce)
- filter
- zip

Like lambda functions, these are generally applicable enough that they have made their way into some non-functional languages

`map` is a generalization of a pattern that we often see in imperative programming

```
for i in range(len(vals)):
    vals2.append(vals[i]*2)

for i in range(len(vals)):
    vals_sq.append(vals[i]**2)
```

We often loop through an array and apply the same function to every element

The idea of higher-order functions allows us to generalize this

```
def times_two(x):
    return x*2

def square(x):
    return x**2

def updated_array(vals, fun):
    new_vals = []
    for i in range(len(vals)):
        new_vals.append(fun(vals[i]))
    return new_vals
```

```
vals2 = updated_array(vals, times_two)
vals_sq = updated_array(vals, square)
```

We do not have looping in functional languages, but we can accomplish the same idea

This looks a lot nicer when we use lambda functions and `map`

```
(map (lambda (x) (* 2 x)) vals)
(map (lambda (x) (expt x 2)) vals)
```

`map` fulfills the job of `updated_array` in previous example and is built directly into Scheme

```
(define vals '(1 2 3))

(map (lambda (x) (* 2 x)) vals)
;Value: (2 4 6)

(map (lambda (x) (expt x 2)) vals)
;Value: (1 4 9)
```

Once you understand this, you can use same idea in Python

```
map(lambda x : 2*x, vals)
map(lambda x : x**2, vals)
```

For efficiency, Python will not compute the values until they are needed. You can confirm that `map` is working by making a list

```
list(map(lambda x : 2*x, vals))
```

```
map(lambda x : 2*x, vals)
# <map at 0x7f35490828b0>

list(map(lambda x : 2*x, vals))
# [2, 4, 6]
```

`map` applies same function to every element of a list

Applied function can be anonymous (lambda) or named

Does not alter list in place -- there is no such thing in functional programming

`fold` is another functional routine that replaces typical imperative pattern

- map
- **fold** (reduce)
- filter
- zip

Consider the following

```
total = 0
for i in range(len(vals)):
    total += vals[i]

product = 1
for i in range(len(vals)):
    product *= vals[i]
```

Once again, we capture this pattern with higher-order functions

```
def add(x, y):
    return x + y

def multiply(x, y):
    return x*y

def value_from_array(vals, fun, initial):
    for i in range(len(vals)):
        initial = fun(initial, vals[i])
    return initial
```

```
total = value_from_array(vals, add, 0)
product = value_from_array(vals, multiply, 1)
```

```
def value_from_array(vals, fun, initial)
```

Note the third argument to `value_from_array`

`fun` takes two arguments, so we need some value to begin with that makes sense to start with

When taking a sum, 0 makes sense, but, when taking a product, 1 makes sense

In Scheme, this pattern is simply

```
(fold + 0 vals)
;Value: 6

(fold * 1 vals)
;Value: 6
```

As with `map`, does not need to use built-in functions

```
(fold (lambda (x y) (+ x y))  0 vals)
;Value: 6
```

There is more than one type of folding depending on whether left-associative or right-associative action is required. Consider

((1 + 2) + 3) vs

(1 + (2 + 3))

`fold` by default assumes left-associativity (first example)

In case of addition, these produce same result either way

Consider exponentiation, which is normally right-associative

With list (2 4 6), sensible exponential is

(2 ** (4 ** 6))

Left fold vs right fold with initial value of 1 gives

(((1 ** 2) ** 4) ** 6) vs

(2 ** (4 ** (6 ** 1)))

```
(define a '(2 4 6))

(fold-left (lambda (x y) (expt x y)) 1 a)
;Value: 1

(fold-right (lambda (x y) (expt x y)) 1 a)

;Value: 1044388881413152506691752710716624382579964249047383780384233483283953907971557456848826811934997558340890106714439262837987573438185793607
...
```

In many places, you will see `fold` and `reduce` used interchangeably

Scheme has both `fold` and `reduce` that do almost, but not exactly, the same thing

Difference in Scheme has to do with how initial value is handled

Python also has (left-associative) `reduce` equivalent to Scheme's `fold`

Was built-in in Python 2, but moved to `functools` in Python 3

```
from functools import reduce

reduce(lambda x, y: x + y, vals, 0)
```

Order of arguments to `reduce` is switched -- initial value comes last in Python (and is technically optional)

Once again, lambda functions are convenient, but not necessary

Using add defined earlier, could do

```
reduce(add, vals, 0)
```

We can create new functions from these higher-order functions

For example, a function that

- accepts a list of lists
- produces a list containing the sums of the sublists

```
(define in-list '( (1 2 3) (4 5 6) (7 8 9) ) )

; want to output ( 6 15 24 )
```

Function to sum sublist is

```
(lambda (sublist) (fold + 0 sublist))
```

Apply to every element of `[lists_to_sum]` with `map`:

```
(map [sum_function_here] [lists_to_sum])
(map (lambda (sublist) (fold + 0 sublist)) [lists_to_sum])
```

Create a *new function* that sums lists of lists using `define`

```
(define (sum-sublists in-list)
  (map (lambda (sublist) (fold + 0 sublist)) in-list))
```

```
(define in-list '( (1 2 3) (4 5 6) (7 8 9) ) )
(define (sum-sublists in-list)
  (map (lambda (sublist) (fold + 0 sublist)) in-list))

(sum-sublists in-list)
;Value: (6 15 24)
```

We use these general tools not just to do specific work, but to create new, reusable functions

Next useful functional tool we investigate is `filter`

- map
- fold (reduce)
- **filter**
- zip

Consider creating a sublist based on some condition

```
evens = []
for v in vals:
    if v % 2 == 0:
        evens.append(v)
```

Or, assuming we had written an `is_prime` function

```
primes = []
for v in vals:
    if is_prime(v):
        primes.append(v)
```

The higher-order function way of writing this is

```
def is_even(x):
    return (x % 2 == 0)

def is_prime(x):
    # ...
    return was_prime_number  # True or False

def sub_array(vals, predicate):
    new_vals = []
    for v in vals:
        if predicate(v):
            new_vals.append(v)
    return new_vals
```

```
vals_even = sub_array(vals, is_even)
vals_prime = sub_array(vals, is_prime)
```

Notice that we called our function `predicate` this time

There is nothing special about this -- a predicate is simply a function that returns only true or false

In Scheme, this would be

```
(filter (lambda (x) (= (modulo x 2) 0)) vals)
;Value: (2)
(filter is-prime? vals)
;Value: (2 3)
```

This assumes we have written `is-prime?` elsewhere

To make first example a little more clear, we could define our own `is-even?` function.

```
(define (is-even? x) (= (modulo x 2) 0))
(filter (lambda (x) (= (modulo x 2) 0)) vals)  ; before
;Value: (2)
(filter is-even? vals)  ; after
;Value: (2)
```

However, `even?` is already built into Scheme

```
(filter even? vals)
;Value: (2)
```

In Python, using `is_even` from earlier, we have

```
filter(is_even, vals)
# <filter at 0x7f35490a22e0>

list(filter(is_even, vals))
# [2]
```

Once again, Python tries to be efficient until we specify `list`

The final tool we will look at is `zip`

- map
- fold (reduce)
- filter
- **zip**

You may see `zip` come up less often, but it is handy when you need it

`zip` is mainly about rearranging our inputs to a more covenient form

```
def rectangle_area(pair):
    return pair[0]*pair[1]

xs = [1, 2, 3]
ys = [4, 5, 6]
areas = []
for i in range(len(xs)):
    areas.append(rectangle_area( (xs[i], ys[i]) ))
```

`xs` and `ys` are given separately, but we really care about `(x, y)` pairs

`zip` allows us to easily combine the lists in this way

```
(define xs '(1 2 3))
(define ys '(4 5 6))
(zip xs ys)
;Value: ((1 4) (2 5) (3 6))
```

Could then call `rectangle_area` directly on elements of `zipped` list

Python has `zip` as well

```
def rectangle_area(pair):
    return pair[0]*pair[1]

xs = [1, 2, 3]
ys = [4, 5, 6]
areas = []
for p in zip(xs, ys):
    areas.append(rectangle_area(p))
```

`zip` allows us to transform several lists of elements into single list of lists

Groups corresponding elements of lists together -- (first elements of each list), (second elements of each list), ...

Works with more than two lists both in Scheme and Python

```
(define xs '(1 2 3))
(define ys '(4 5 6))
(zip xs ys xs ys)
;Value: ((1 4 1 4) (2 5 2 5) (3 6 3 6))
```

Let's combine everything!

- **map**
- **fold** (reduce)
- **filter**
- **zip**

Here is previous example, already improved with `zip`

```
def rectangle_area(pair):
    return pair[0]*pair[1]

xs = [1, 2, 3]
ys = [4, 5, 6]
areas = []
for p in zip(xs, ys):
    areas.append(rectangle_area(p))
```

However, computing area is lame -- what if we wanted volume in 3d? Or whatever 4d equivalent is?

```
def block_volume(coords):
    volume = 1
    for c in coords:
        volume *= c
    return volume
```

But, we've seen this pattern before

```
def block_volume(coords):
    return reduce(lambda x, y: x*y, coords, 1)
```

Can we get rid of another `for` loop?

```
def block_volume(coords):
    return reduce(lambda x, y: x*y, coords, 1)

xs = [1, 2, 3]
ys = [4, 5, 6]
areas = []
for p in zip(xs, ys):
    areas.append(rectangle_area(p))
```

```
def block_volume(coords):
    return reduce(lambda x, y: x*y, coords, 1)

xs = [1, 2, 3]
ys = [4, 5, 6]
areas = map(block_volume, zip(xs, ys))
```

```
def block_volume(coords):
    return reduce(lambda x, y: x*y, coords, 1)

xs = [1, 2, 3]
ys = [4, 5, 6]
areas = map(block_volume, zip(xs, ys))
```

Simple, and, more importantly, general!

Works just as well in 3d, 4d, ...

```
def block_volume(coords):
    return reduce(lambda x, y: x*y, coords, 1)

xs = [1, 2, 3]
ys = [4, 5, 6]
zs = [7, 8, 9]
areas = map(block_volume, zip(xs, ys, zs))
```

If you try it, remember to use `list(areas)` to see values

As long as we're at it, let's keep only areas less than 100

```
def block_volume(coords):
    return reduce(lambda x, y: x*y, coords, 1)

xs = [1, 2, 3]
ys = [4, 5, 6]
zs = [7, 8, 9]
areas = filter(lambda x: x < 100,
          map(block_volume, zip(xs, ys, zs)))
```

No real reason, just because we can

Even if you never program in Scheme again, functional ideas appear in other languages and can give you another tool for decomposing problems and creating modular solutions

Note that if you are going to use functional ideas in Python in particular, you may want to look up generator expressions, which express similar ideas in a way many consider more "pythonic"