

## Lecture #6: Higher-Order Functions at Work

### Announcents:

- Free drop-in tutoring from HKN, the EECS honor society. Week-days 11am-5pm 345 Soda or 290 Cory. For more information see [hkn.eecs.berkeley.edu](http://hkn.eecs.berkeley.edu).
- A message from the AWE:  
"The Association of Women in EECS is hosting a 61A party this Sunday (2/9) from 1-3PM in the Woz! Come hang out, befriend other girls in 61A and meet AWE members who have taken it before! There will be lots of food, games, and fun!"
- Hog project released last Friday. Don't miss it!

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 1

## Iterative Update

- A general strategy for solving an equation:  

Guess a solution

  
while 

your guess isn't good enough

:  

update your guess
- The three boxed segments are parameters to the process.
- The last two segments clearly require functions for their representation—a *predicate* function (returning true/false values), and a function from values to values.
- In code,  

```
def iter_solve(guess, done, update):  
    """Return the result of repeatedly applying UPDATE,  
    starting at GUESS, until DONE yields a true value  
    when applied to the result. UPDATE takes a guess  
    and returns an updated guess."""  
    What goes here?
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 2

## Recursive Version (I)

```
def iter_solve(guess, done, update):  
    """Return the result of repeatedly applying UPDATE,  
    starting at GUESS, until DONE yields a true value  
    when applied to the result. UPDATE takes a guess  
    and returns an updated guess."""  
    if done(guess):  
        return guess  
    else:  
        return iter_solve(update(guess), done, update)
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 3

## Recursive Version (II)

```
def iter_solve(guess, done, update):  
    """Return the result of repeatedly applying UPDATE,  
    starting at GUESS, until DONE yields a true value  
    when applied to the result. UPDATE takes a guess  
    and returns an updated guess."""  
    def solution(guess):  
        if done(guess):  
            return guess  
        else:  
            return solution(update(guess))  
    return solution(guess)
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 4

## Iterative Version

```
def iter_solve(guess, done, update):  
    """Return the result of repeatedly applying UPDATE,  
    starting at GUESS, until DONE yields a true value  
    when applied to the result. UPDATE takes a guess  
    and returns an updated guess."""  
    while not done(guess):  
        guess = update(guess)  
    return guess
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 5

## Adding a Safety Net

- In real life, we might want to make sure that the function doesn't just loop forever, getting no closer to a solution.  

```
def iter_solve(guess, done, update, iteration_limit=32):  
    """Return the result of repeatedly applying UPDATE,  
    starting at GUESS, until DONE yields a true value  
    when applied to the result. Causes error if more than  
    ITERATION_LIMIT applications of UPDATE are necessary."""  
  
    def solution(guess, iteration_limit):  
        if done(guess):  
            return guess  
        elif iteration_limit <= 0:  
            raise ValueError("failed to converge")  
        else:  
            return solution(update(guess), iteration_limit-1)  
    return solution(guess, iteration_limit)
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 6

## Iterative Version with Safety Net.

```
def iter_solve(guess, done, update, iteration_limit=32):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS, until DONE yields a true value
    when applied to the result. Causes error if more than
    ITERATION_LIMIT applications of UPDATE are necessary."""

    while not done(guess):
        if iteration_limit <= 0:
            raise ValueError("failed to converge")
        guess, iteration_limit = update(guess), iteration_limit-1
    return guess
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 7

## Using Iterative Solving For Newton's Method

- *Newton's method* (aka the *Newton-Raphson method*) is a general numerical technique for finding approximate solutions to  $f(x) = 0$ , given the function  $f$ , its derivative  $f'$ , and an initial guess,  $x_0$ . It produces a result to some desired tolerance (that is, to some definition of "close enough").
- See <http://en.wikipedia.org/wiki/File:NewtonIterationAni.gif>
- Given a guess,  $x_k$ , compute the next guess,  $x_{k+1}$  by

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

```
def newton_solve(func, deriv, start, tolerance):
    """Return x such that |FUNC(x)| < TOLERANCE, given initial
    estimate START, assuming DERIV is the derivative of FUNC."""
    def close_enough(x):
        return abs(func(x)) < tolerance
    def newton_update(x):
        return x - func(x) / deriv(x)

    return iter_solve(start, close_enough, newton_update)
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 8

## Using newton\_solve for $\sqrt{\cdot}$ and $\sqrt[3]{\cdot}$

```
def square_root(a):
    return newton_solve(lambda x: x*x - a, lambda x: 2 * x,
                        a/2, a * 1e-10)

def cube_root(a):
    return newton_solve(lambda x: x**3 - a, lambda x: 3 * x ** 2,
                        a/3, a * 1e-10)
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 9

## Dispensing With Derivatives

- What if we just want to work with a function, without knowing its derivative?
- Book uses an approximation:  

```
def find_root(func, start=1, tolerance=1e-5):
    def approx_deriv(f, delta = 1e-5):
        return lambda x: (func(x + delta) - func(x)) / delta
    return newton_solve(func, approx_deriv(func), start, tolerance)
```
- This is nice enough, but looks a little ad hoc (how did I pick delta?).
- Another alternative is the *secant method*.

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 10

## The Secant Method

- Newton's method was

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

- The secant method uses that last two values to get (in effect) a replacement for the derivative:

$$x_{k+1} = x_k - f(x_k) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

- See [http://en.wikipedia.org/wiki/File:Secant\\_method.svg](http://en.wikipedia.org/wiki/File:Secant_method.svg)
- But this is a problem for us: so far, we've only fed the update function the value of  $x_k$  each time. Here we also need  $x_{k-1}$ .
- How do we generalize to allow arbitrary extra data (not just  $x_{k-1}$ )?

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 11

## Generalized iter\_solve

```
def iter_solve2(guess, done, update, state=None):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS and STATE, until DONE yields a true value
    when applied to the result. Besides a guess, UPDATE
    also takes and returns a state value, which is also passed to
    DONE."""
    while not done(guess, state):
        guess, state = update(guess, state)
    return guess
```

Last modified: Mon Feb 3 16:38:44 2014

CS61A: Lecture #5 12

## Using *Generalized* `iter_solve2` for the Secant Method

The secant method:

$$x_{k+1} = x_k - f(x_k) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

```
def secant_solve(func, start0, start1, tolerance):
    def close_enough(x, state):
        return abs(func(x)) < tolerance
    def secant_update(xk, xk1):
        return (xk - func(xk) * (xk - xk1)
                / (func(xk) - func(xk1),
                xk)
    return iter_solve2(start1, close_enough, secant_update, start0)
```