ENVIRONMENTS, LAMBDAS, AND NEWTON'S METHOD 3

COMPUTER SCIENCE 61A

February 7, 2013

1 Warmup

1. What would Python print?

```
>>> albert = 2
>>> def robert(albert):
...     def robert(albert):
...         albert += 2
...         return albert
...     return robert
>>> print(robert(albert)(albert))
______
>>> print(albert)
______
```

2 Lambda Expressions

One way of returning functions is by using nested inner functions. But, what if the function you need is very short and will only be used in one particular situation? The solution would be the `lambda` expression. A `lambda` expression has the following syntax:

```
lambda <args> :  <body>
```

With this simple expression, you can define functions on the fly, without having to use `def` statements and without having to give them names. In other words, lambda expressions allow you to create anonymous functions. There is a catch though; the body must be a single expression, which is also the return value of the function.

One other difference between using the `def` keyword and `lambda` expressions we would like to point out is that `def` is a *statement*, while `lambda` is an *expression*. Evaluating a `def` statement will have a side effect, namely it creates a new function binding in the current environment. On the other hand, evaluating a `lambda` expression will not change the environment unless we do something with the function created by the lambda. For instance, we could assign it to a variable or pass it as a function argument.

1. What would Python do?

```
>>> square = lambda x: x * x
>>> def double(f):
...     def doubler(x):
...         return f(f(x))
...     return doubler
>>> foo = double(square)
>>> foo(4)
```

2. Using a lambda function, complete the `make_offsetter` definition so that it returns a function. The new function should take one argument and returns that argument added to some `num`.

```
def make_offsetter(num):
    """
    Returns a function that takes one argument and returns
    num + some offset.

    >>> x = make_offsetter(3)
    >>> y = make_offsetter(8)
    >>> x(2)
    5
    >>> y(-1)
    7
    """
    return ________________________________________
```

3 Environment Diagrams with Higher Order Functions

Environment diagrams will feature prominently in CS61A, so here is one to try for practice. Environment diagrams can help you understand difficult coding problems, and also give you an idea of what's happening inside the interpreter.

1. Draw the environment diagram for the following code. What is x's value?

```
>>> a, b = 2, lambda x: x * 4
>>> def foo(bar, cond):
...     if cond:
...         return bar(a)
...     return b(a)
>>> x = foo(b, True)
```

2. Draw the environment diagram for the following code:

```
>>> from operator import add
>>> def curry2(f):
...     return lambda x: lambda y: f(x, y)
>>> make_adder  = curry2(add)
>>> add_three = make_adder(3)
>>> five = add_three(2)
```

4 Currying

We can transform multiple-argument functions into a chain of single-argument, higher order functions by taking advantage of lambda expressions. This is useful when dealing with functions that take only single-argument functions. We will see some examples of these later on.

1. Write a higher order function `rev_curry2` that reverses the order of the arguments of a curried function.

```
def rev_curry2(f):
    """
    Return a curried version of the given curried function,
    with the arguments reversed.

    >>> f = rev_curry2(curry2(lambda x, y: x / y))
    >>> f(4)(2)
    0.5
    """
```

5 Newton's Method

Newton's method is an algorithm that is widely used to compute the zeros of functions. It can be used to approximate a root of any continuous, differentiable function.

Intuitively, Newton's method works based on two observations:

- At a point $P = (x, f(x))$, a root of the function f is in the same direction relative to P as the root of the linear function L that not only passes through P, but also has the same slope as f at that point.
- Over any *very small* region, we can approximate f as a linear function. This is one of the fundamental principles of calculus.

Starting at an initial guess $(x_0, f(x_o))$, we estimate the function f as a linear function L, solve for the zero $(x', 0)$ of L, and then use the point $(x', f(x'))$ as the new guess for the root of f. We repeat this process until we have determined that $(x', f('x))$ is a zero of f.

Mathematically, we can derive the update equation by using two different ways to write the slope of L:

Let x be our current guess for the root, and x^* be the point we want to update our guess to. Let L be the linear function tangent to f at $(x, f(x))$.

Remember that x^* is the root of L. So, we know two points L passes through, namely $(x, f(x))$ and $(x^*, 0)$.

We can write the slope of L as

$$L'(x) = \frac{0 - f(x)}{x^* - x} = \frac{-f(x)}{x^* - x} \tag{1}$$

We also know that L is tangent to f as x, so:

$$L'(x) = f'(x) \tag{2}$$

We can equate these to get our update equation:

$$\frac{-f(x)}{x^* - x} = f'(x) \Rightarrow x^* = x - \frac{f(x)}{f'(x)} \tag{3}$$

We know $f(x)$, and from calculus, for some very small ε:

$$f'(x) = \frac{f(x+\varepsilon) - f(x)}{(x+\varepsilon) - x} = \frac{f(x+\varepsilon) - f(x)}{\varepsilon} \tag{4}$$

From the above, we get this algorithm:

```python
def approx_deriv(fn, x, dx=0.00001):
    return (fn(x+dx)-fn(x))/dx

def newtons_method(fn, guess=1, max_iterations=100):
    ALLOWED_ERROR_MARGIN = 0.0000001
    i = 1
    while abs(fn(guess)) > ALLOWED_ERROR_MARGIN and i <= max_iterations:
        guess = guess - fn(guess) / approx_deriv(fn, guess)
        i += 1
    return guess
```

We can generalize this idea into a framework known as *iterative improvement*. Basically, you start out by guessing a value, and then continuously update the guess until it is a reasonable approximation of the value we are looking for. Here is an implementation for `iter_improve`. The `update` function takes the current guess, and returns an updated guess. The `isdone` function also takes the current guess, and returns `True` if and only if the current guess is "good enough", according to some set criterion.

```
def iter_improve(update, isdone, guess=1, max_iterations=100):
    i = 1
    while not isdone(guess) and i <= max_iterations:
        guess = update(guess)
        i += 1
    return guess

def newtons_method2(fn, guess=1, max_iterations=100):
    def newtons_update(guess):
        return guess - fn(guess) / derivative(fn, guess)
    def newtons_isdone(guess):
        ALLOWED_ERROR_MARGIN= 0.0000001
        return abs(fn(guess)) <= ALLOWED_ERROR_MARGIN
    return iter_improve(newtons_update,
                        newtons_isdone,
                        max_iterations)
```

1. Write a function `cube_root` that computes the cube root of the input number `x`. (*Hint*: Use `newtons_method` with a function that is zero at the cube root of the input.)

```
def cube_root(x):
```

2. Newton's method converges very slowly (or not at all) if the algorithm happens to land on a point where the derivative is very small. Modify the implementation that uses `iter_improve` to return `None` if the derivative is under some threshold, say 0.001.

```
def newtons_method2(fn, guess=1, max_iterations=100):
    def newtons_update(guess, min_size=0.001):




    def newtons_done(guess):




    return iter_improve(newtons_update, newtons_done, guess,
        max_iterations)
```