61A LECTURE 5 – LAMBDA, NEWTON’S METHOD

Steven Tang and Eric Tzeng

June 30, 2013

Announcements

- hw2 is due tonight at 11:59PM; hw3 due **Saturday** at 11:59PM
- Project1 is due this Wednesday at 11:59PM
- Midterm 1 next Thursday, at 7PM
  - Covers everything THROUGH July 9 (includes Dictionaries, excludes Mutable Data)
  - 2050 VLSB: for logins cs61a-aa through cs61a-hz
  - 10 Evans: for everyone else
- Extra Office Hours: This Sunday from Noon to 6PM in 310 Soda
- **POTLUCK!**
  - **This Friday at 6pm in the Woz (4th floor Soda)**
    - **(Confirmation still pending though...)**
  - **If you're in Berkeley, come hang out with your staff! We'd love to get to know you better!**

Homework Syntax

- Starting with homework 2, if your file has a syntax error, you will automatically receive a zero.
- Before you submit, make sure that there are at least no syntax errors. Simply type in to the terminal or command prompt:

python filename.py

or, depending on your setup, python3 filename.py

- Other useful commands:

python3 –m doctest filename.py

python3 –i filename.py

Let’s recap...

- Last week we covered
  - Names as a means of abstraction
  - Functions as data
  - The environment model of computation
- This week:
  - More about function expressions (lambda)
  - Implementing Newton’s Method as an application
  - Recursion
  - Data abstraction

Lambda Expressions

```
>>> ten = 10
```

An expression: this one evaluates to a number

```
>>> square = x * x
```

Also an expression: evaluates to a function

```
>>> square = lambda x: x * x
```

Notice: no "return"

A function

with formal parameter x

and body "return x * x"

Must be a single expression

```
>>> square(4)
16
```

Lambda expressions are rare in Python, but important in general

Interpreter session

Evaluation of Lambda vs. Def

```
lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

Execution procedure for def *statements*:

1. Create a function value with signature <name>(<formal parameters>) and the current frame as parent
2. Bind <name> to that value in the current frame

Evaluation procedure for lambda *expressions*:

1. Create a function value with signature λ(<formal parameters>) and the current frame as parent
2. Evaluate to that value

No intrinsic name

Lambda vs. Def Statements

```
square = lambda x: x * x
```

VS

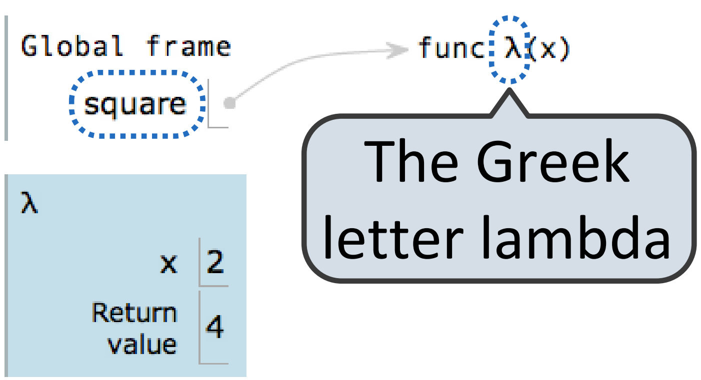
```
def square(x):
    return x * x
```

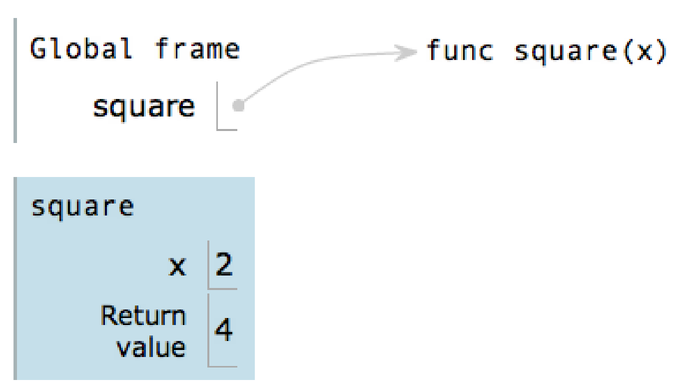
Both create a function with the same arguments & behavior

Both of those functions are associated with the environment in which they are defined

Both bind that function to the name "square"

Only the def statement gives the function an intrinsic name





Using Lambda

- Lambda expressions are useful when you want to quickly express a function, and don’t necessarily need to name it
- Also known as an “anonymous function” – no intrinsic name
- In Python, the body is only one expression. Other languages have more powerful lambdas.
- Demo

Short Break

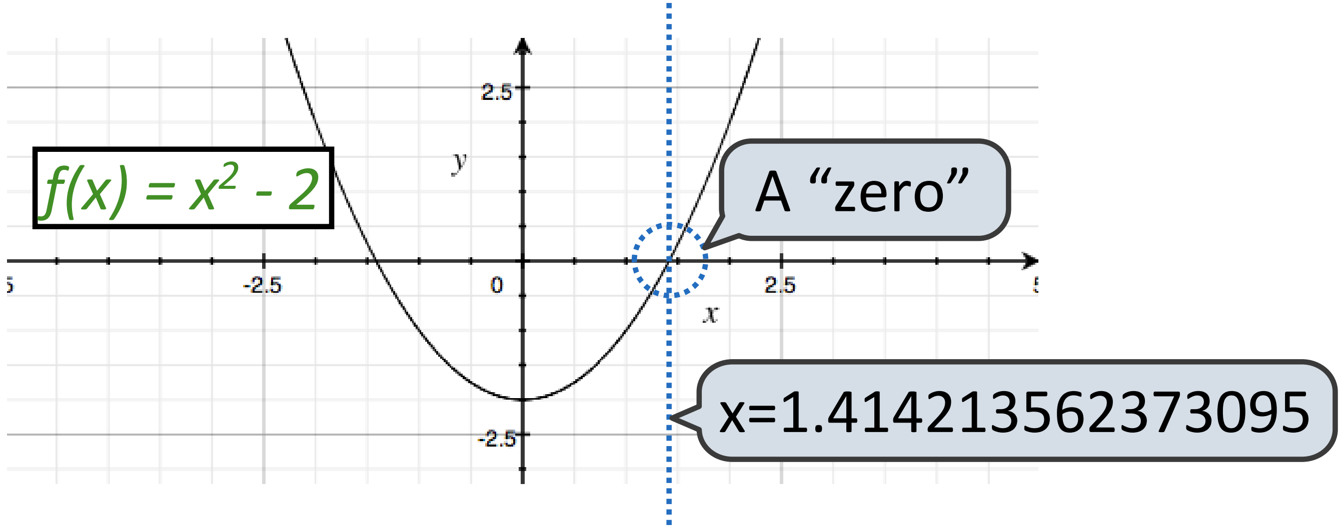
- Come up with any questions you have about lambda!
- http://goo.gl/bDm2W - Environment Diagram for example in code

Newton's Method

- So far, we've talked about a lot of syntax and abstract concepts
- Now, we're going to dive into an in-depth code example dealing with Newton's Method
- Newton's Method is used in a variety of real world applications
    - http://en.wikipedia.org/wiki/Newton%27s_method#Applications
- For CS61A, Newton's Method is a code example that makes use of HOFs and also implements the idea of "iterative improvement", which is a powerful programming technique

Newton's Method Background

Finds approximations to zeroes of differentiable functions



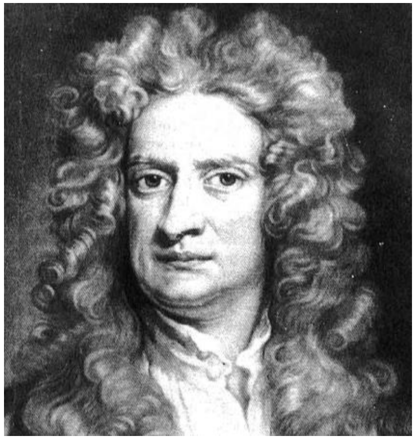
Application: a method for (approximately) computing square roots, using only basic arithmetic.

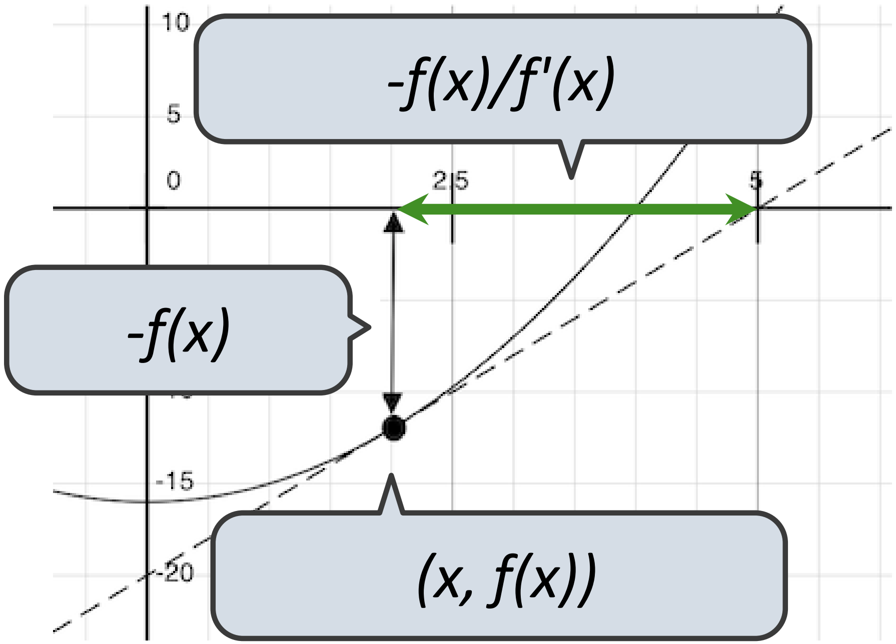
The positive zero of $f(x) = x^2 - a$ is $\sqrt{a}$

Newton's Method

Begin with a function f and

an initial guess x





Compute the value of *f* at the guess: *f(x)*

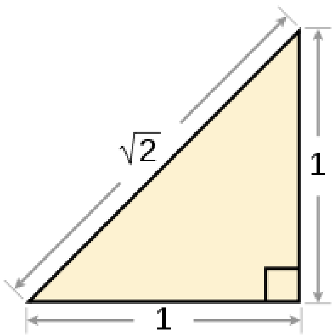
Compute the derivative of *f* at the guess: *f'(x)*

Update guess to be: $$x - \frac{f(x)}{f'(x)}$$

Visualization: http://upload.wikimedia.org/wikipedia/commons/e/e0/NewtonIteration_Ani.gif

Using Newton's Method

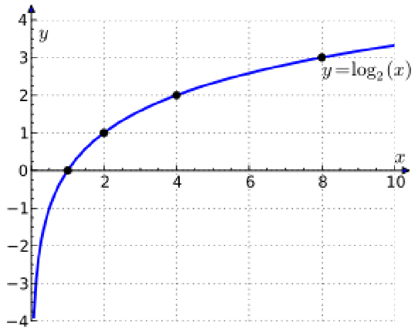
How to find the square root of 2?




```
>>> f = lambda x: x*x - 2
>>> find_zero(f)
1.4142135623730951
```

$f(x) = x^2 - 2$

How to find the log base 2 of 1024?




```
>>> g = lambda x: pow(2, x) - 1024
>>> find_zero(g)
10.0
```

$g(x) = 2^x - 1024$

Special Case: Square Roots

How to compute **`square_root(a)`**

Idea: Iteratively refine a guess *x* about the square root of *a*

Recall:
$f(x) = x^2 - a$
$f'(x) = 2x$

Update:

$$x = \frac{x + \frac{a}{x}}{2}$$

x - f(x)/f'(x)

Babylonian Method

Implementation questions:

What guess should start the computation?

How do we know when we are finished?

Special Case: Cube Roots

How to compute **`cube_root(a)`**

Idea: Iteratively refine a guess *x* about the cube root of *a*

Recall:
$f(x) = x^3 - a$
$f'(x) = 3x^2$

x - f(x)/f'(x)

Update: $$x = \frac{2x + \frac{a}{x^2}}{3}$$

Implementation questions:

What guess should start the computation?

How do we know when we are finished?

Interpreter Session

Iterative Improvement

First, identify common structure.

Then define a function that generalizes the procedure.

```
def iter_improve(update, done, guess=1, max_updates=1000):
    """Iteratively improve guess with update until done
    returns a true value.

    >>> iter_improve(golden_update, golden_test)
    1.618033988749895
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess
```

Newton's Method for nth Roots

```
def nth_root_func_and_derivative(n, a):
    def root_func(x):
        return pow(x, n) - a
    def derivative(x):
        return n * pow(x, n-1)
    return root_func, derivative

def nth_root_newton(a, n):
    """Return the nth root of a.

    >>> nth_root_newton(8, 3)
    2.0
    """
    root_func, deriv = nth_root_func_and_derivative(n, a)
    def update(x):
        return x - root_func(x) / deriv(x)
    def done(x):
        return root_func(x) == 0
    return iter_improve(update, done)
```

Exact derivative

$x - f(x)/f'(x)$

Definition of a function zero

Break

Factorial

The factorial of a non-negative integer *n* is

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1) * \cdots * 1, & n > 1 \end{cases}$$

$$(n-1)!$$

Factorial

The factorial of a non-negative integer *n* is

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

This is called a *recurrence relation*;

Factorial is defined in terms of itself

Can we write code to compute factorial using the same pattern?

Computing Factorial

We can compute factorial using the direct definition

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1) * \cdots * 1, & n > 1 \end{cases}$$

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    total = 1
    while n >= 1:
        total, n = total * n, n - 1
    return total
```

Computing Factorial

Can we compute it using the recurrence relation?

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

This is much shorter! But can a function call itself?

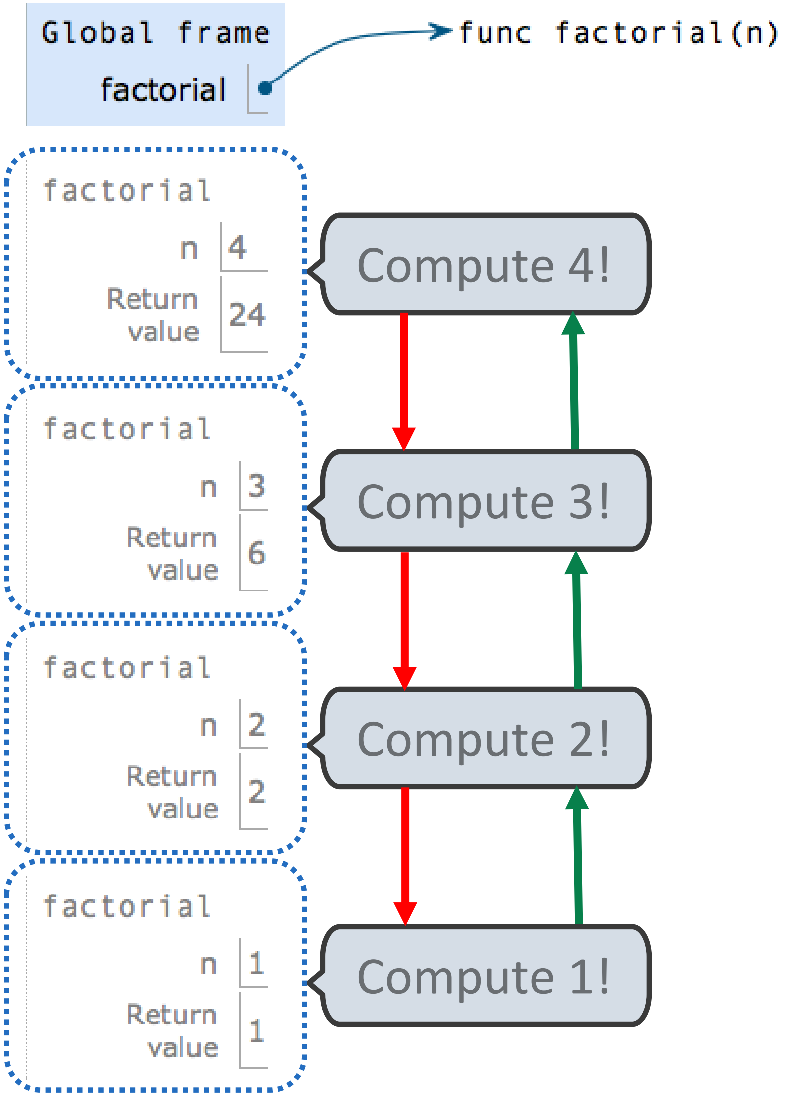
```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

Factorial Environment Diagram

Let’s see what happens!

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

factorial(4)
```



Recursive Functions

A function is *recursive* if the body calls the function itself, either directly or indirectly

Recursive functions have two important components:

1. *Base case(s)*, where the function directly computes an answer without calling itself
2. *Recursive case(s)*, where the function calls itself as part of the computation

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

Base case

Recursive case