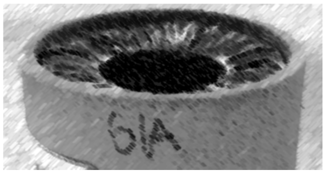


CS61A Lecture 8

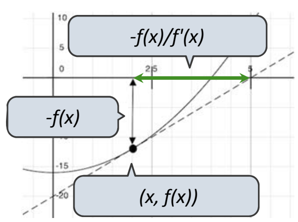
Amir Kamil
UC Berkeley
February 8, 2013

Announcements

- HW3 out, due Tuesday at 7pm
- Midterm next Wednesday at 7pm
  - Keep an eye out for your assigned location
  - Old exams posted
  - Review sessions
    - Saturday 2-4pm in 2050 VLSB
    - Extended office hours Sunday 11-3pm in 310 Soda
    - HKN review session Sunday 3-6pm in 145 Dwinelle
- Environment diagram handout on website
- Code review system online
  - See Piazza post for details

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://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif

Special Case: Square Roots

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

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

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*

Update: $x = \frac{2x + \frac{a}{x^2}}{3}$ *x - f(x)/f'(x)*

Implementation questions:

What guess should start the computation?

How do we know when we are finished?

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

Factorial

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

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

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?

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

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

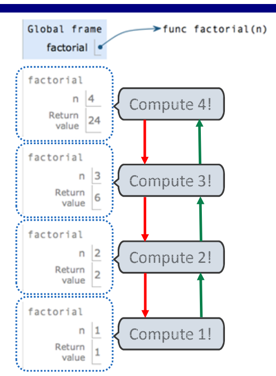
This is much shorter! But can a function call itself?

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)
```



Example: http://goo.gl/NjCKG

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

Practical Guidance: Choosing Names

Names typically don't matter for correctness,

but they matter tremendously for legibility

`boolean` → `turn_is_over`   `d` → `dice`   `play_helper` → `take_turn`

Use names for repeated compound expressions

```
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```
h = sqrt(square(a) + square(b))
if h > 1:
    x = x + h
```

Use names for meaningful parts of compound expressions

```
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
disc_term = sqrt(square(b) - 4 * a * c)
x = (-b + disc_term) / (2 * a)
```

Practical Guidance: DRY

Sometimes, removing repetition requires restructuring the code

```
def find_quadratic_root(a, b, c, plus=True):
    """Applies the quadratic formula to the polynomial
    ax^2 + bx + c."""
    if plus:
        return (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
    else:
        return (-b - sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```
def find_quadratic_root(a, b, c, plus=True):
    """Applies the quadratic formula to the polynomial
    ax^2 + bx + c."""
    disc_term = sqrt(square(b) - 4 * a * c)
    if not plus:
        disc_term *= -1
    return (-b + disc_term) / (2 * a)
```

Test-Driven Development

Write the test of a function before you write a function

- A test will clarify the (one) job of the function
- Your tests can help identify tricky edge cases

Develop incrementally and test each piece before moving on

- You can't depend upon code that hasn't been tested
- Run your old tests again after you make new changes