CS61A Lecture 6
Recursion

Tom Magrino and Jon Kotker
UC Berkeley EECS

June 26, 2012

Computer Science in the News


The New York Times
Business Day
Technology
WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION

How Many Computers to Identify a Cat? 16,000

Jim Wilson/The New York Times

An image of a cat that a neural network taught itself to recognize.

By JOHN MARKOFF
Published: June 25, 2012

MOUNTAIN VIEW, Calif. — Inside Google's secretive X laboratory, known for inventing self-driving cars and augmented reality glasses, a small group of researchers began working several years ago on a simulation of the human brain.

FACEBOOK
TWITTER
GOOGLE+
EMAIL

2

Today

- Quick review of Iterative Improvement.
- Defining functions that call themselves.
  - Sometimes more than once.

3

Recap: Newton's Method

```
def iter_improve(update, isclose, guess=1):
    while not isclose(guess):
        guess = update(guess)
    return guess

def find_root(f, initial_guess=10):
    return iter_improve(newton_update(f),
                        lambda x: approx_zero(f(x)),
                        initial_guess)
```

4

Recap: Newton's Method

Incredibly powerful, but does not always work!
Certain conditions need to be satisfied: for example, the function needs to be differentiable.

The method can fail in many ways, including:

1. Infinite loop among a set of guesses. (Try $f(x) = x^3 - 2x + 2$.)
2. Guesses may never fall within the tolerance for approximate equality.
3. Guesses converge to the answer *very slowly*.

5

Recap: Newton's Method
Iterative Improvement (with one fix)

We can add a limit on the number of *iterations*.

```
def iter_improve(update, isclose, guess=1, max_iter=5000):
    counter = 1
    while not isclose(guess) and counter <= max_iter:
        guess = update(guess)
        counter += 1
    return guess
```

6

Practice: Newton's Method

Using `find_root`, write a function `intersection(f, g)` which takes two functions, `f` and `g`, and finds a point at which the two are equal.

```
def intersection(f, g):
```


Practice: Newton's Method

Using `find_root`, write a function `intersection(f, g)` which takes two functions, `f` and `g`, and finds a point at which the two are equal.

```
def intersection(f, g):
    return find_root(lambda x: f(x) - g(x))
```


Computing Factorial

The factorial of a positive integer n is:

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

Computing Factorial

The factorial of a positive integer *n* was:

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

Computing Factorial

The factorial of a positive integer n was:

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

We *generalized* our definition. Can we do that with our code?

Computing Factorial

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

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

Can we generalize here like we did with our mathematical expression?

Computing Factorial

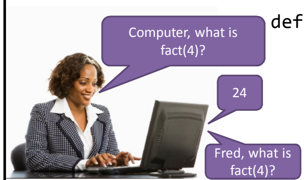
$$n! = \begin{cases} 1, & n = 0 \; or \; n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

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

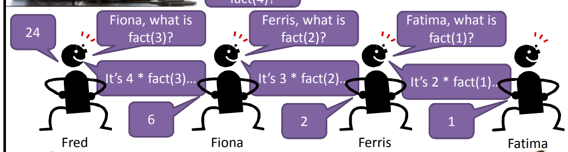
http://cdn.shopify.com/s/files/1/0070/7032/files/wat_grande.jpg?113123

How can `fact` be defined by calling `fact`?!?!?!

Computing Factorial




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



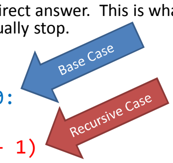
Recursive Functions

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

Recursive functions typically have 2 main pieces:

1. Recursive case(s), where the function calls itself.
2. Base case(s), where the function does NOT recursively call itself and instead returns a direct answer. This is what ensures that the recursion will eventually stop.

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



Recursion in Every Day Life: Eating Chocolate

You have a bar of chocolate with n small pieces. How do you eat it?

1. You eat 1 piece of chocolate.
2. You eat a bar of n – 1 pieces of chocolate.

What's your base case?

– You have no more chocolate.

http://i0.kym-cdn.com/photos/images/original/000/000/578/1234931504682.jpg

Practice: Recursion

How would I rewrite the summation function from last week to use recursion?

```
def summation(n, term):
```


Practice: Recursion

How would I rewrite the summation function from last week to use recursion?

```
def summation(n, term):
    if n == 0:
        return 0
    return term(n) + summation(n - 1, term)
```

Practice: Recursion

What does the following function calculate?

```
def fun(a, b):
    if b == 0:
        return 0
    elif b % 2 == 0:
        return fun(a + a, b / 2)
    return fun(a, b – 1) + a
```


Practice: Recursion

What does the following function calculate?

```
def fun(a, b):
    if b == 0:
        return 0
    elif b % 2 == 0:
        return fun(a + a, b / 2)
    return fun(a, b – 1) + a
a * b
```


Practice: Recursion

Using recursion, write the function `log(b, x)` which finds $log_b(x)$, assuming x is some power of b.

```
def log(b, x):
```


Practice: Recursion

Using recursion, write the function `log(b, x)` which finds $log_b(x)$, assuming x is some power of b.

```
def log(b, x):
    if x == 1:
        return 0
    return 1 + log(b, x / b)
```


Announcements

- Bug-Submit is now available!
- Project 1 is due Friday
- Homework 3 is due Friday

Tree Recursion

You can have a function defined in terms of itself using more than one recursive call. This is called ***tree recursion.***

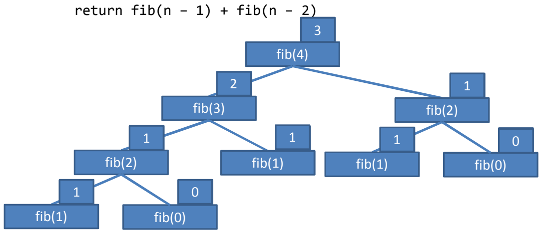
$$F_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F_{n-1} + F_{n-2}, & n > 1 \end{cases}$$

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n – 1) + fib(n – 2)
```

Tree Recursion

```
def fib(n):
        if n == 0:
                return 0
        elif n == 1:
                return 1
        return fib(n - 1) + fib(n - 2)
```

So why is it called ***tree recursion***?



Practice: Tree Recursion

Suppose I want to count all the different routes I could take from (0, 0) to (x, y) on a grid moving only up and right. Write the function `paths(x, y)` to calculate the number of routes to (x, y).

```
def paths(x, y):
```


Practice: Tree Recursion

Suppose I want to count all the different routes I could take from (0, 0) to (x, y) on a grid moving only up and right. Write the function `paths(x, y)` to calculate the number of routes to (x, y).

```
def paths(x, y):
    if x == 0 or y == 0:
        return 1
    return paths(x - 1, y) \
           + paths(x, y - 1)
```


Practice: Tree Recursion

Suppose I want to **print** all the different routes I could take from (0, 0) to (x, y) on a grid moving only up and right. Write the function `directions(x, y)` which prints the each different set of directions using a combination of “UP” and “RIGHT” that one could take. *Hint:* use a helper function that does the recursion and keeps track of the “directions so far.”

```
def directions(x, y):
```


Practice: Tree Recursion

Suppose I want to **print** all the different routes I could take from (0, 0) to (x, y) on a grid moving only up and right. Write the function `directions(x, y)` which prints the each different set of directions using a combination of “UP” and “RIGHT” that one could take. *Hint:* use a helper function that does the recursion and keeps track of the “directions so far.”

```
def directions(x, y):
        def dir_helper(x, y, so_far):
                if x == 0 and y == 0:
                        print(so_far)
                elif y > 0:
                        dir_helper(x, y - 1, so_far + “ UP”)
                elif x > 0:
                        dir_helper(x-1, y, so_far + “ RIGHT”)
        dir_helper(x, y, “”)
```


Conclusion

- Recursion is a way for functions to be defined using themselves.
- Recursive functions have two parts:
  - Recursive case(s), where the function calls itself.
  - Base case(s), where the function does not call itself (stopping the recursion).
- Tree recursion is used by functions that make more than one recursive call (at the same time) in the definition.