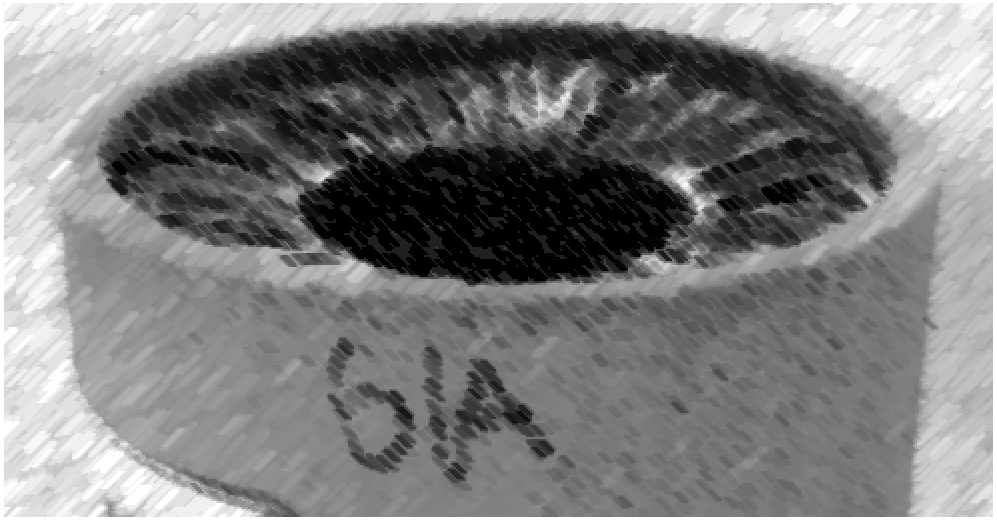


CS61A Lecture 9

Amir Kamil
UC Berkeley
February 11, 2013

Announcements

- HW3 due Tuesday at 7pm
- Hog due today!
    - Hog contest due later; see announcement tonight
- Midterm Wednesday at 7pm
    - See course website for assigned locations, more info
- Midterm review in lab this week

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

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

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_iter(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?

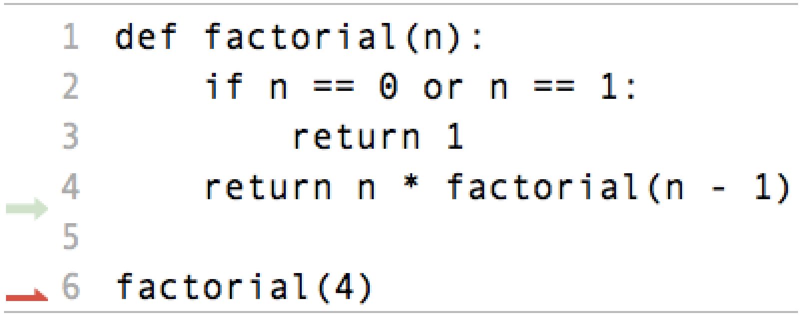
$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

```python
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?

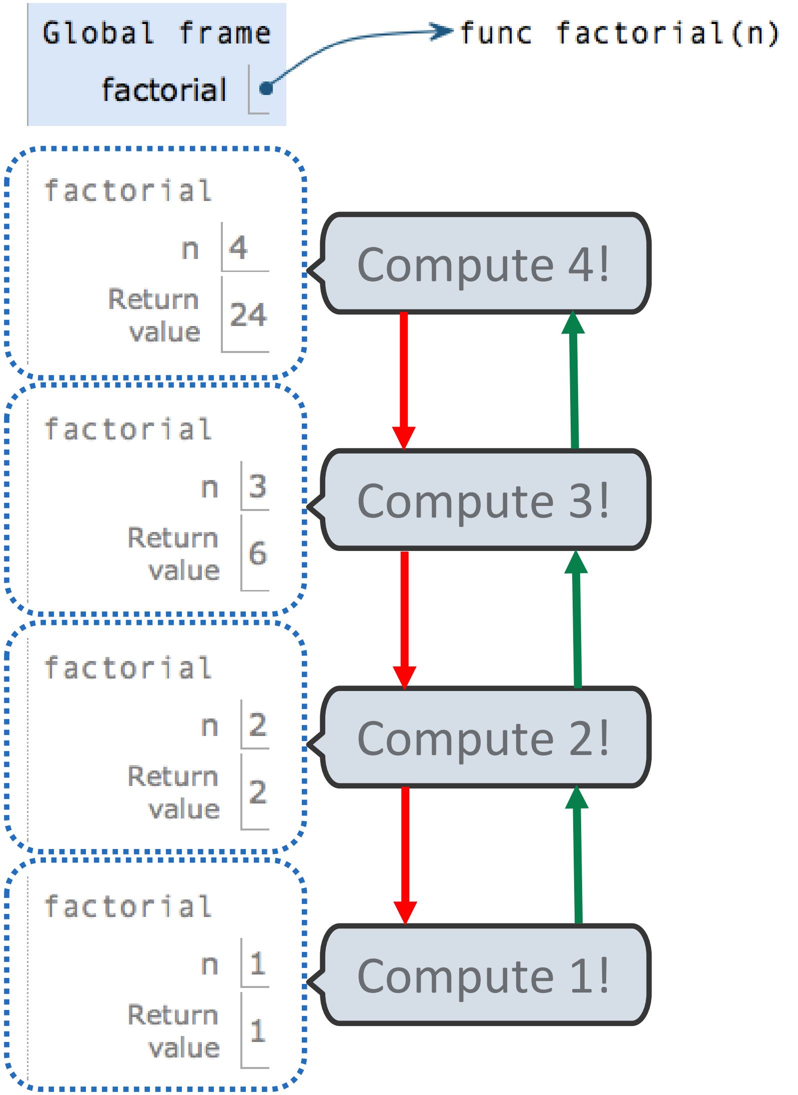
Factorial Environment Diagram

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

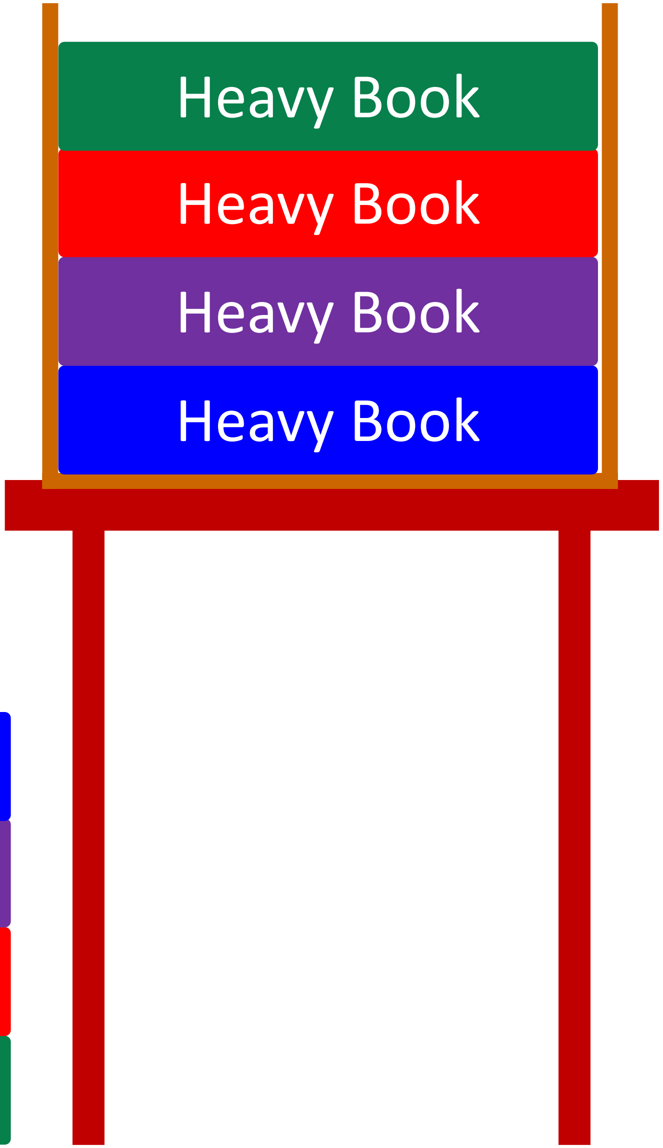
```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

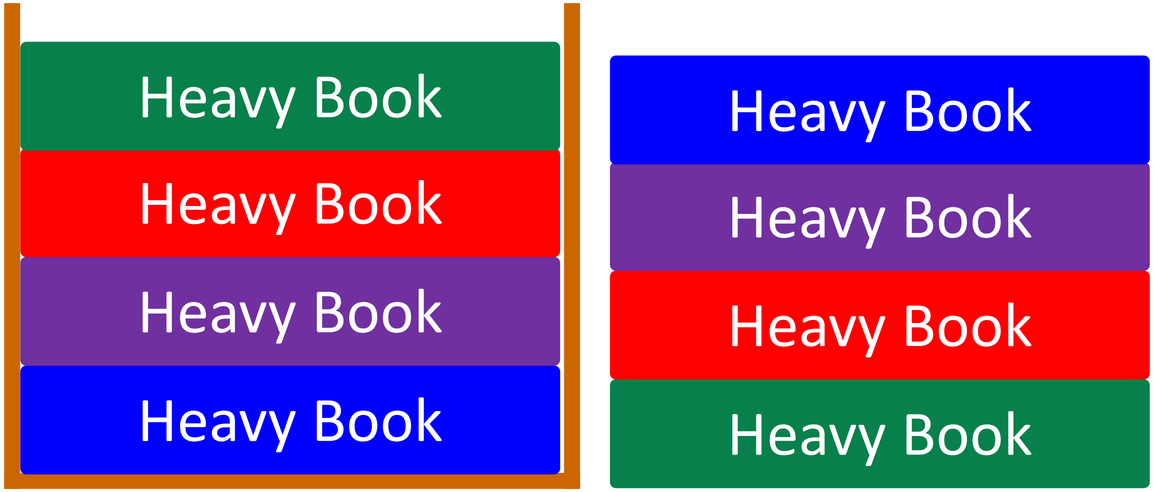
Base case

Recursive case

Recursion Example: Heavy Box

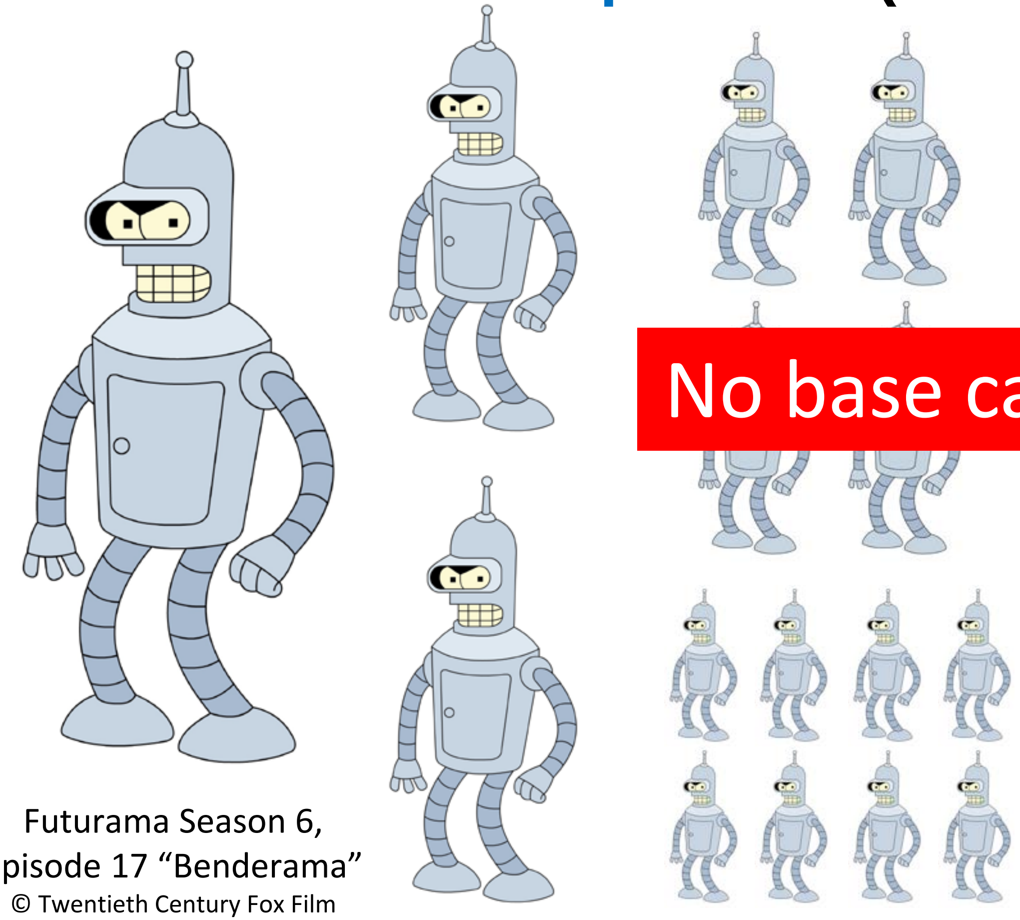
```
def lift_box(box):
    if too_heavy(box):
        book = remove_book(box)
        lift_box(box)
        add_book(box, book)
    else:
        move_box(box)
```



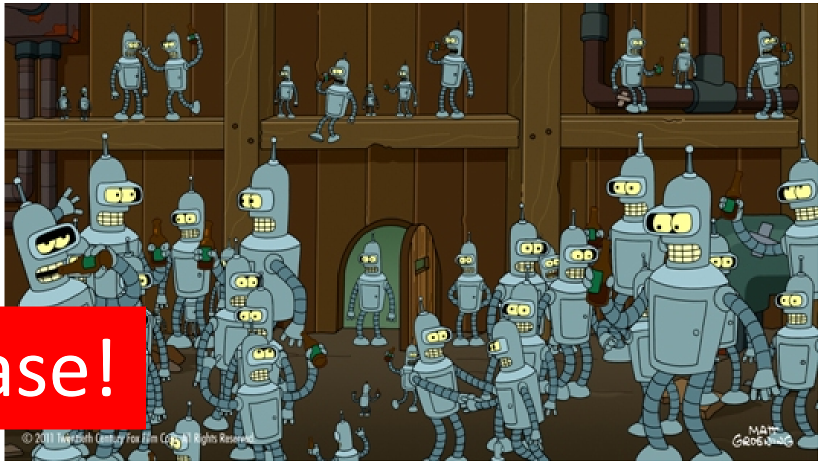


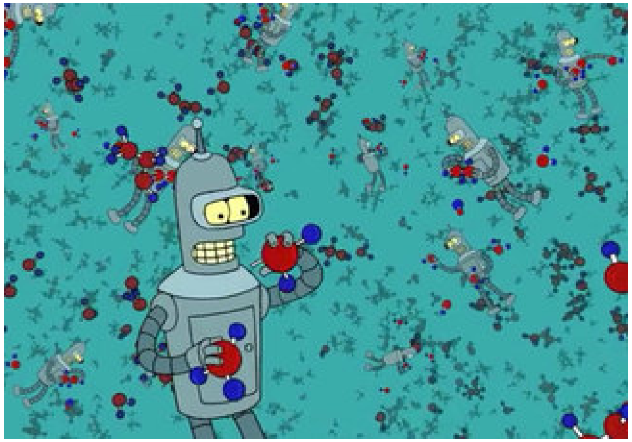
Recursion Example: Duplication

```
def duplicate(size):
    return (duplicate(0.6 * size) +
            duplicate(0.6 * size))
```



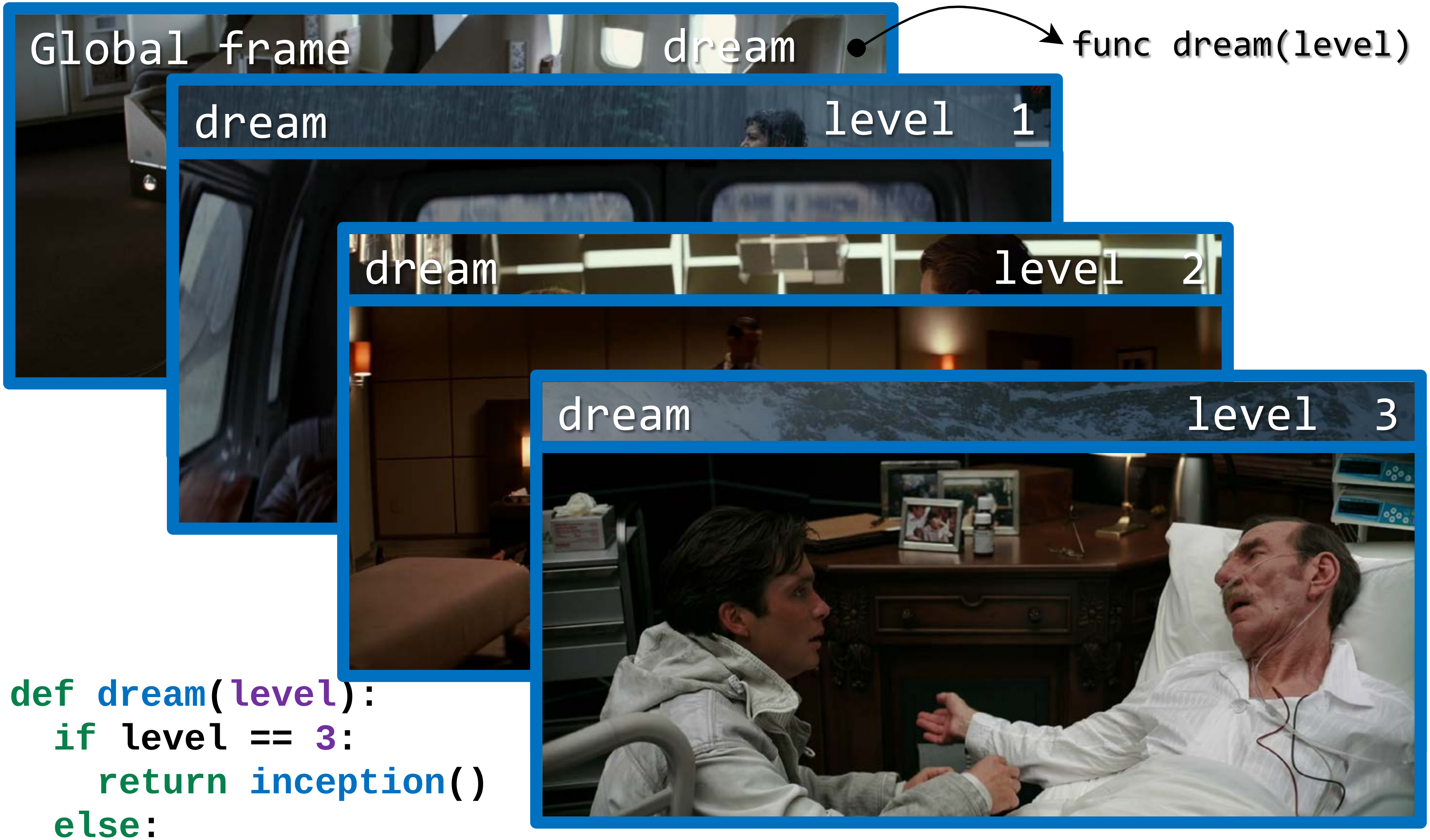
No base case!





Futurama Season 6, Episode 17 “Benderama”

Recursion Example: Dreaming



```
def dream(level):
  if level == 3:
    return inception()
  else:
    return dream(level + 1)
```

Inception
© Warner Brothers Entertainment

Reversing the Order of Recursive Calls

Some recursive computations may be done more easily by reversing the order of recursive calls.

A *helper function* helps us to do this.

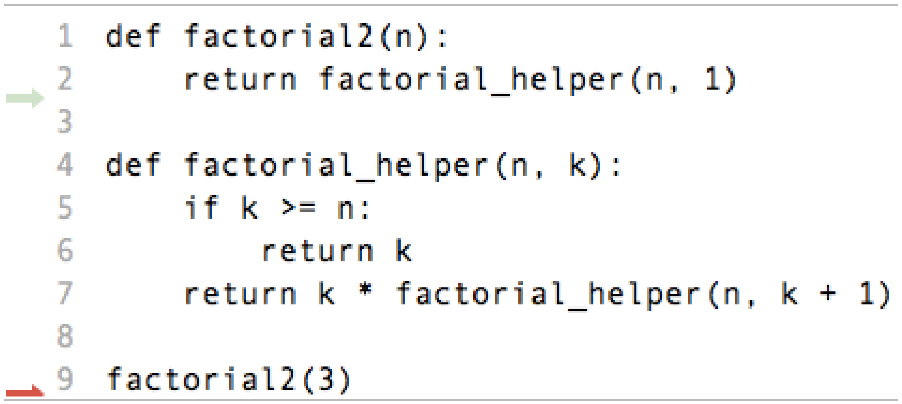
$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n-1)!, & n > 1 \end{cases}$$

```python
def factorial2(n):
    return factorial_helper(n, 1)

def factorial_helper(n, k):
    if k >= n:
        return k
    return k * factorial_helper(n, k + 1)
```

Reverse Environment Diagram

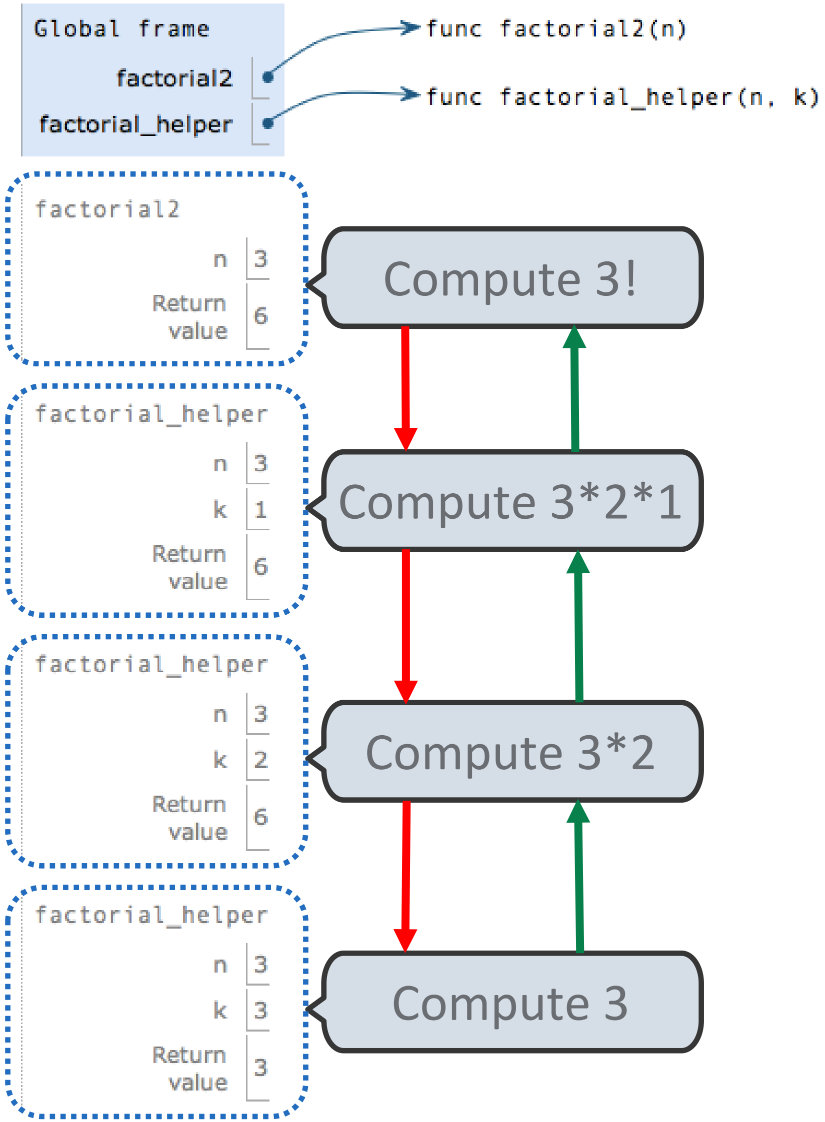
Here is how the reversed computation evolves



```
def factorial2(n):
    return factorial_helper(n, 1)

def factorial_helper(n, k):
    if k >= n:
        return k
    return k * factorial_helper(n, k + 1)

factorial2(3)
```



Example: http://goo.gl/6zz0z

Fibonacci Sequence

The Fibonacci sequence is defined as

$$\mathrm{fib}(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2), & n > 1 \end{cases}$$

```
def fib_iter(n):
    if n == 0:
        return 0
    fib_n, fib_n_1 = 1, 0
    k = 1
    while k < n:
        fib_n, fib_n_1 = fib_n_1 + fib_n, fib_n
        k += 1
    return fib_n
```

Example: http://goo.gl/9UJxG

Fibonacci Sequence

The Fibonacci sequence is defined as

$$\text{fib}(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2), & n > 1 \end{cases}$$

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

Two recursive calls!

Example: http://goo.gl/DZbRG

Tree recursion

Executing the body of a function may entail more than one recursive call to that function

This is called *tree recursion*