



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



Factorial



The factorial of a non-negative integer n is

Factorial



The factorial of a non-negative integer n is

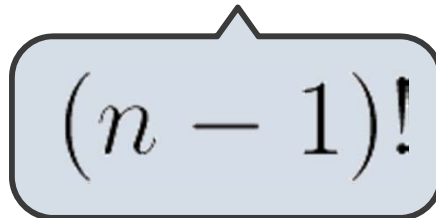
$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n * (n - 1) * \cdots * 1, & n > 1 \end{cases}$$

Factorial



The factorial of a non-negative integer n is

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

A diagram illustrating the recursive definition of factorial. A blue dotted oval encircles the expression $(n - 1) * \dots * 1$ in the equation above. A callout box with a light blue background and a black border points to this expression, containing the text $(n - 1)!$.
$$(n - 1)!$$

Factorial



The factorial of a non-negative integer n is

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

Factorial



The factorial of a non-negative integer n is

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

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



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

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



Computing Factorial



We can compute factorial using the direct definition

Computing Factorial



We can compute factorial using the direct definition

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

Computing Factorial



We can compute factorial using the direct definition

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

```
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



Computing Factorial



Can we compute it using the recurrence relation?

Computing Factorial



Can we compute it using the recurrence relation?

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

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)
```

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)
```

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



Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

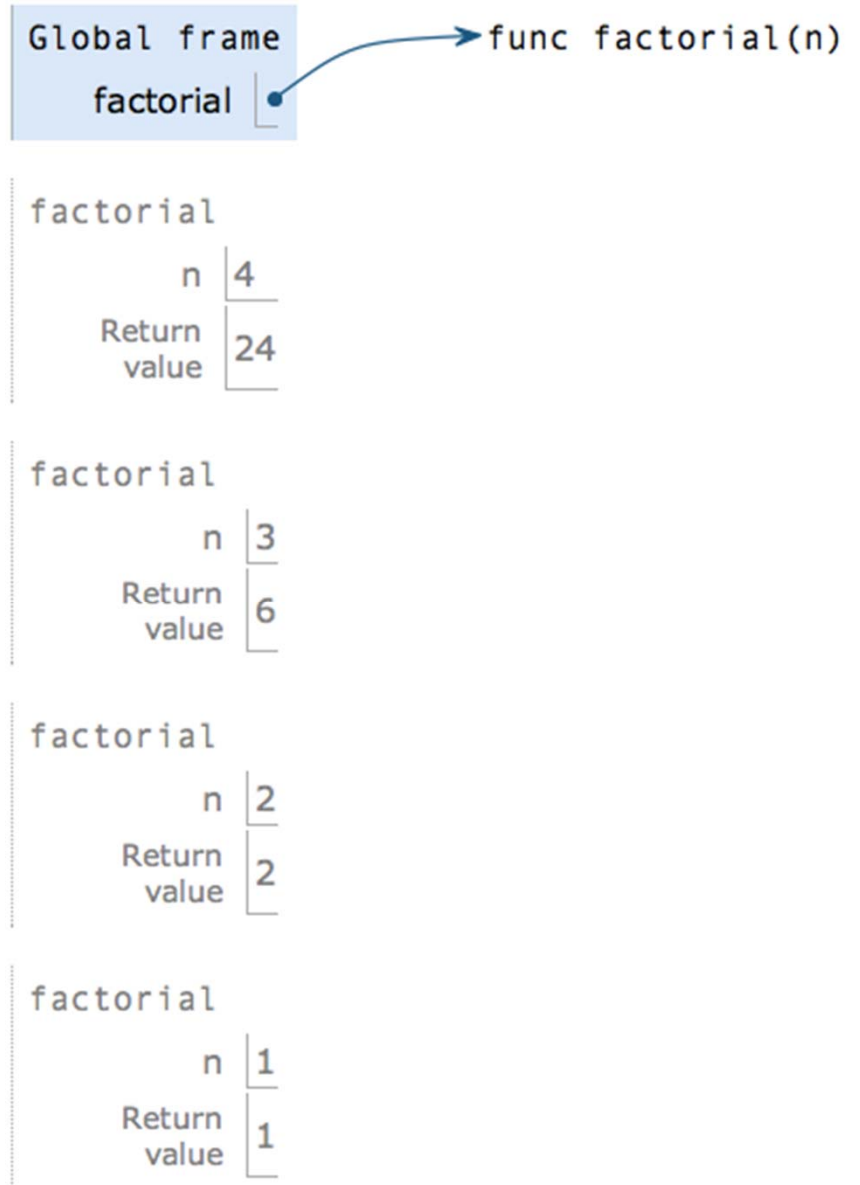
```
1 def factorial(n):
2     if n == 0 or n == 1:
3         return 1
4     return n * factorial(n - 1)
5
6 factorial(4)
```

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



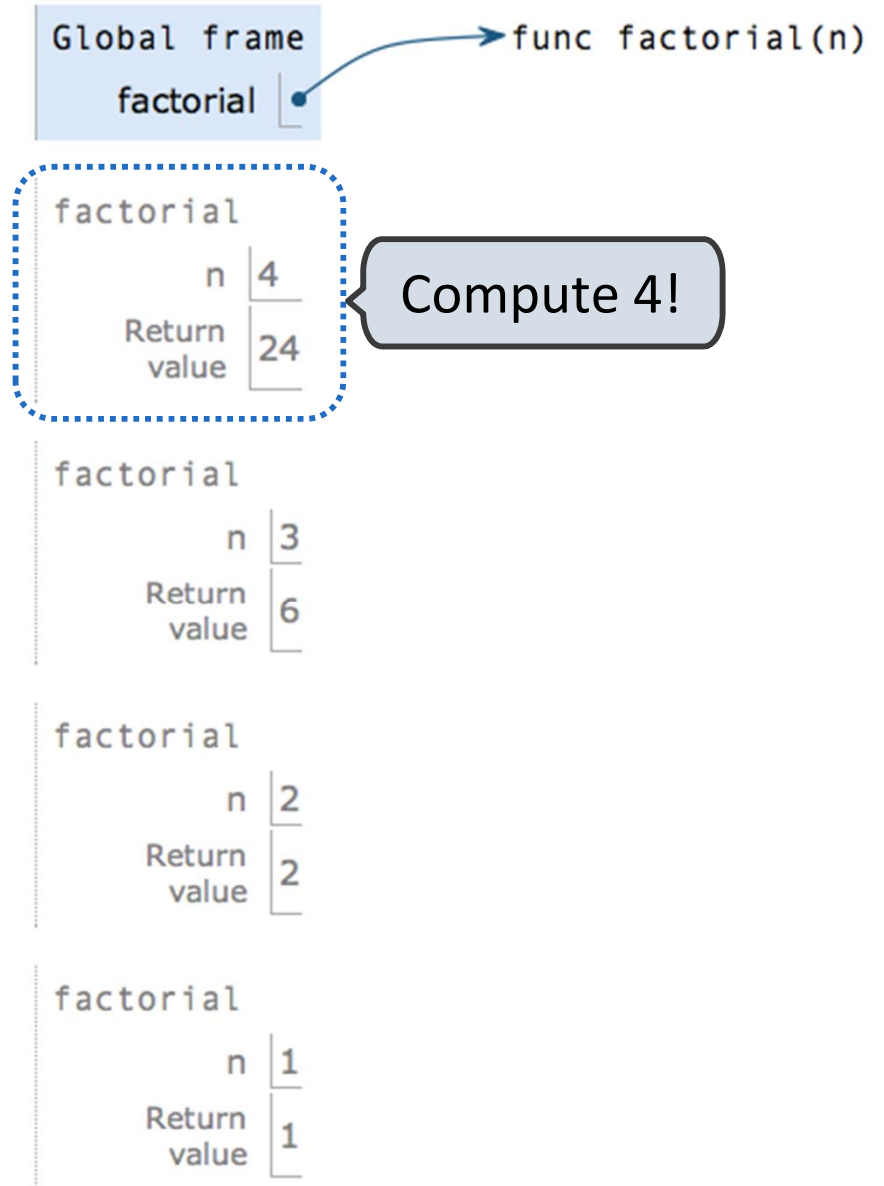
Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



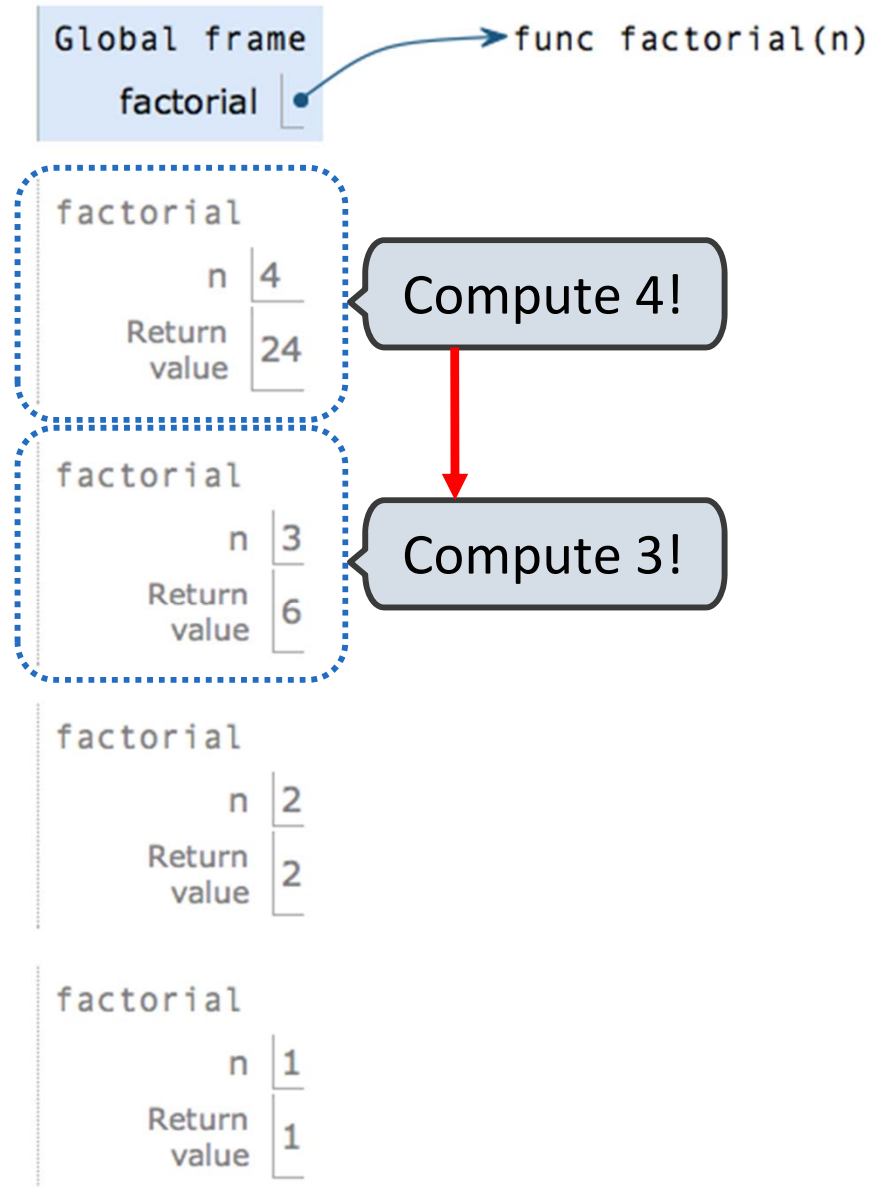
Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



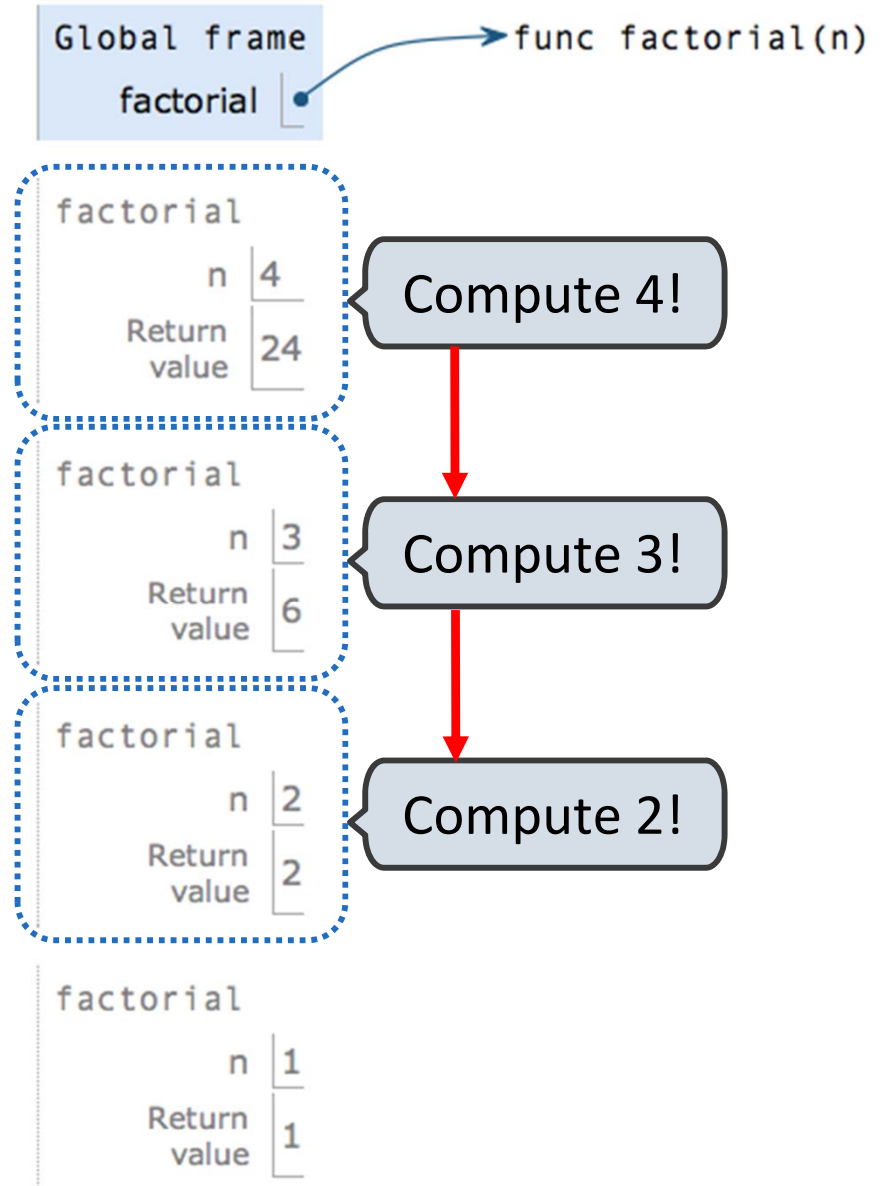
Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



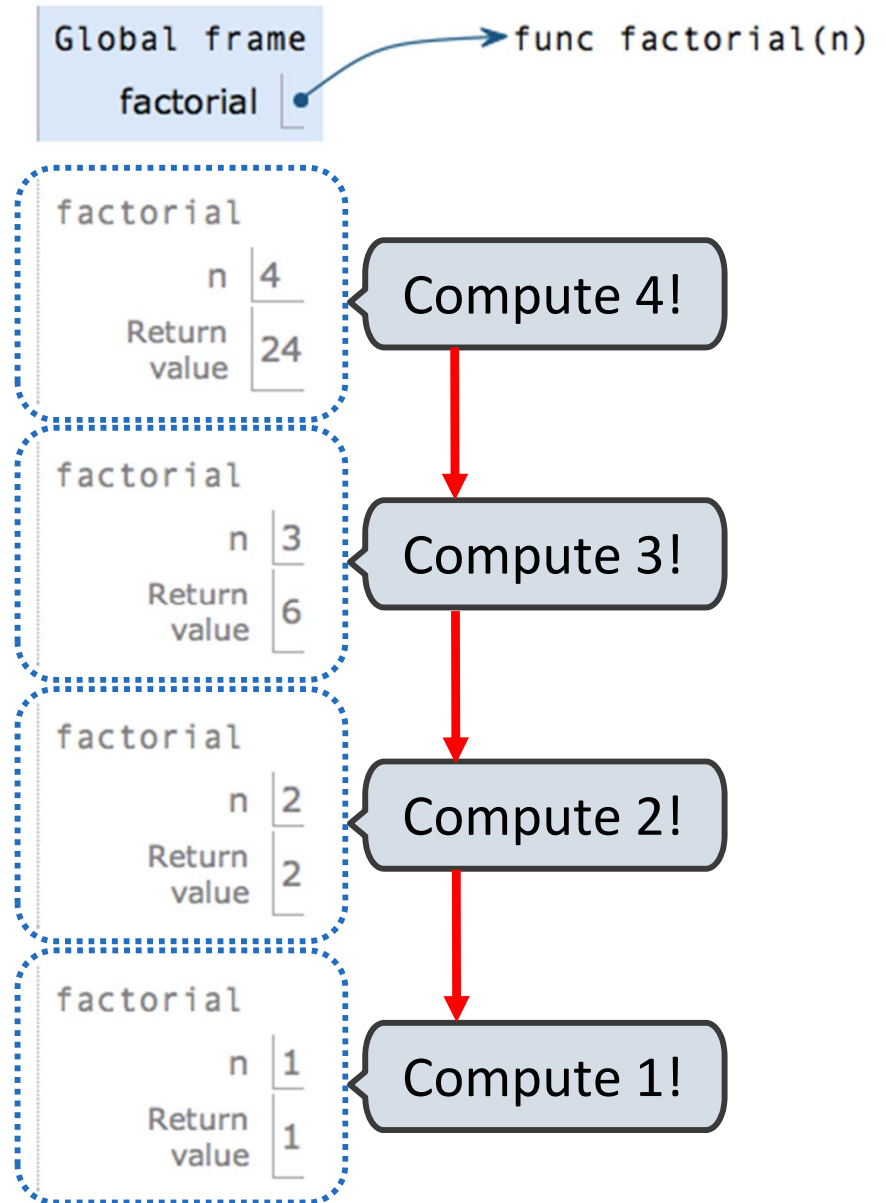
Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



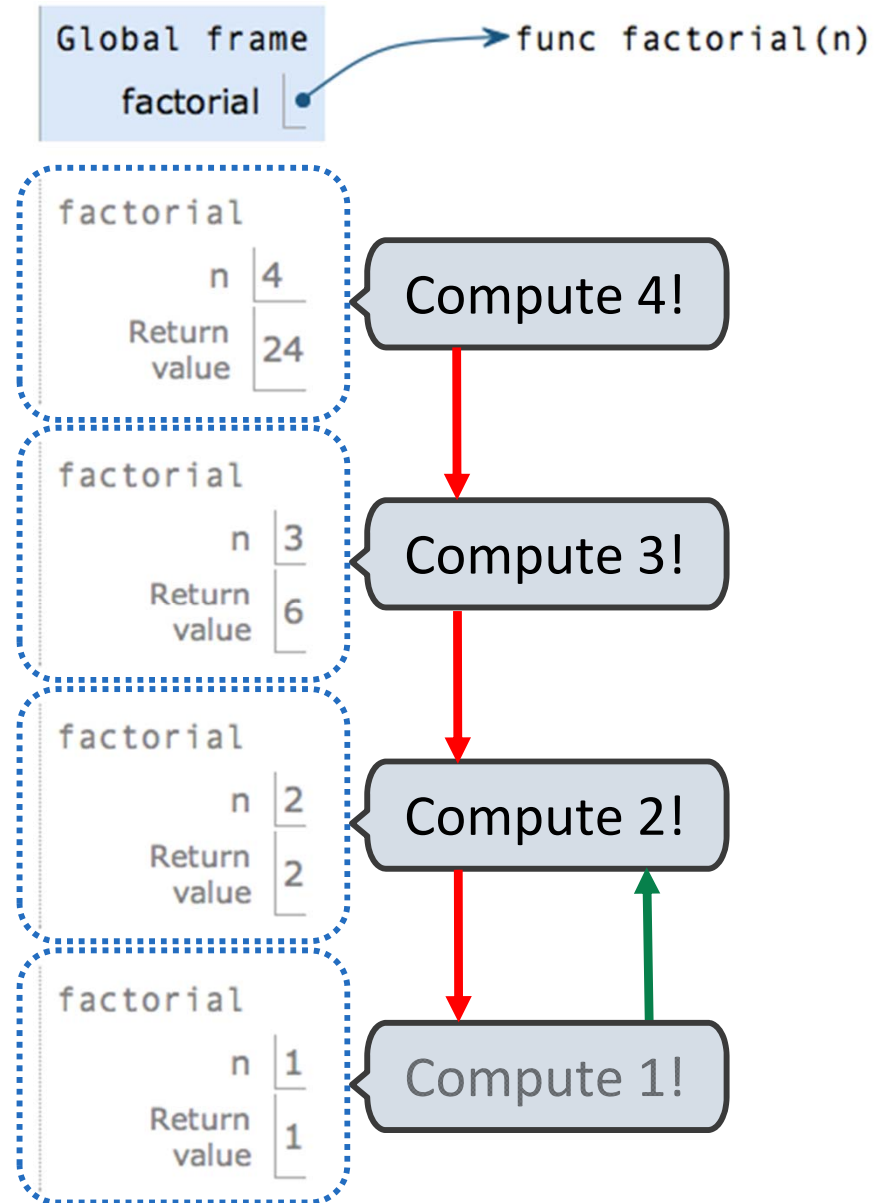
Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



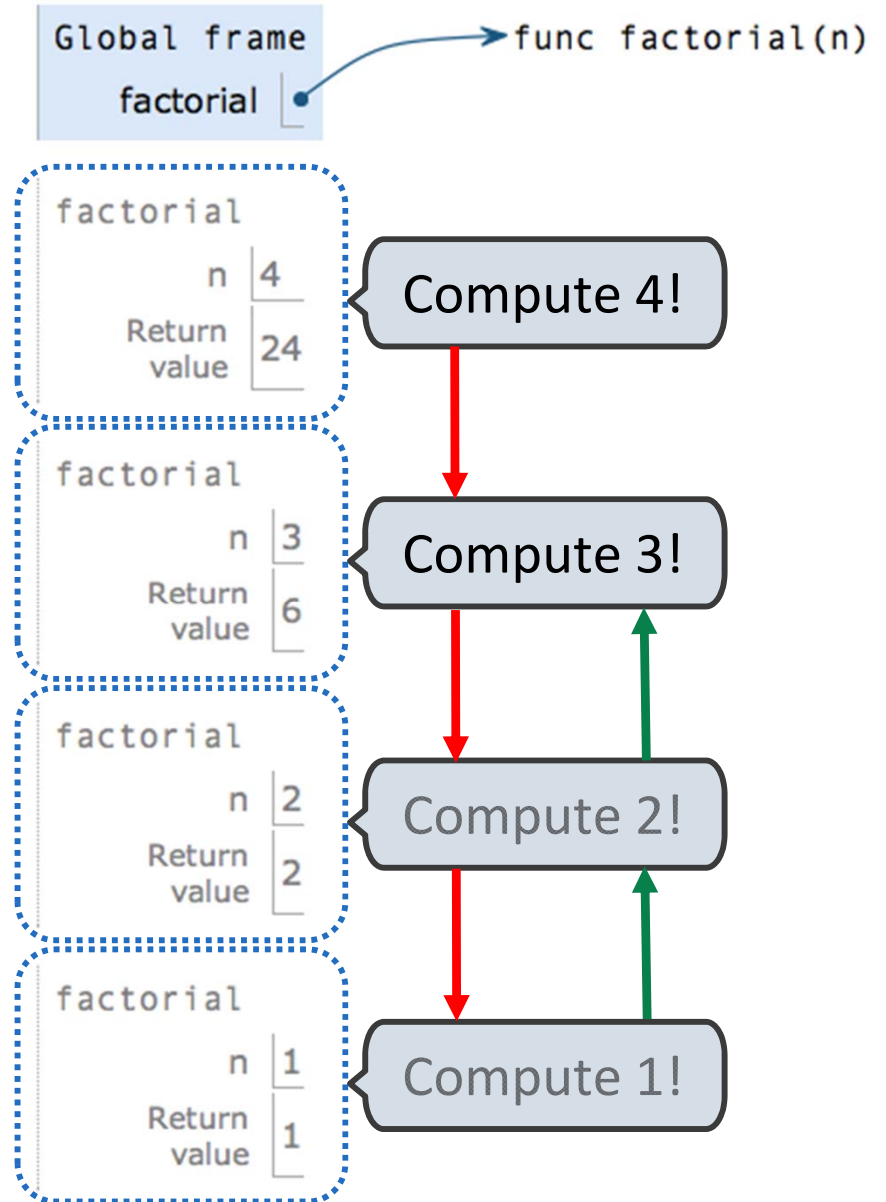
Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



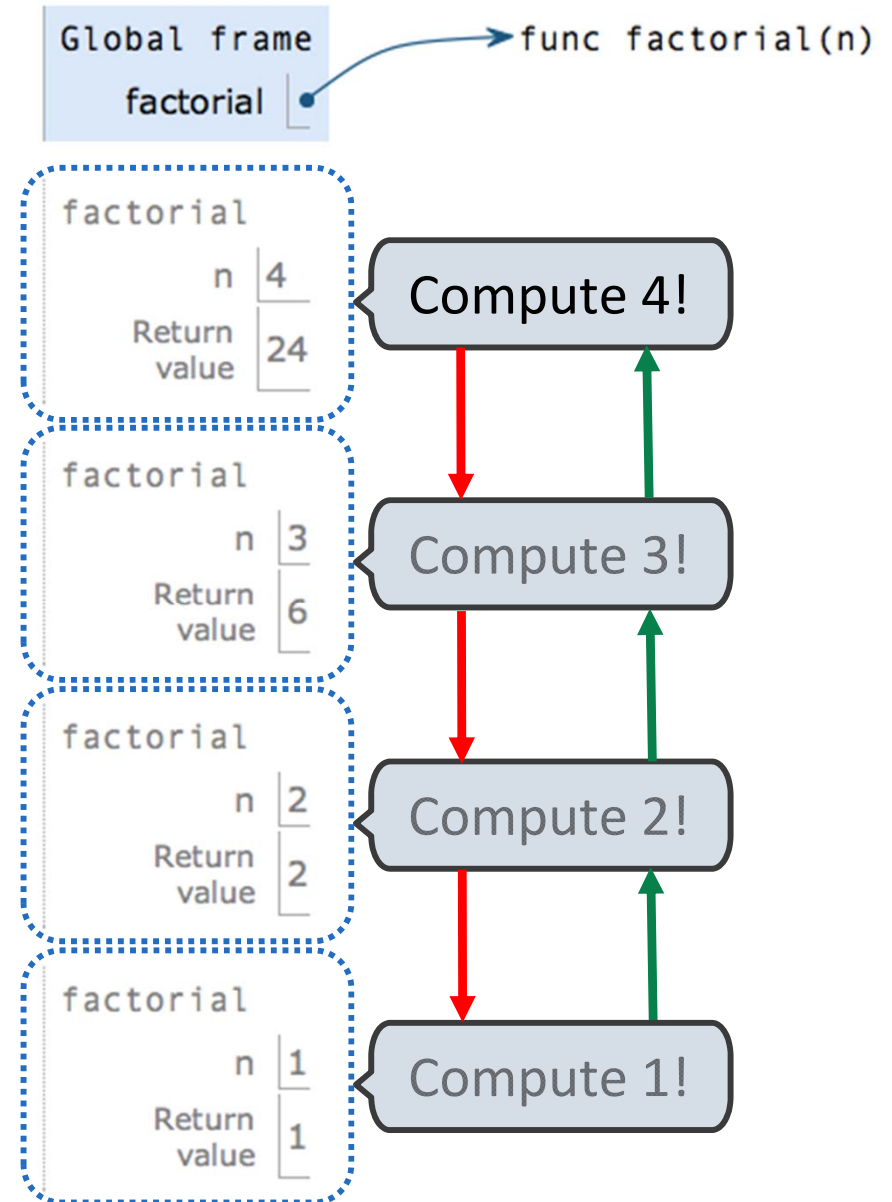
Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



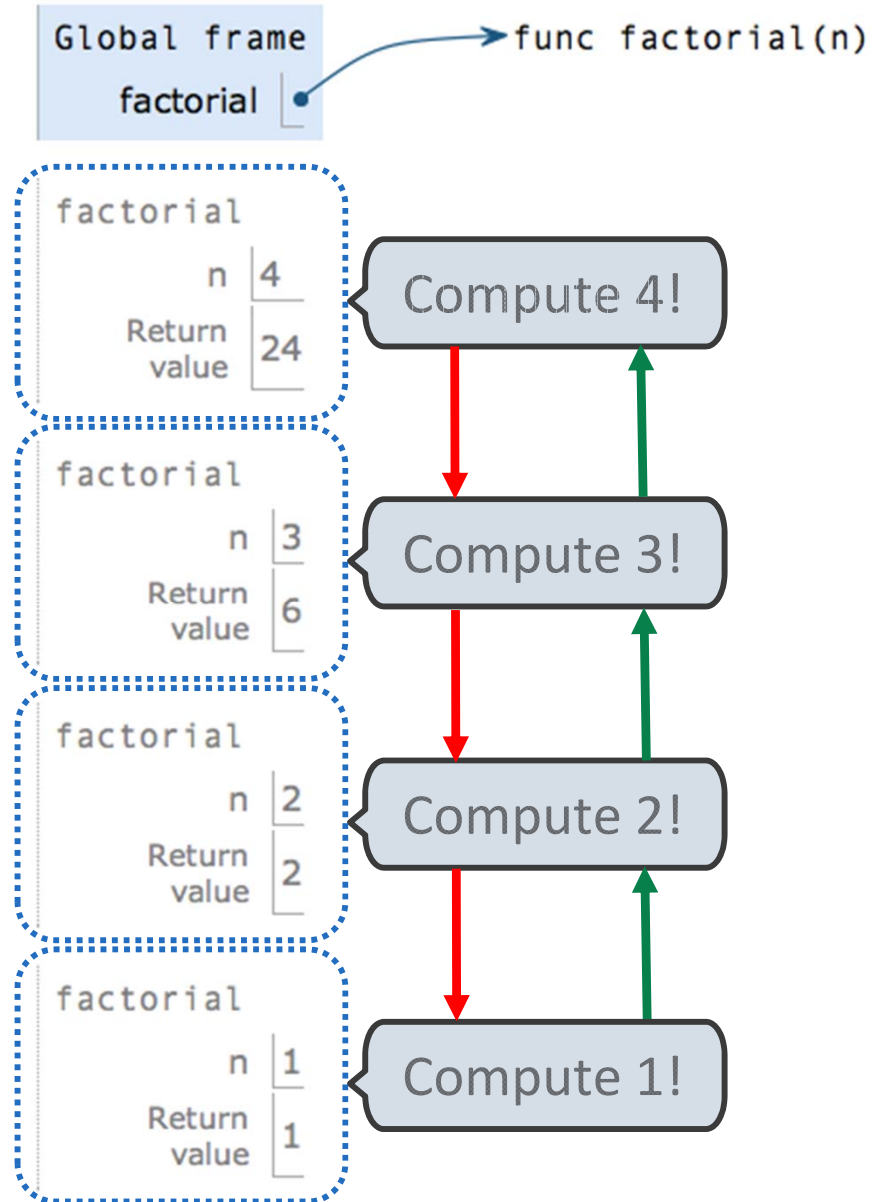
Example: <http://goo.gl/NjCKG>

Factorial Environment Diagram



Let's see what happens!

```
1 def factorial(n):  
2     if n == 0 or n == 1:  
3         return 1  
4     return n * factorial(n - 1)  
5  
6 factorial(4)
```



Example: <http://goo.gl/NjCKG>

Recursive Functions



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

Recursive Functions



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

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

Recursive Functions



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

Recursive functions have two important components:
ion itself, either directly or indirectly

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

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

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

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

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

Base case

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 Functions



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

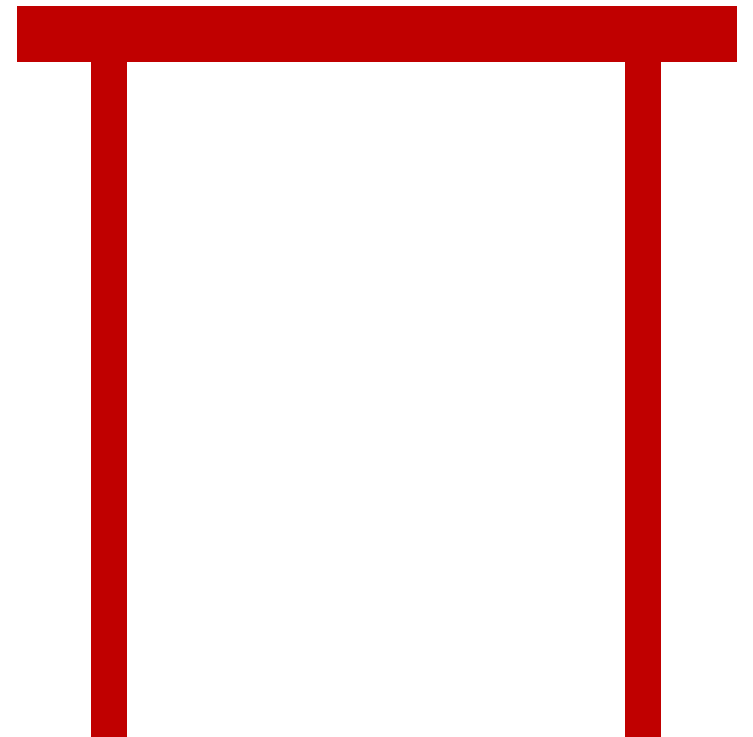
Recursive functions have two important components:

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)
```

A diagram illustrating the components of a recursive function. The code is shown in a monospaced font. The first part, `if n == 0 or n == 1:` followed by `return 1`, is enclosed in a blue dashed box. A callout bubble labeled "Base case" points to this box. The second part, `return n * factorial(n - 1)`, is also enclosed in a blue dashed box. A callout bubble labeled "Recursive case" points to this box.

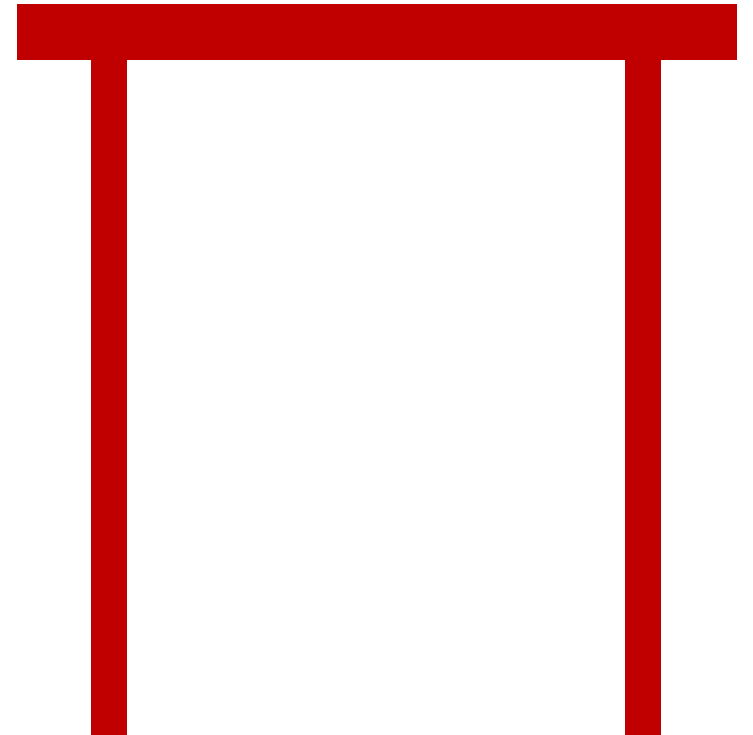
Recursion Example: Heavy Box



Recursion Example: Heavy Box



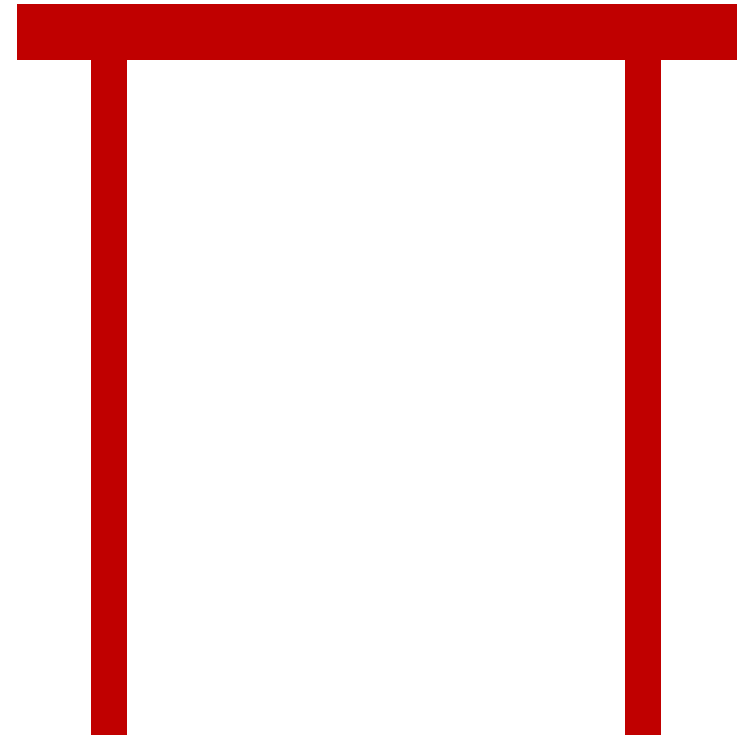
```
def lift_box(box):  
    if too_heavy(box):  
        book = remove_book(box)  
        lift_box(box)  
        add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



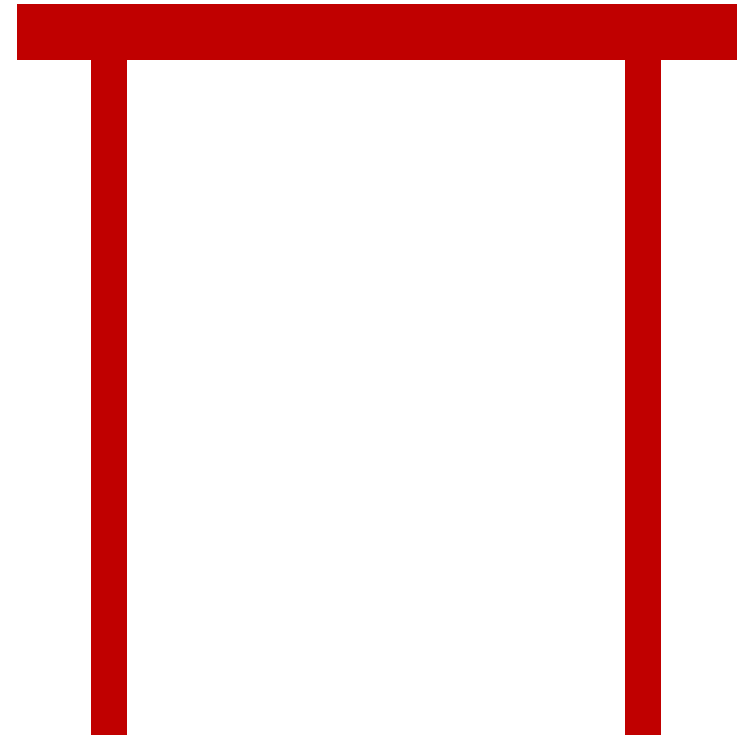
```
def lift_box(box):  
    → if too_heavy(box):  
        book = remove_book(box)  
        lift_box(box)  
        add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



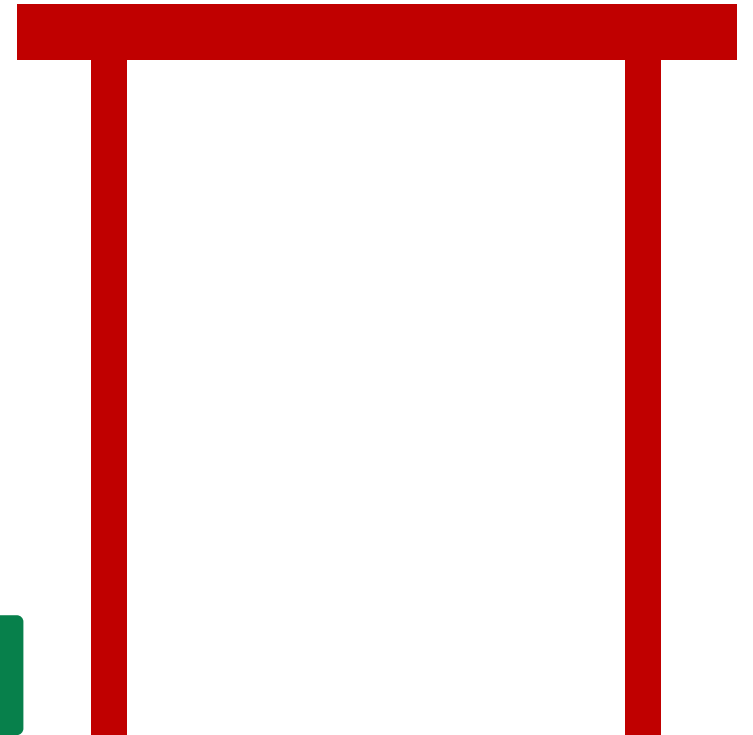
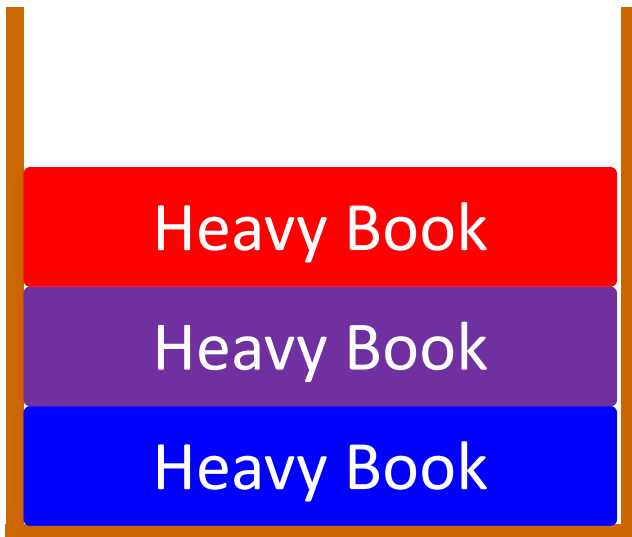
```
def lift_box(box):  
    if too_heavy(box):  
        ➔ book = remove_book(box)  
        lift_box(box)  
        add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



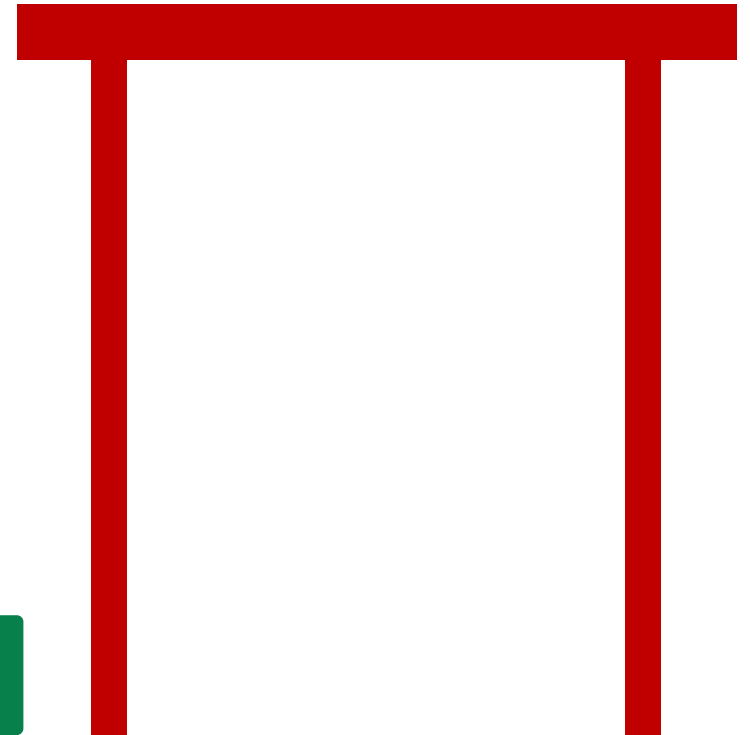
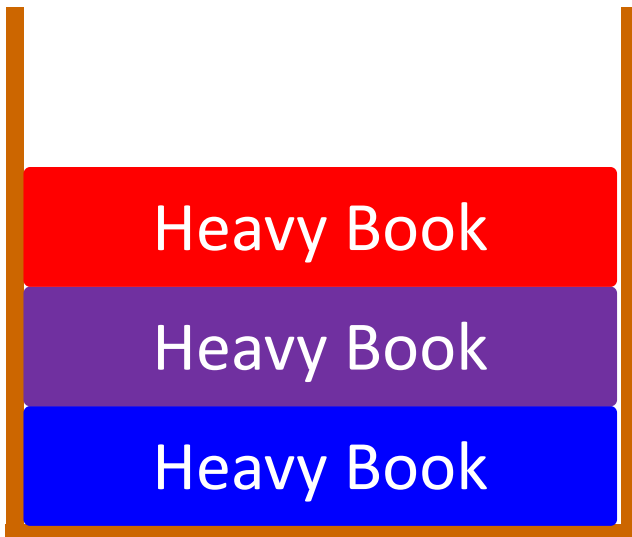
```
def lift_box(box):  
    if too_heavy(box):  
        book = remove_book(box)  
        → lift_box(box)  
        add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



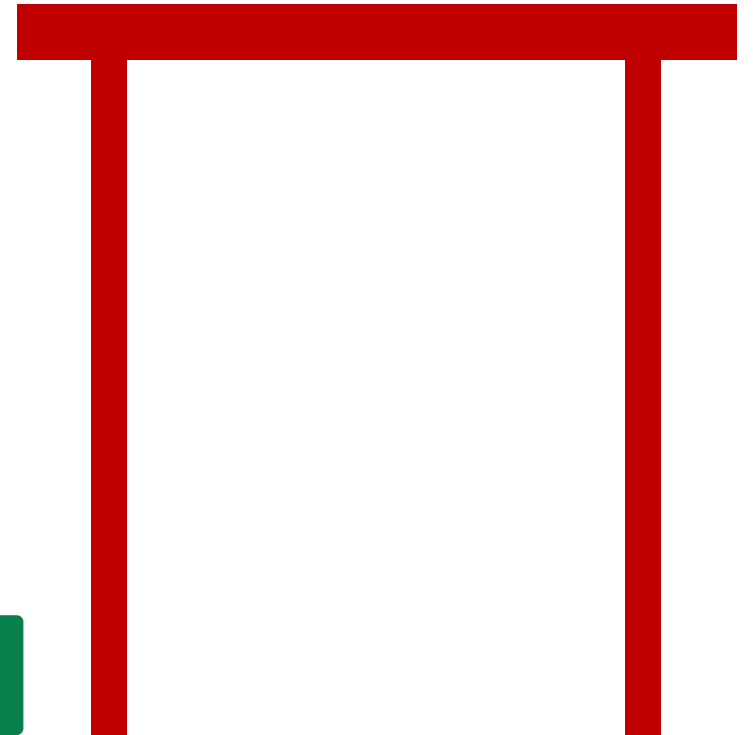
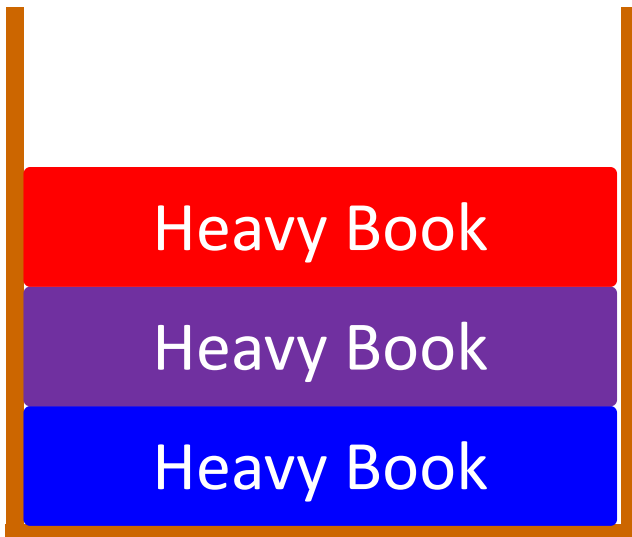
```
def lift_box(box):  
    → if too_heavy(box):  
        book = remove_book(box)  
        lift_box(box)  
        add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



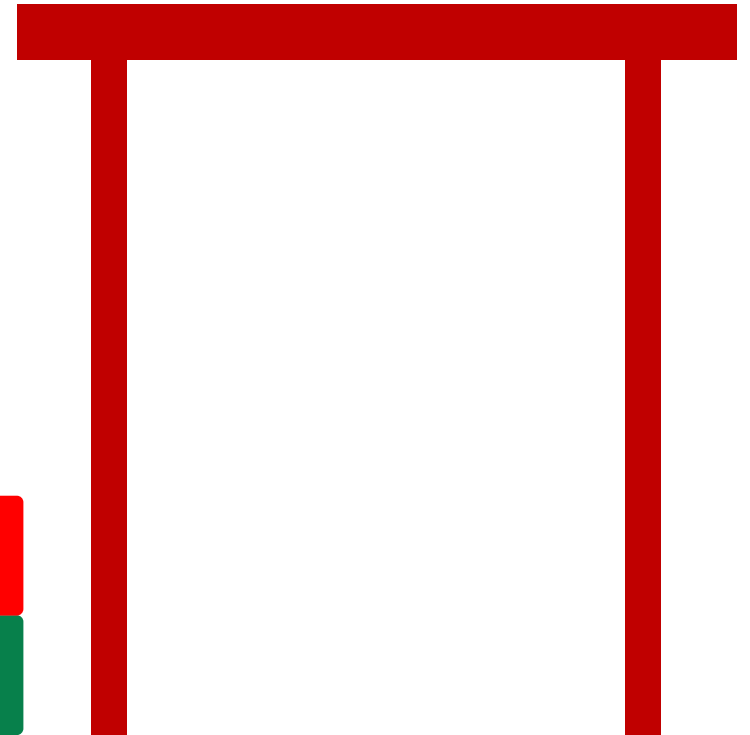
```
def lift_box(box):  
    if too_heavy(box):  
        → book = remove_book(box)  
           lift_box(box)  
           add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



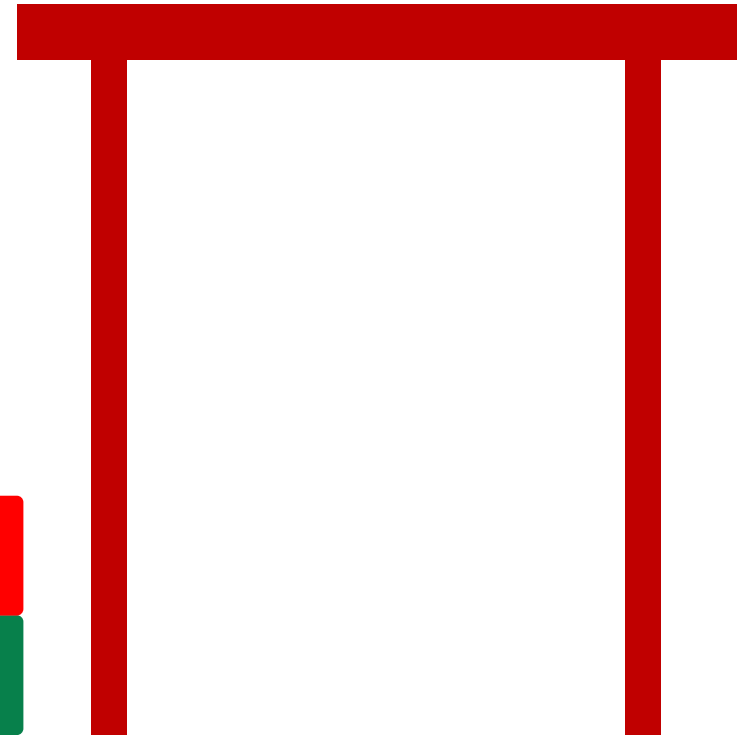
```
def lift_box(box):  
    if too_heavy(box):  
        book = remove_book(box)  
        → lift_box(box)  
        add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



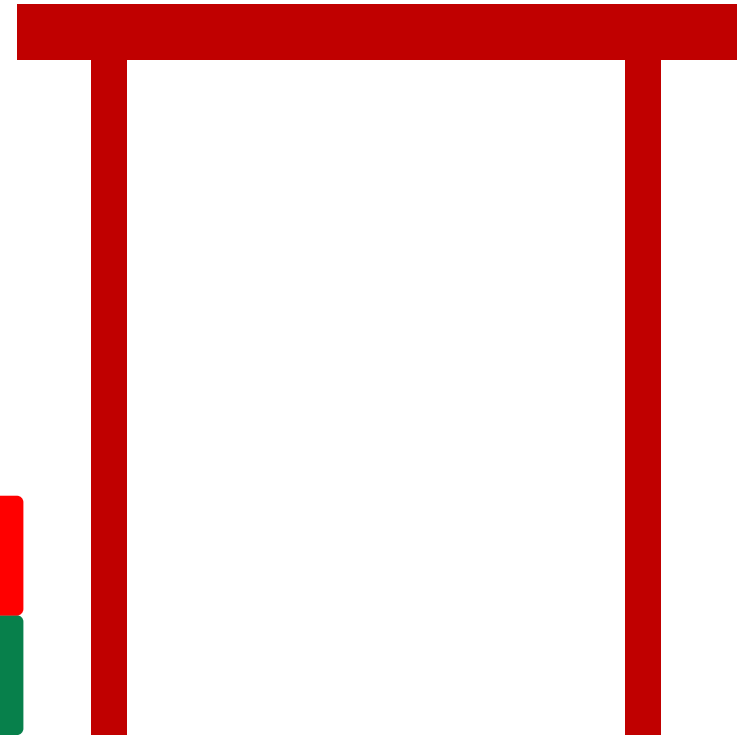
```
def lift_box(box):  
    → if too_heavy(box):  
        book = remove_book(box)  
        lift_box(box)  
        add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: 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: 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: 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: 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: 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: 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: Heavy Box



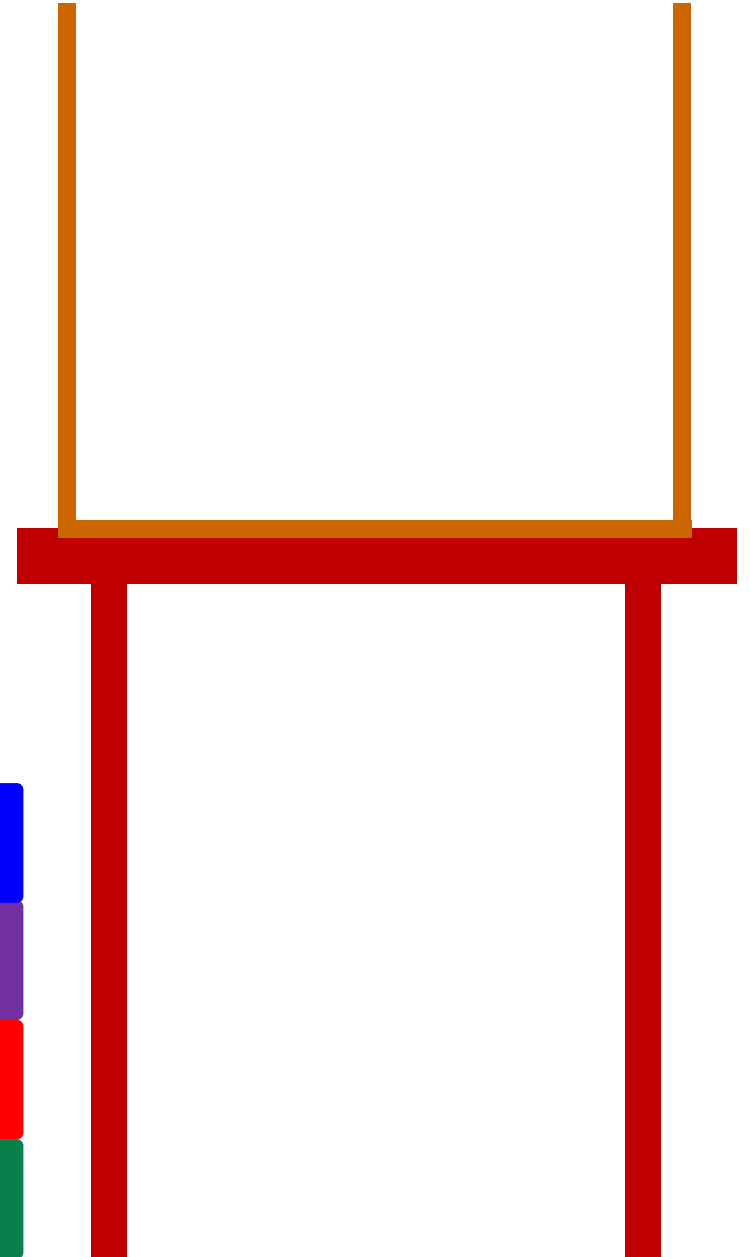
```
def lift_box(box):  
    if too_heavy(box):  
        book = remove_book(box)  
        lift_box(box)  
        add_book(box, book)  
    else:  
        → move_box(box)
```



Recursion Example: Heavy Box



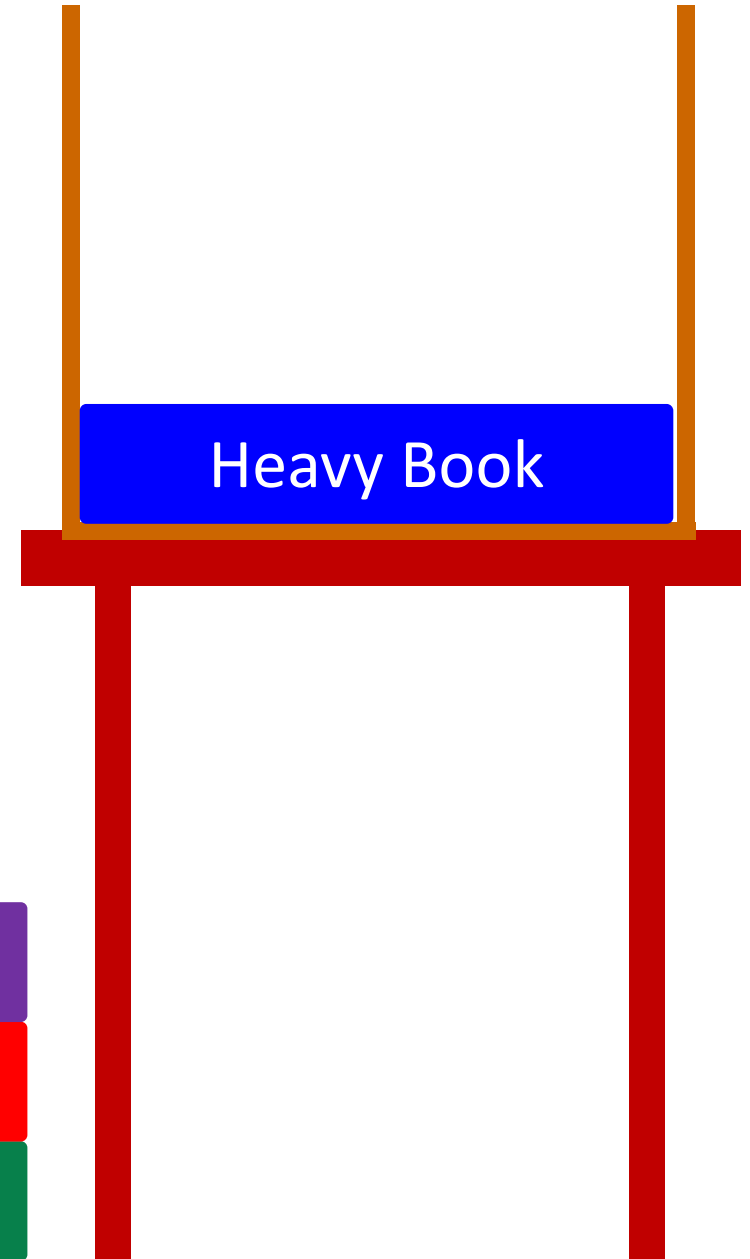
```
def lift_box(box):  
    if too_heavy(box):  
        book = remove_book(box)  
        lift_box(box)  
        → add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



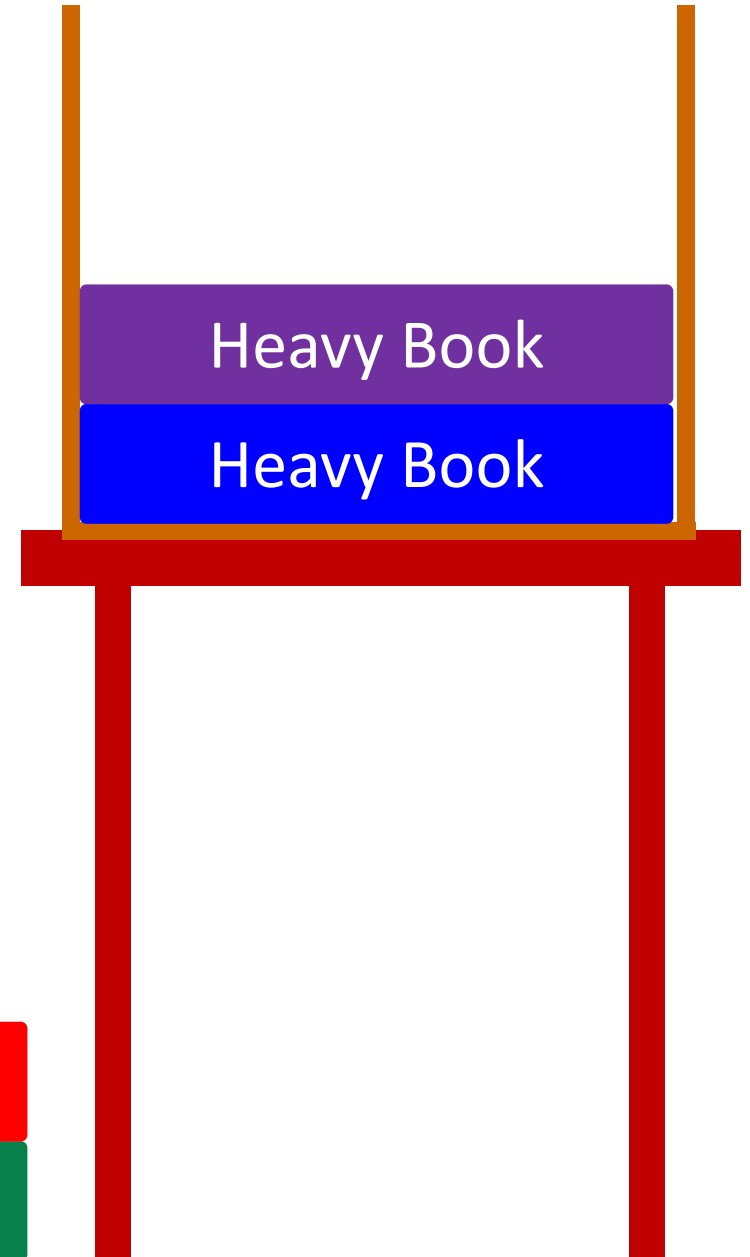
```
def lift_box(box):  
    if too_heavy(box):  
        book = remove_book(box)  
        lift_box(box)  
        → add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: Heavy Box



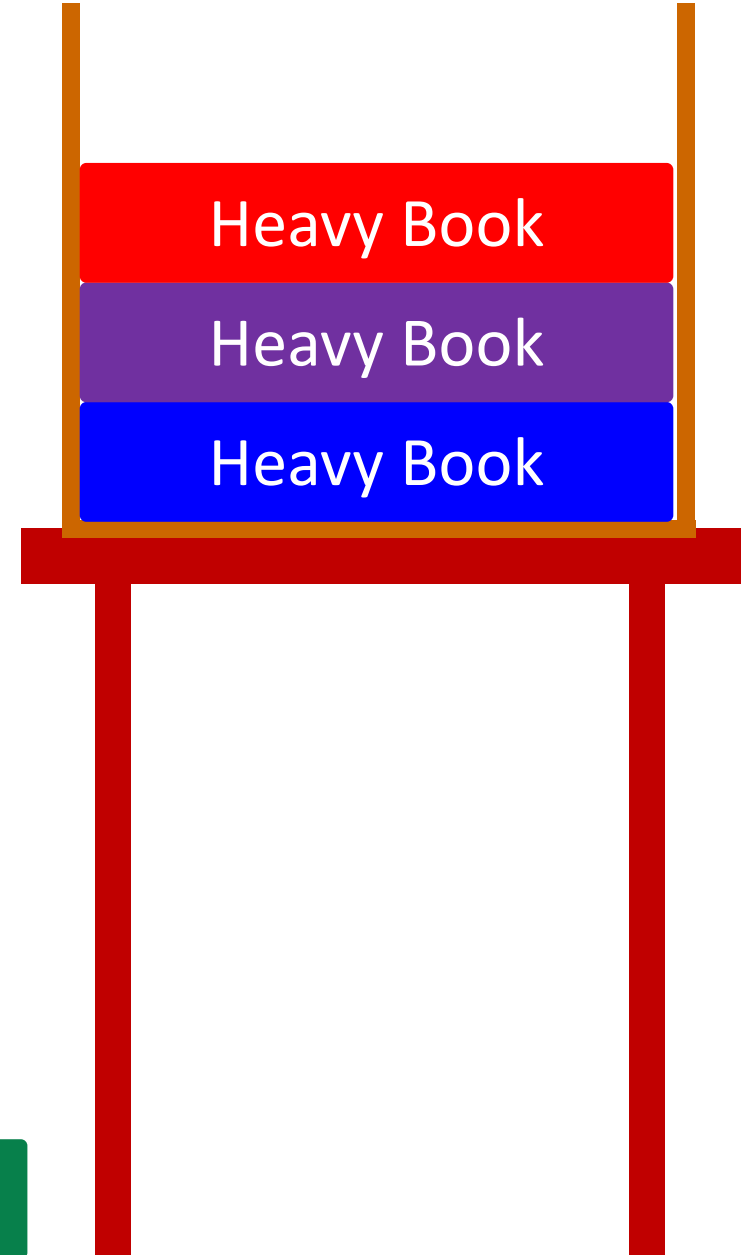
```
def lift_box(box):  
    if too_heavy(box):  
        book = remove_book(box)  
        lift_box(box)  
        → add_book(box, book)  
    else:  
        move_box(box)
```



Recursion Example: 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)
```

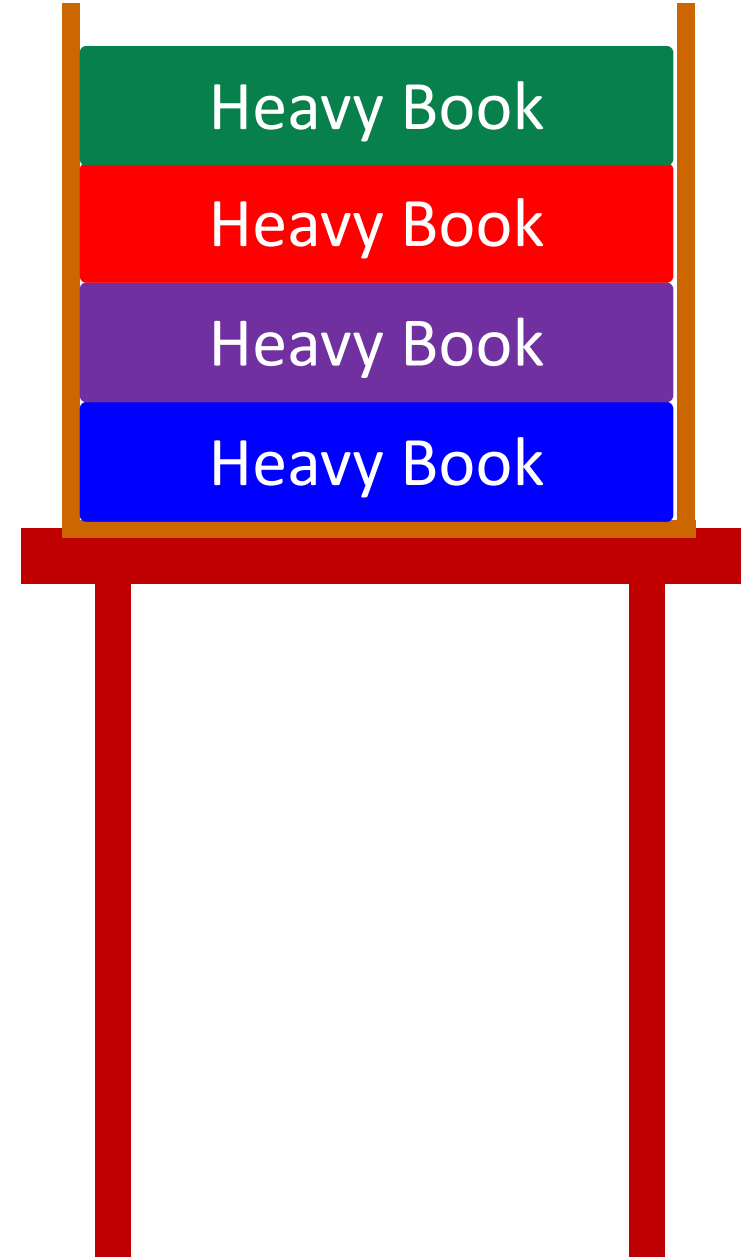


Heavy Book

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



Recursion Example: Duplication

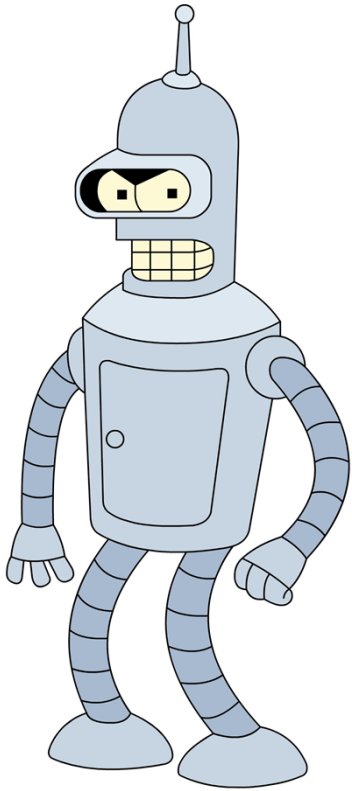


```
def duplicate(size):  
    return (duplicate(0.6 * size) +  
            duplicate(0.6 * size))
```

Recursion Example: Duplication



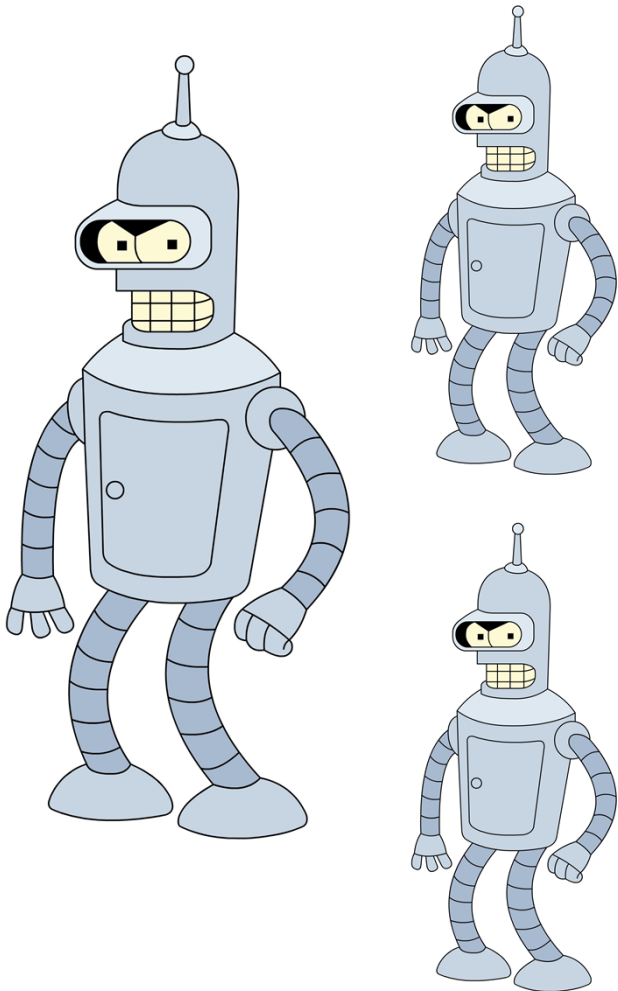
```
def duplicate(size):  
    return (duplicate(0.6 * size) +  
            duplicate(0.6 * size))
```



Recursion Example: Duplication



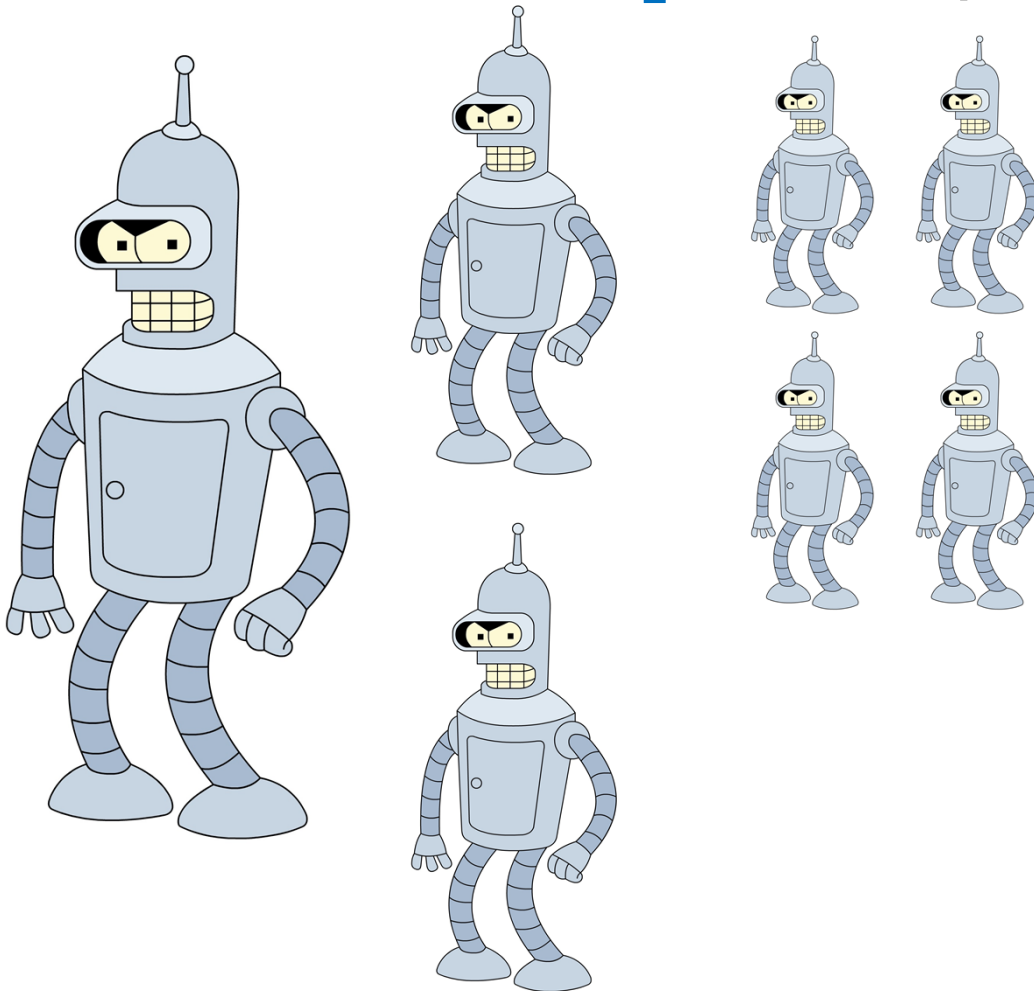
```
def duplicate(size):  
    return (duplicate(0.6 * size) +  
            duplicate(0.6 * size))
```



Recursion Example: Duplication



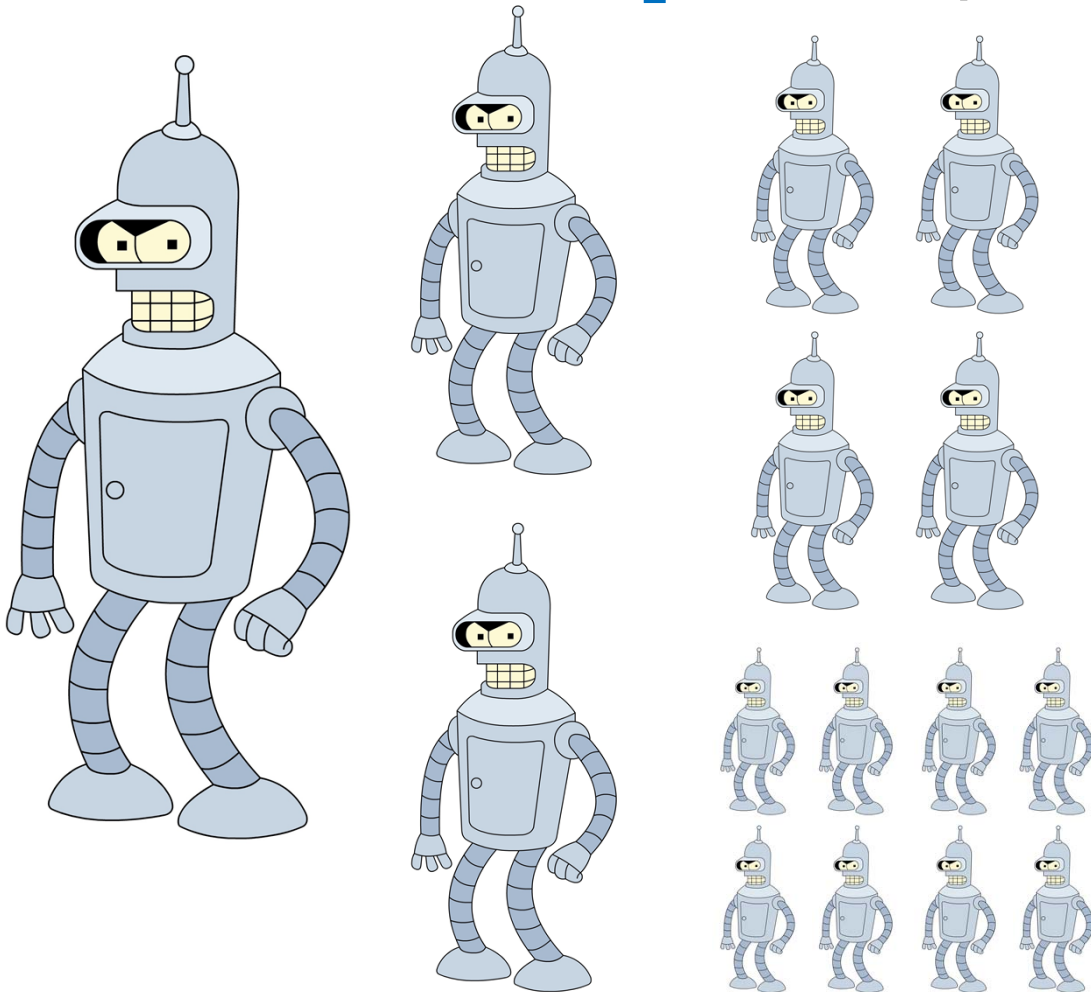
```
def duplicate(size):  
    return (duplicate(0.6 * size) +  
           duplicate(0.6 * size))
```



Recursion Example: Duplication



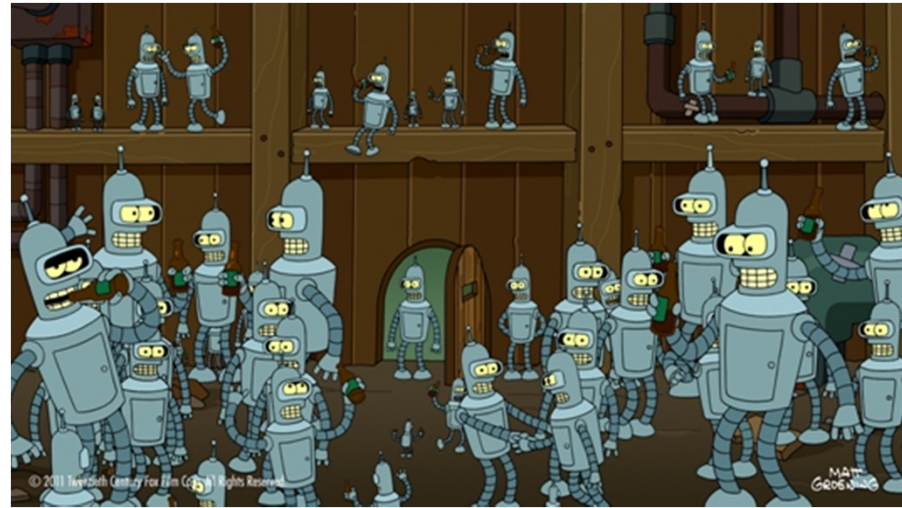
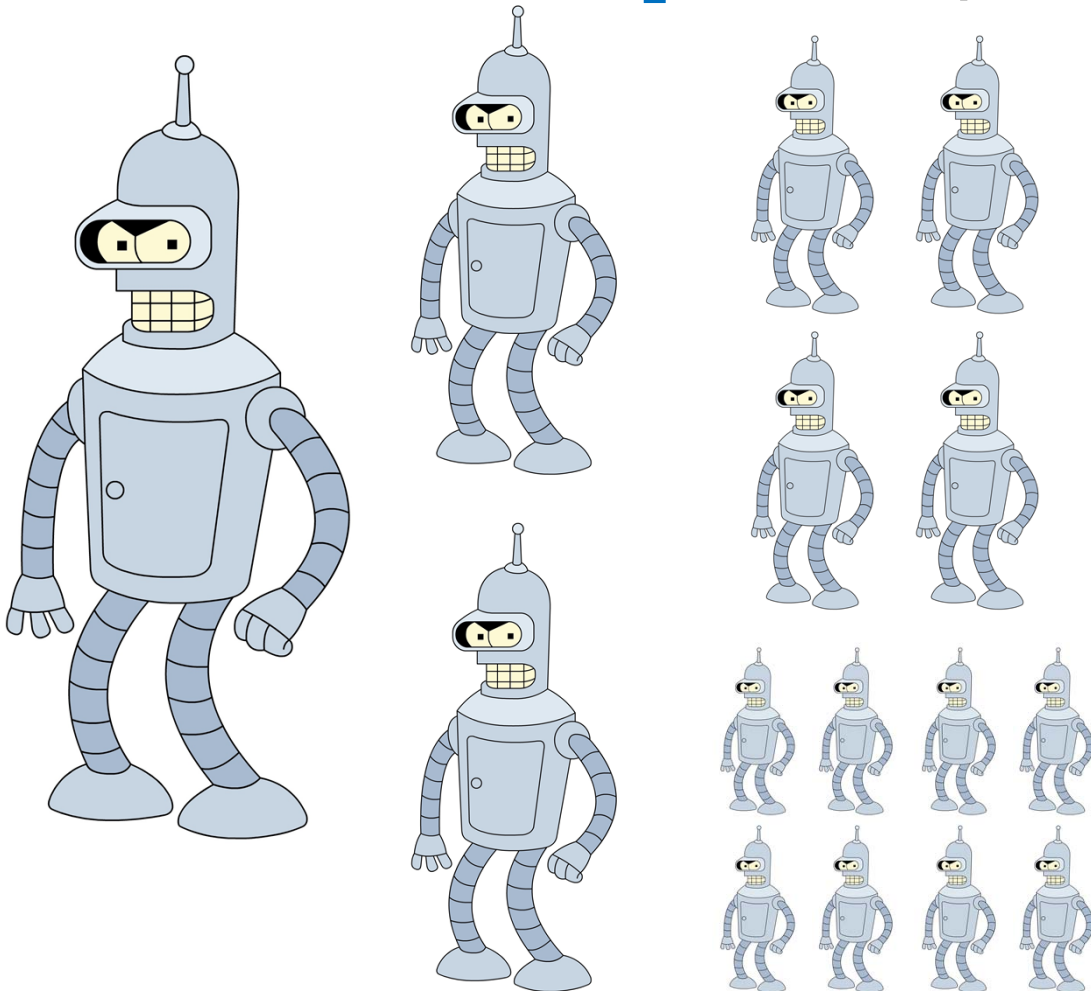
```
def duplicate(size):  
    return (duplicate(0.6 * size) +  
           duplicate(0.6 * size))
```



Recursion Example: Duplication



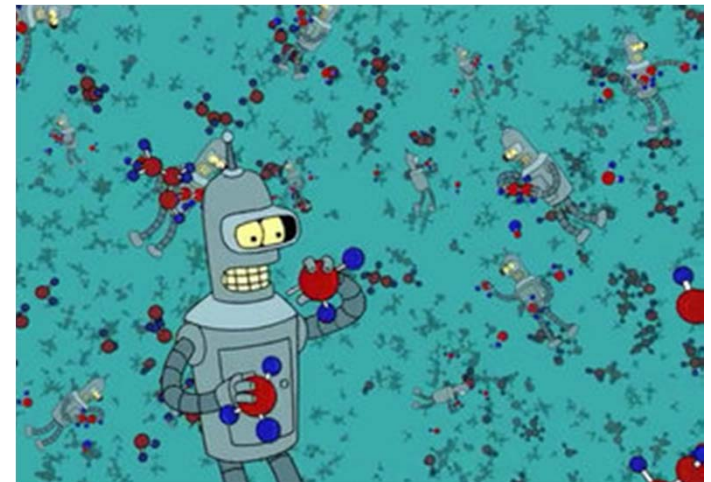
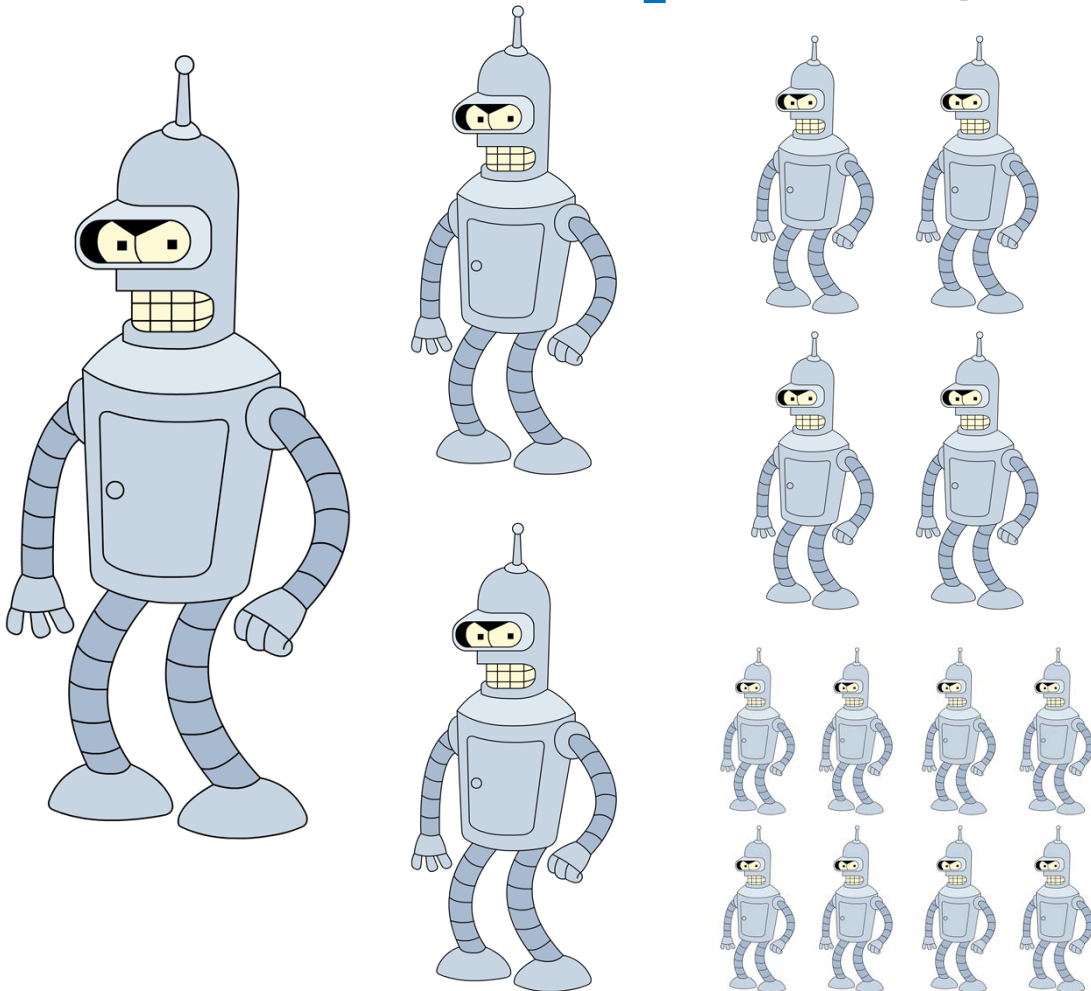
```
def duplicate(size):  
    return (duplicate(0.6 * size) +  
            duplicate(0.6 * size))
```



Recursion Example: Duplication



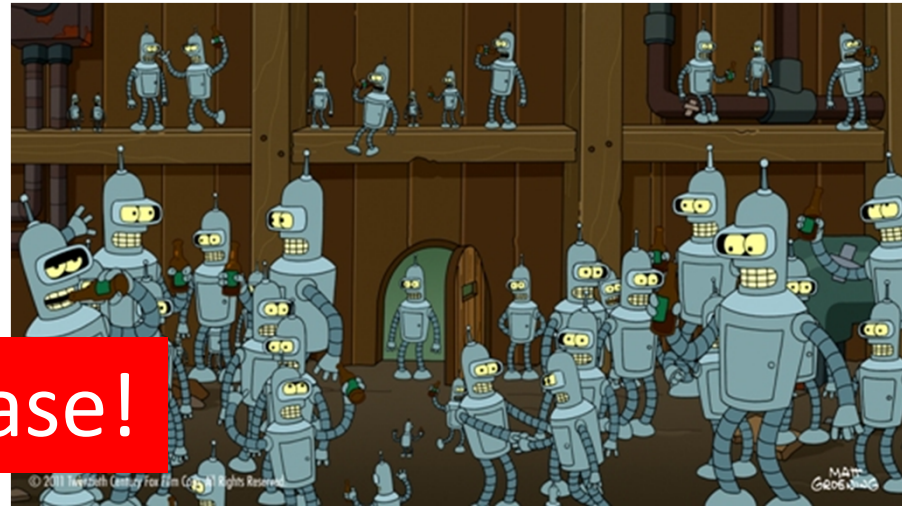
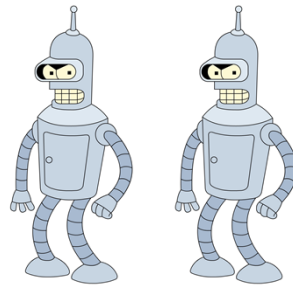
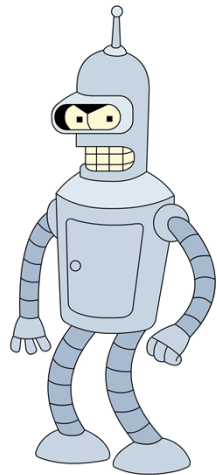
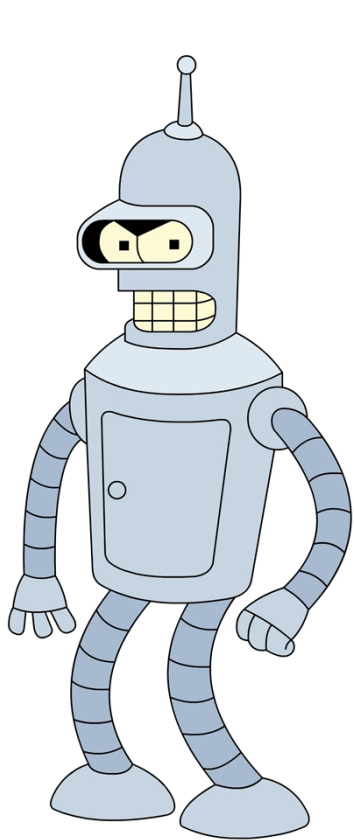
```
def duplicate(size):  
    return (duplicate(0.6 * size) +  
           duplicate(0.6 * size))
```



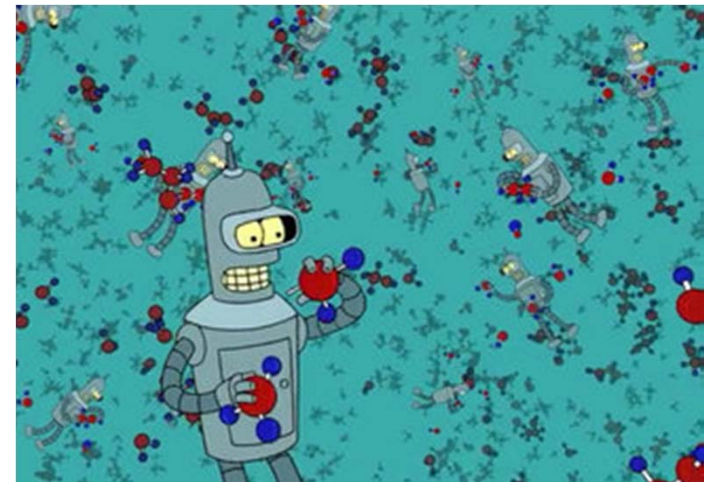
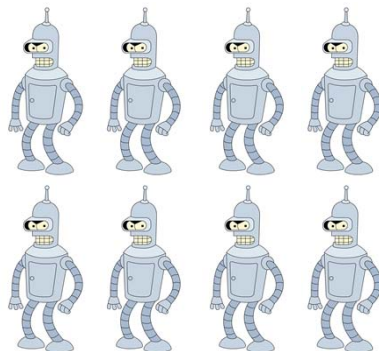
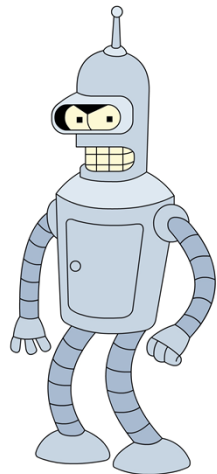
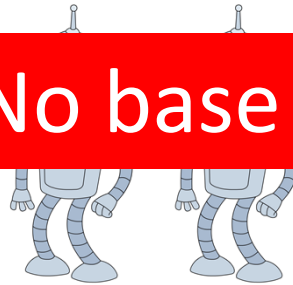
Recursion Example: Duplication



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



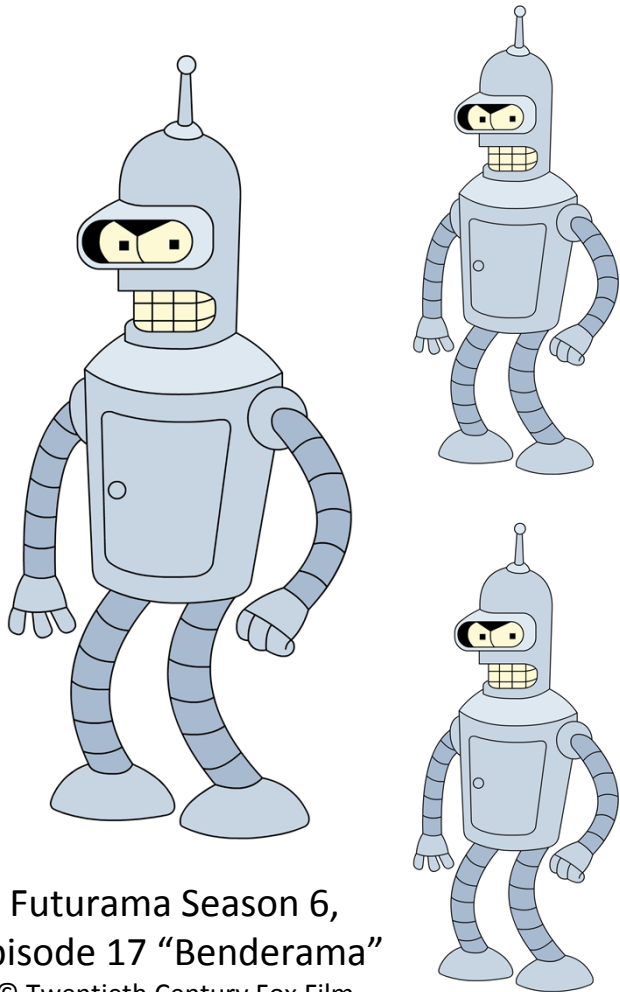
No base case!



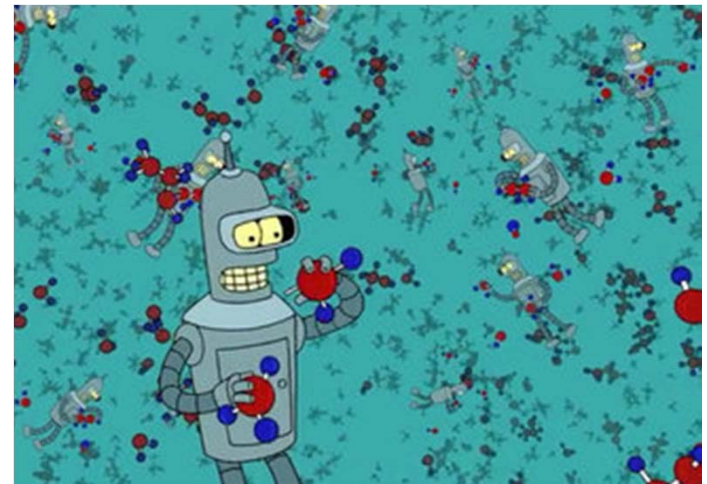
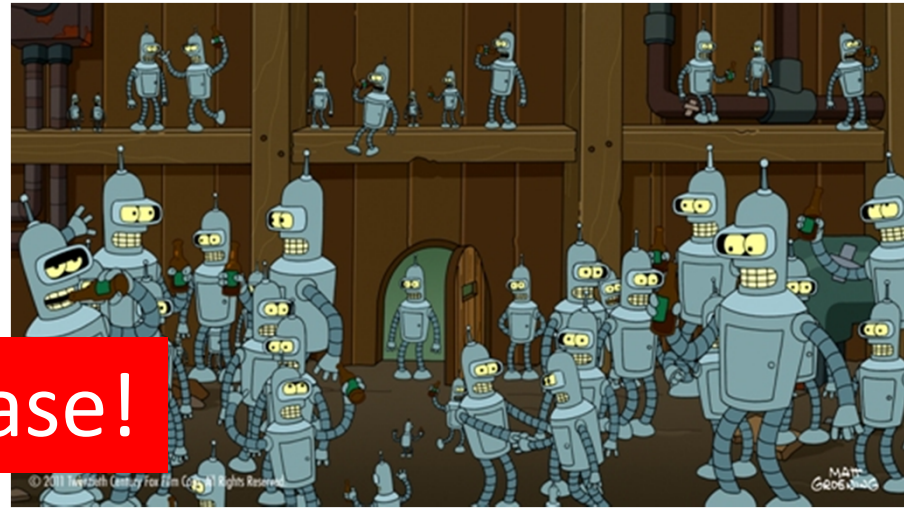
Recursion Example: Duplication



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



No base case!



Recursion Example: Dreaming



Recursion Example: Dreaming



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

Recursion Example: Dreaming



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

Recursion Example: Dreaming



Global frame



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

Recursion Example: Dreaming



func dream(level)

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

Recursion Example: Dreaming



Global frame

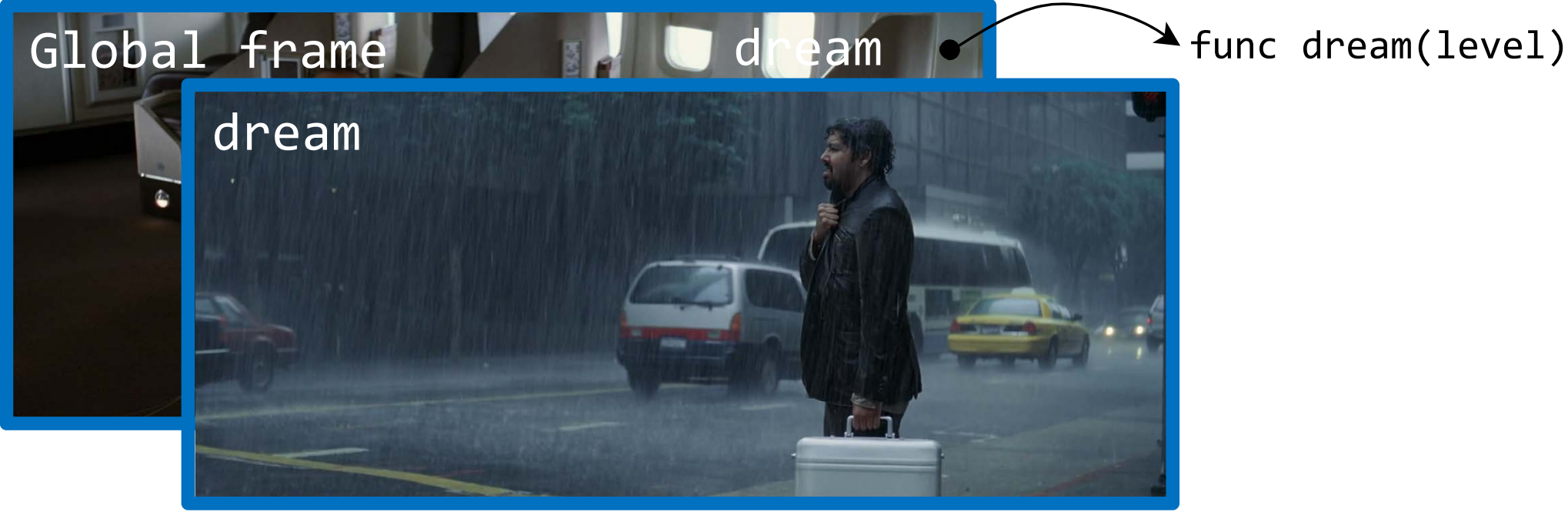
dream

func dream(level)



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

Recursion Example: Dreaming



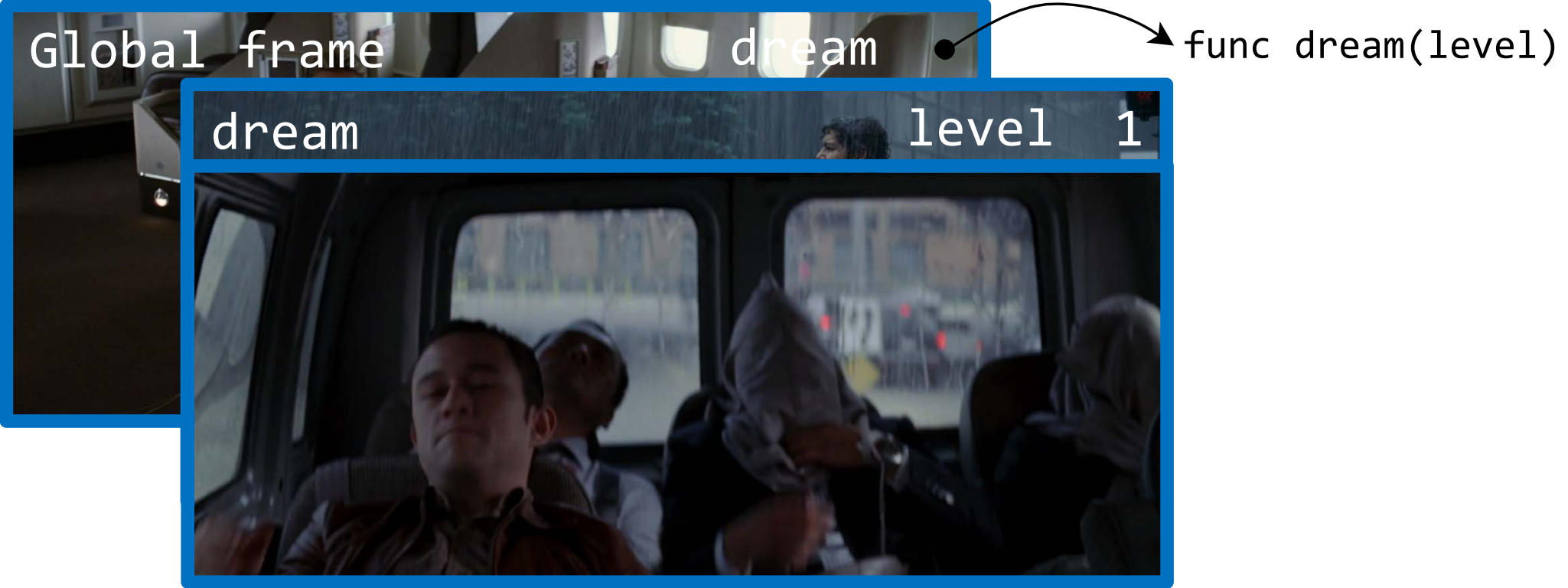
```
def dream(level):  
    if level == 3:  
        return inception()  
    else:  
        return dream(level + 1)
```

Recursion Example: Dreaming



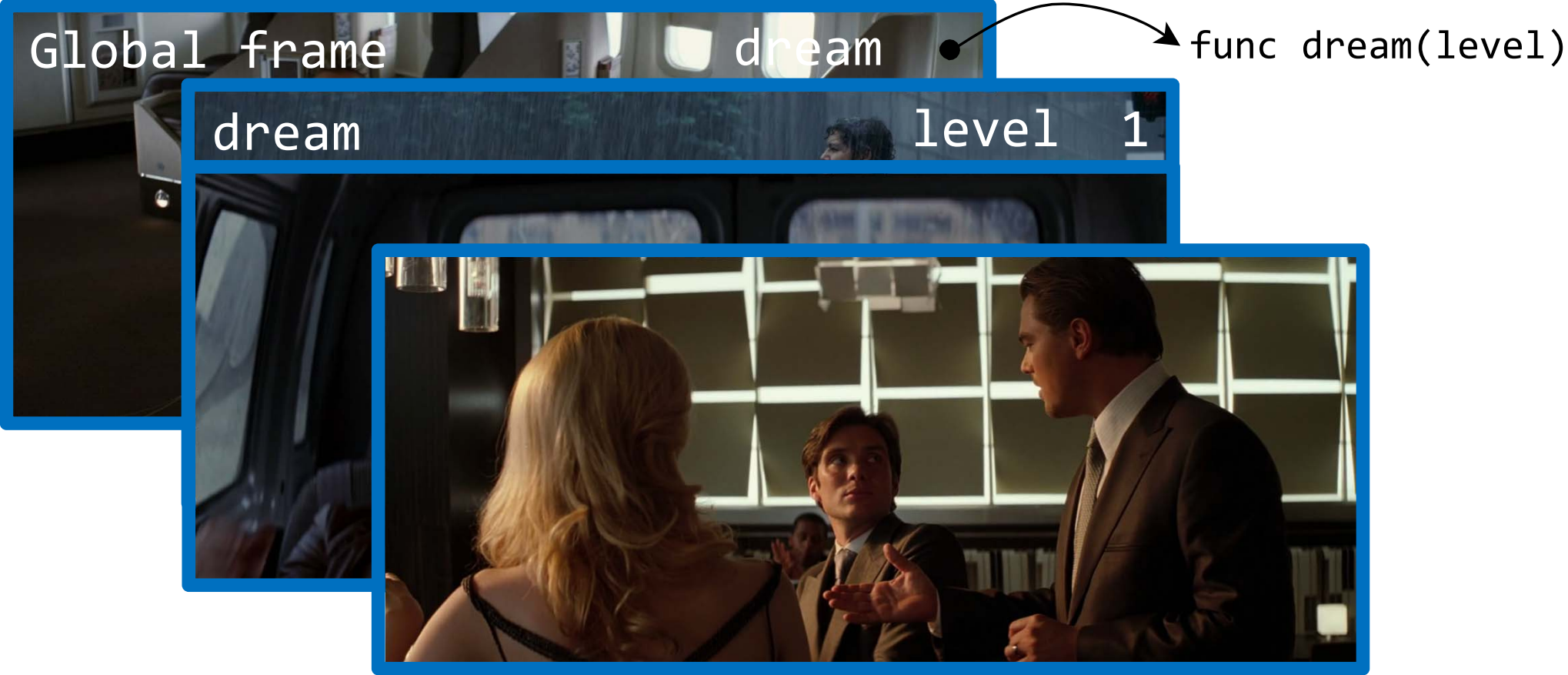
```
def dream(level):  
    if level == 3:  
        return inception()  
    else:  
        return dream(level + 1)
```

Recursion Example: Dreaming



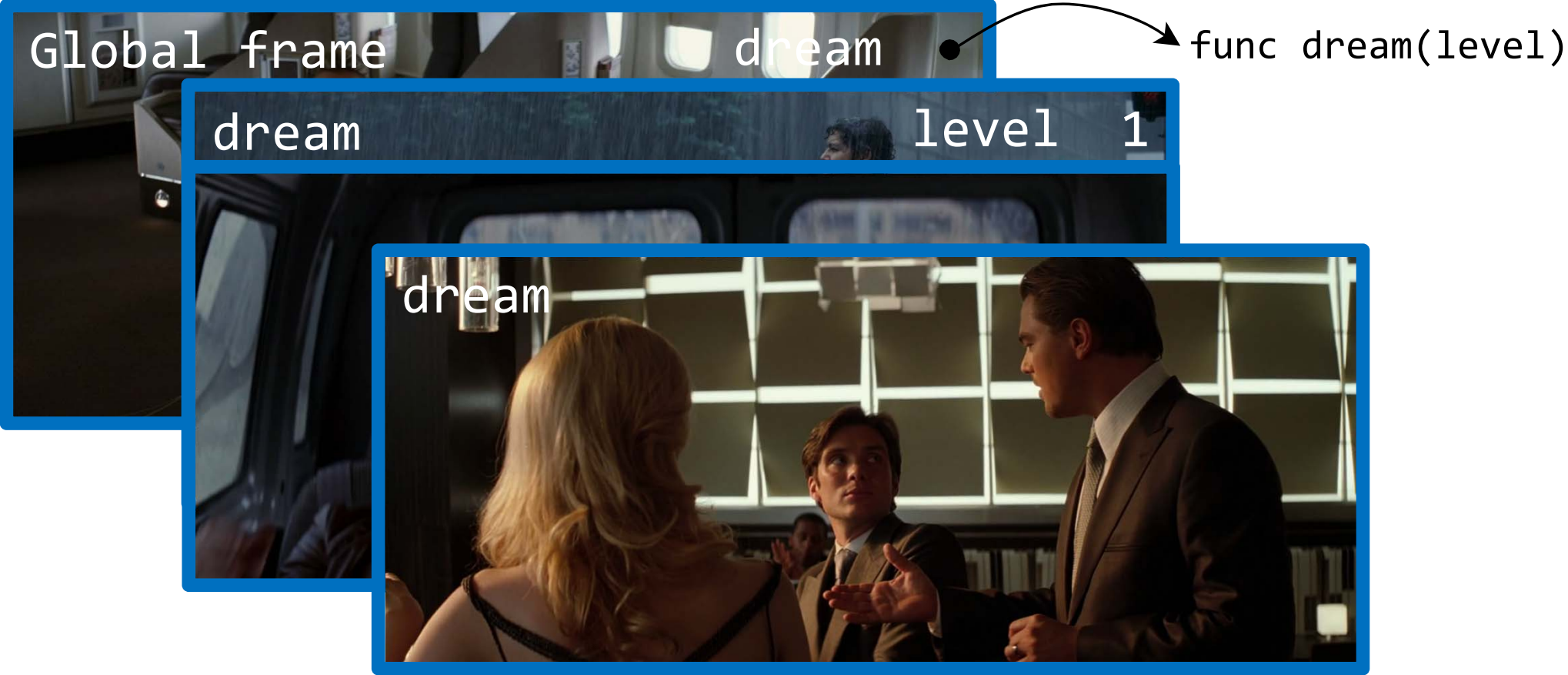
```
def dream(level):  
    if level == 3:  
        return inception()  
    else:  
        return dream(level + 1)
```


Recursion Example: Dreaming



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

Recursion Example: Dreaming



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

Recursion Example: Dreaming



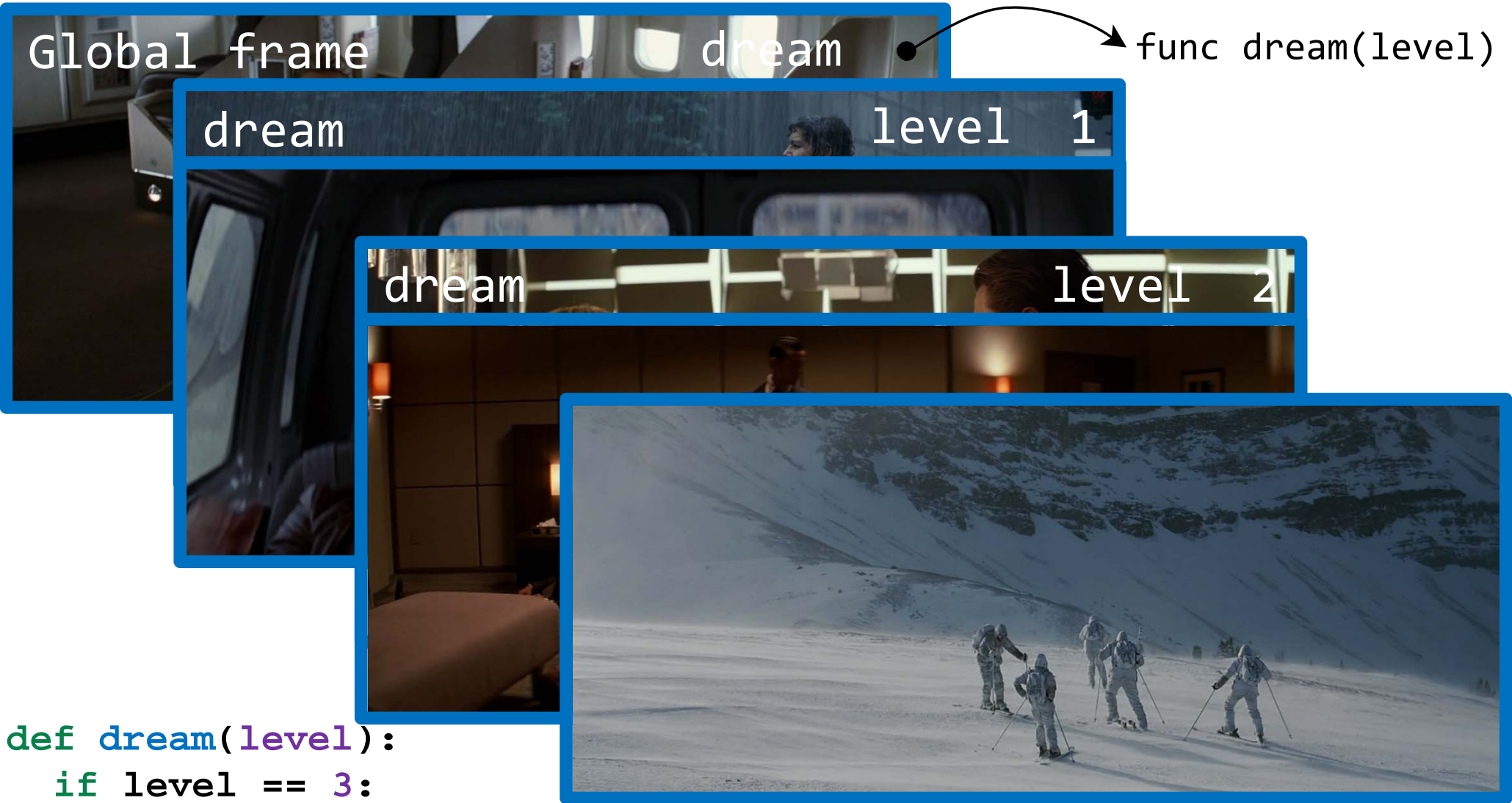
```
def dream(level):  
    if level == 3:  
        return inception()  
    else:  
        return dream(level + 1)
```

Recursion Example: Dreaming



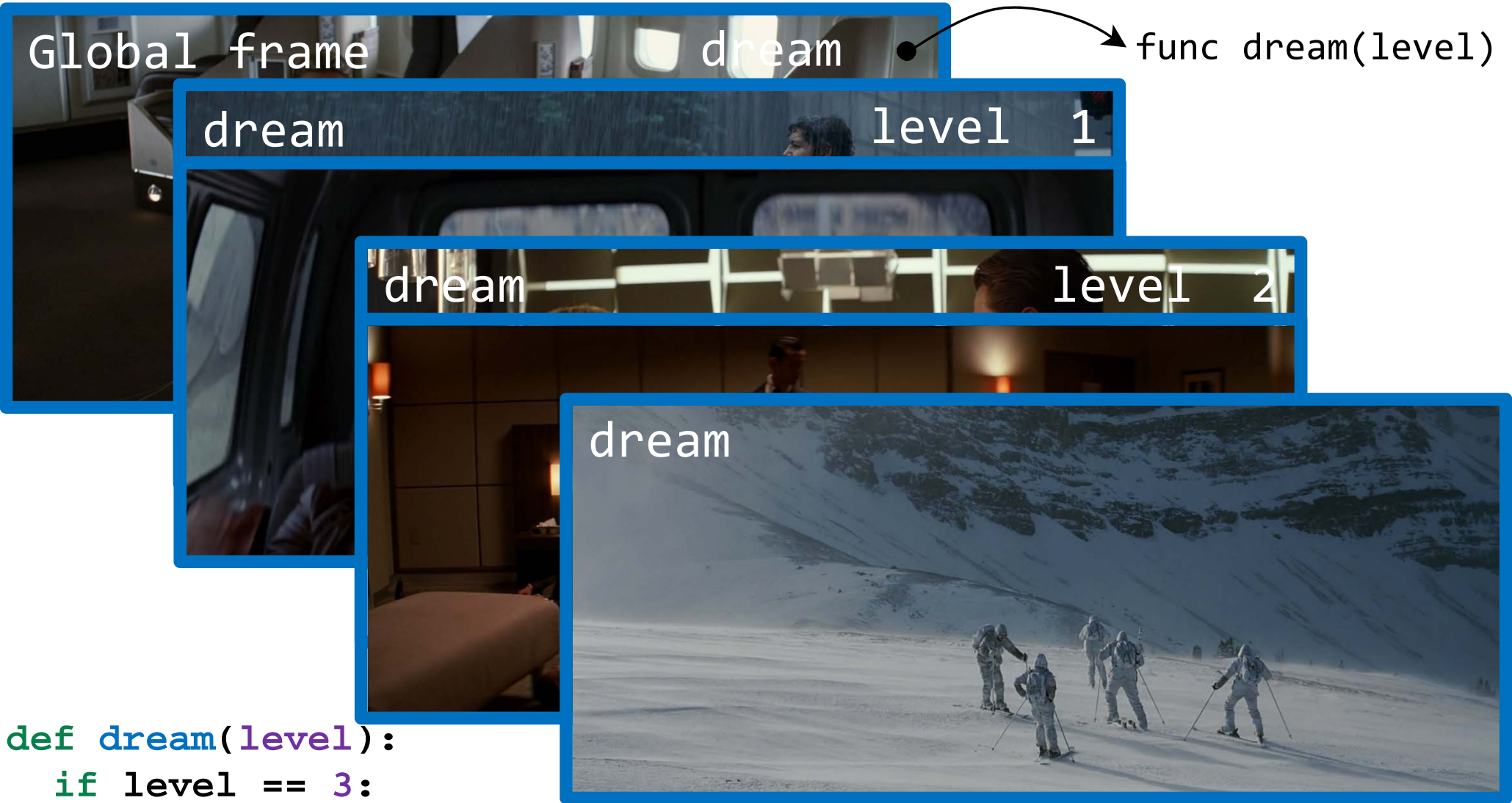
```
def dream(level):  
    if level == 3:  
        return inception()  
    else:  
        return dream(level + 1)
```

Recursion Example: Dreaming



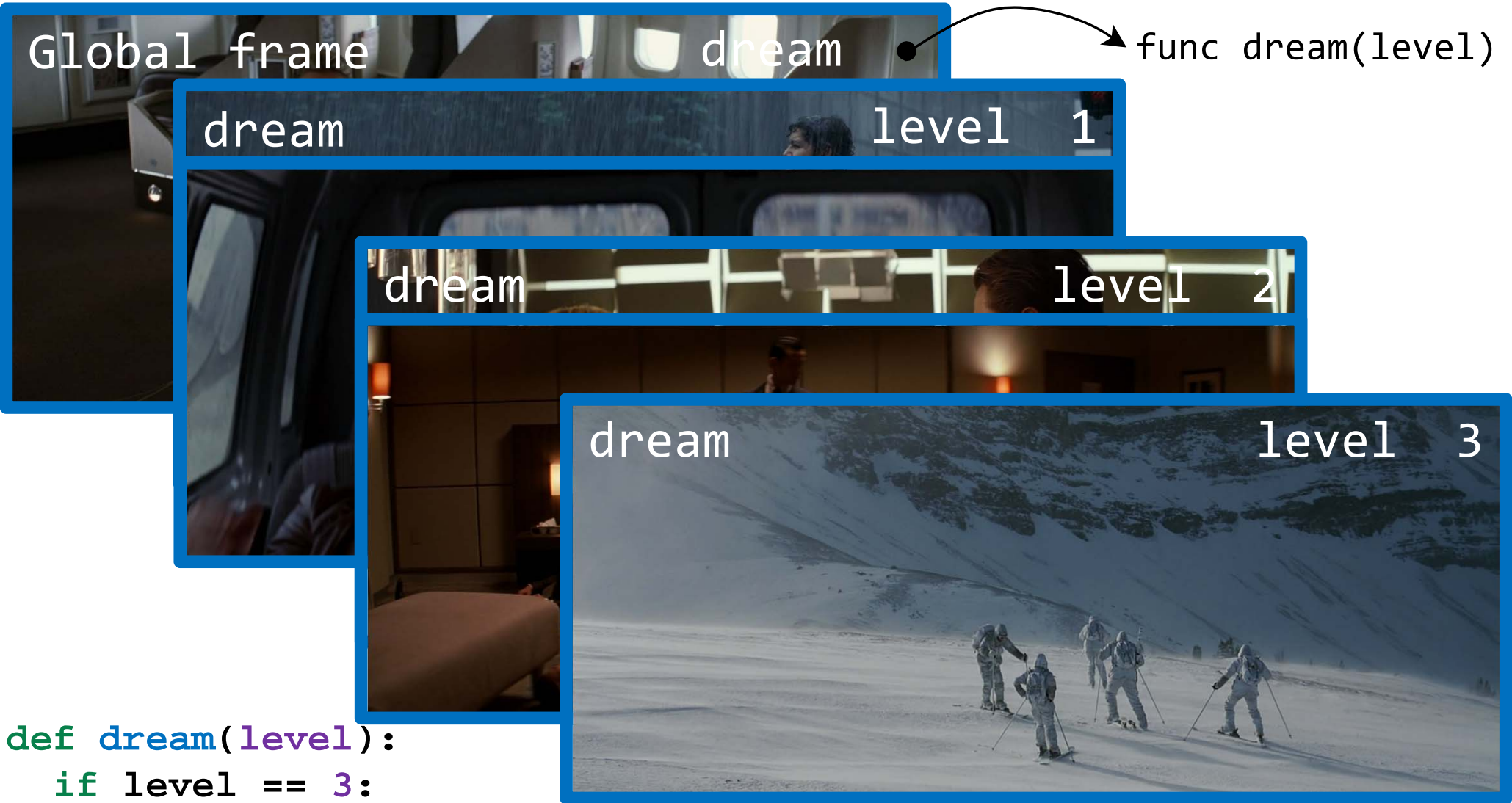
```
def dream(level):  
    if level == 3:  
        return inception()  
    else:  
        return dream(level + 1)
```

Recursion Example: Dreaming



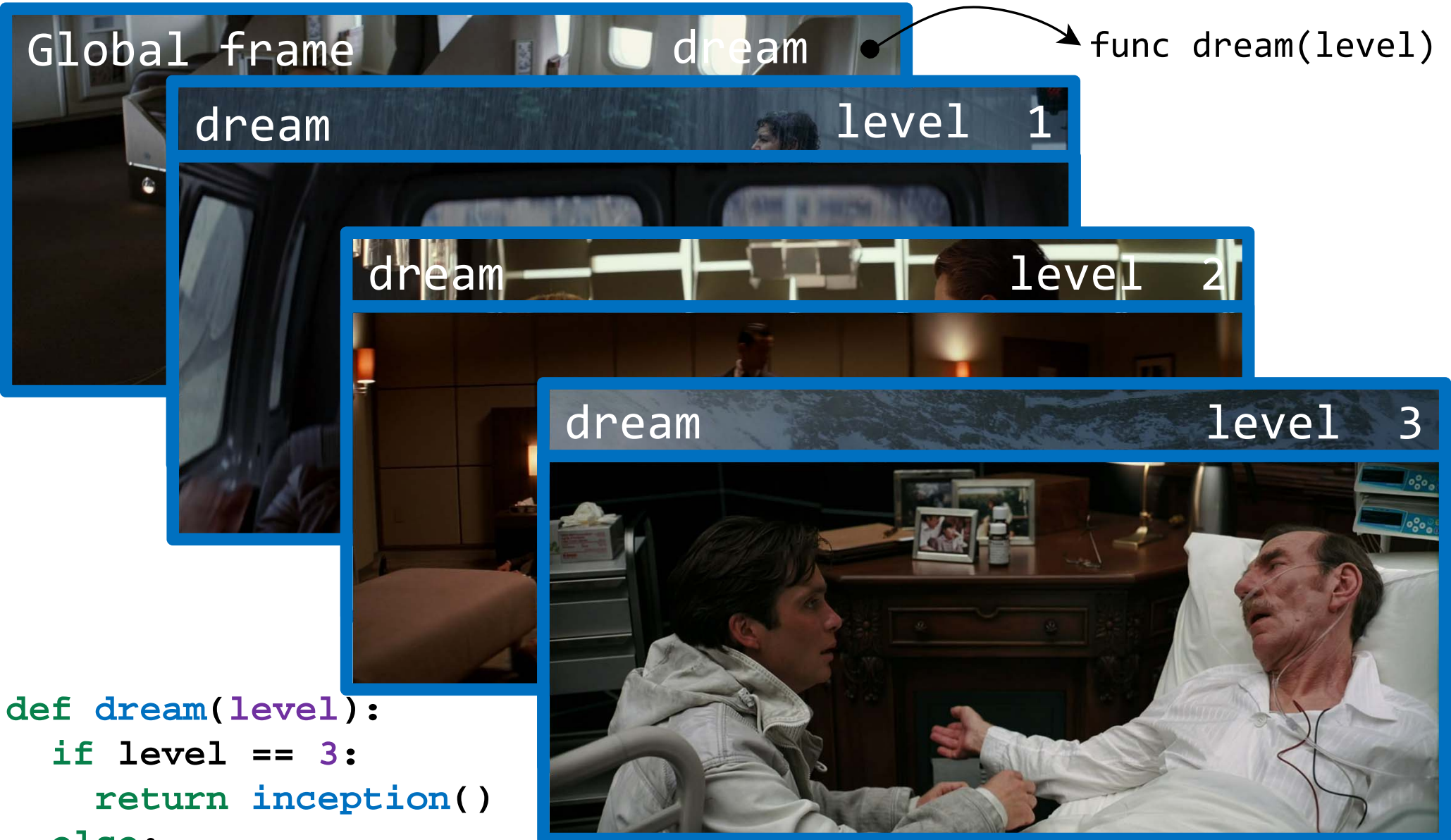
```
def dream(level):  
    if level == 3:  
        return inception()  
    else:  
        return dream(level + 1)
```

Recursion Example: Dreaming



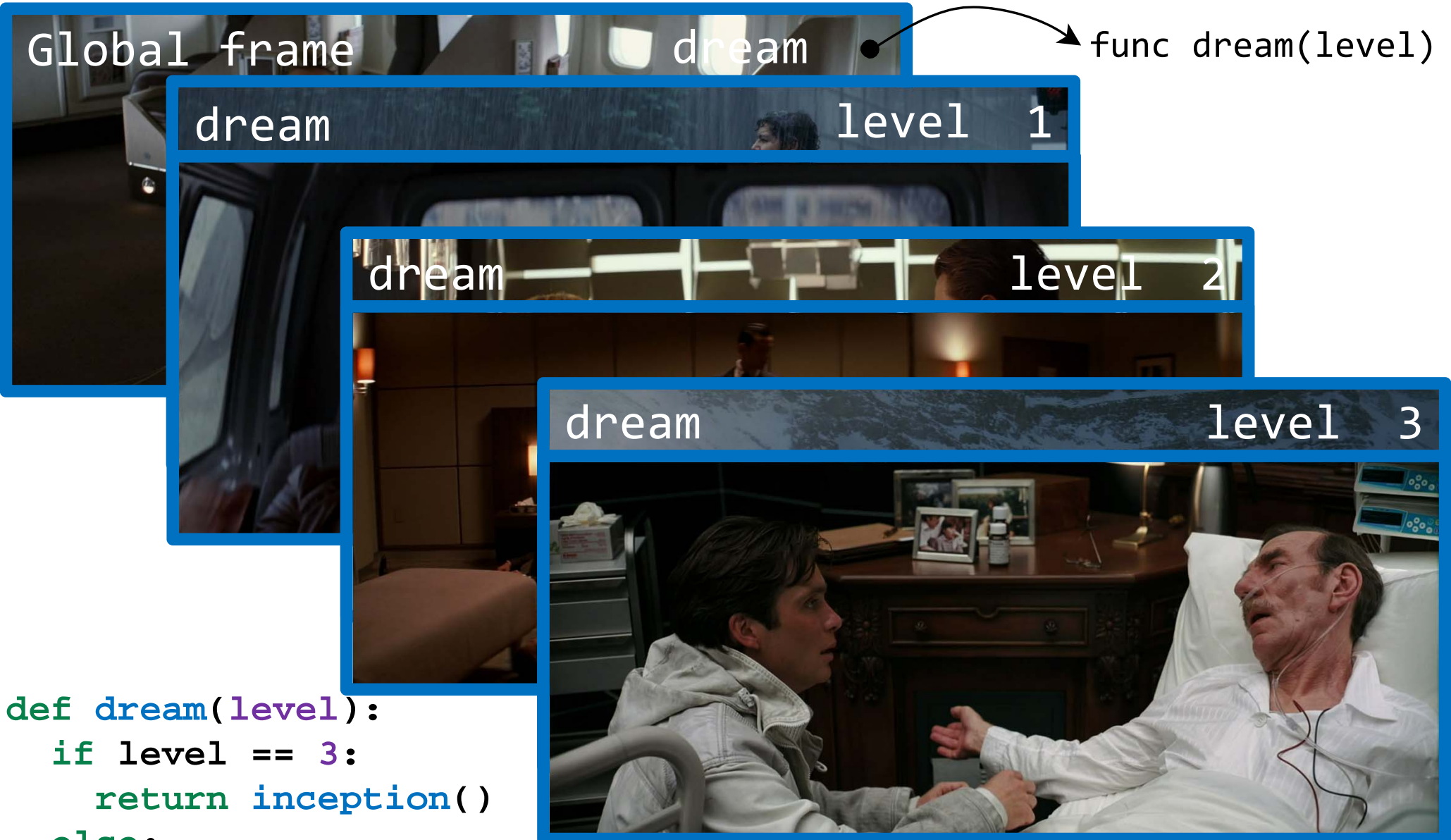
```
def dream(level):  
    if level == 3:  
        return inception()  
    else:  
        return dream(level + 1)
```

Recursion Example: Dreaming



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


Recursion Example: Dreaming



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



Reversing the Order of Recursive Calls



Reversing the Order of Recursive Calls



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

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.

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.

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

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.

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

```
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



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

Reverse Environment Diagram



Here is how the reversed
computation evolves

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

Reverse Environment Diagram



Here is how the reversed computation evolves

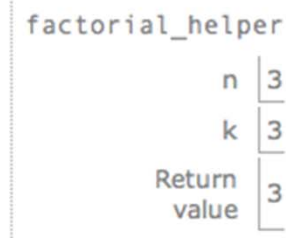
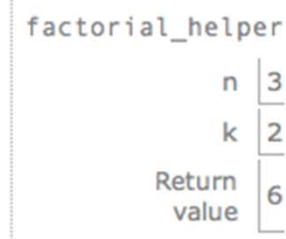
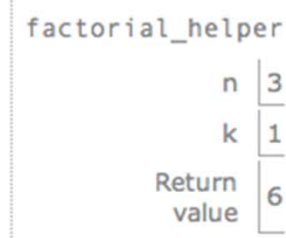
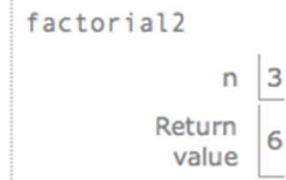
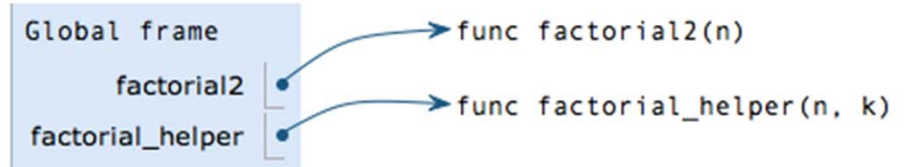
```
1 def factorial2(n):
2     return factorial_helper(n, 1)
→ 3
4 def factorial_helper(n, k):
5     if k >= n:
6         return k
7     return k * factorial_helper(n, k + 1)
8
↪ 9 factorial2(3)
```

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



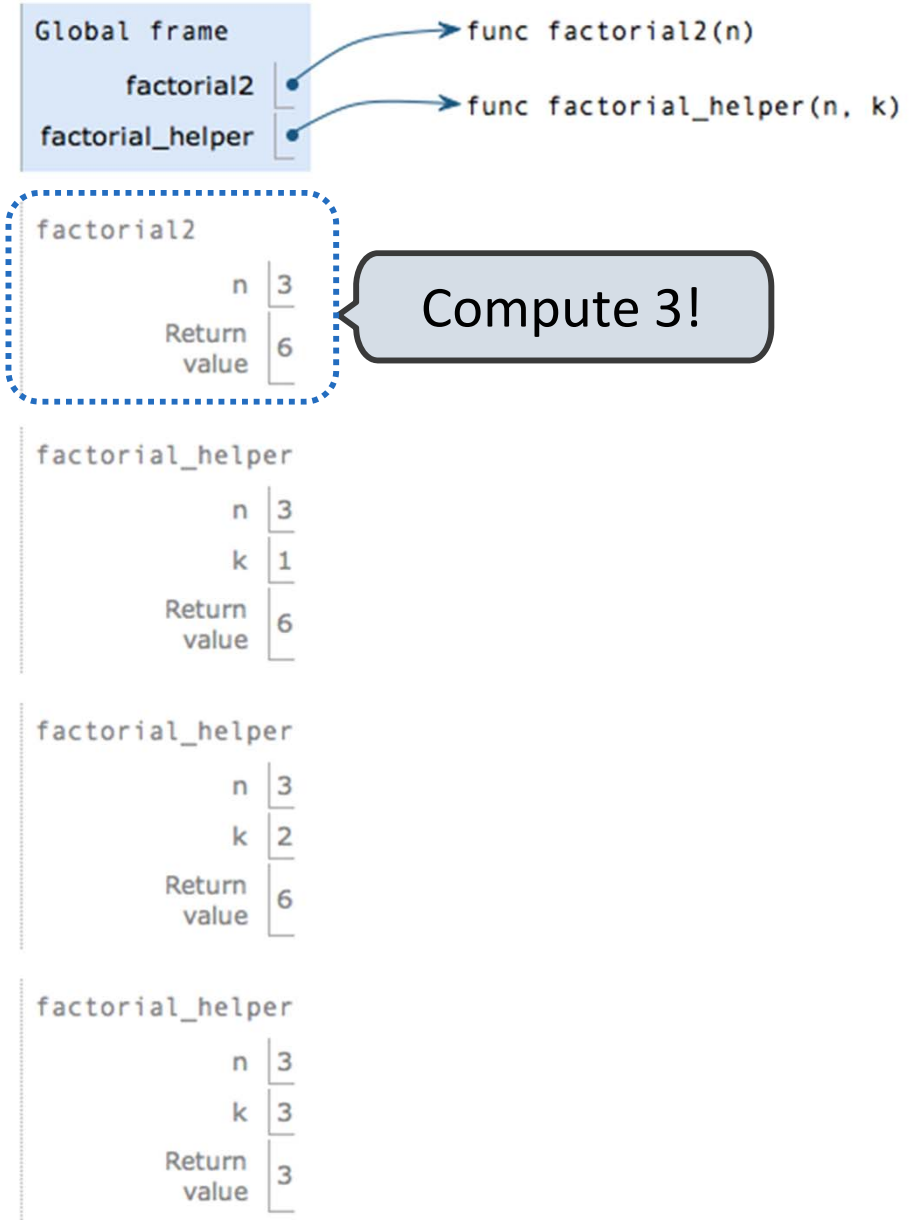
Example: <http://goo.gl/6zz0z>

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



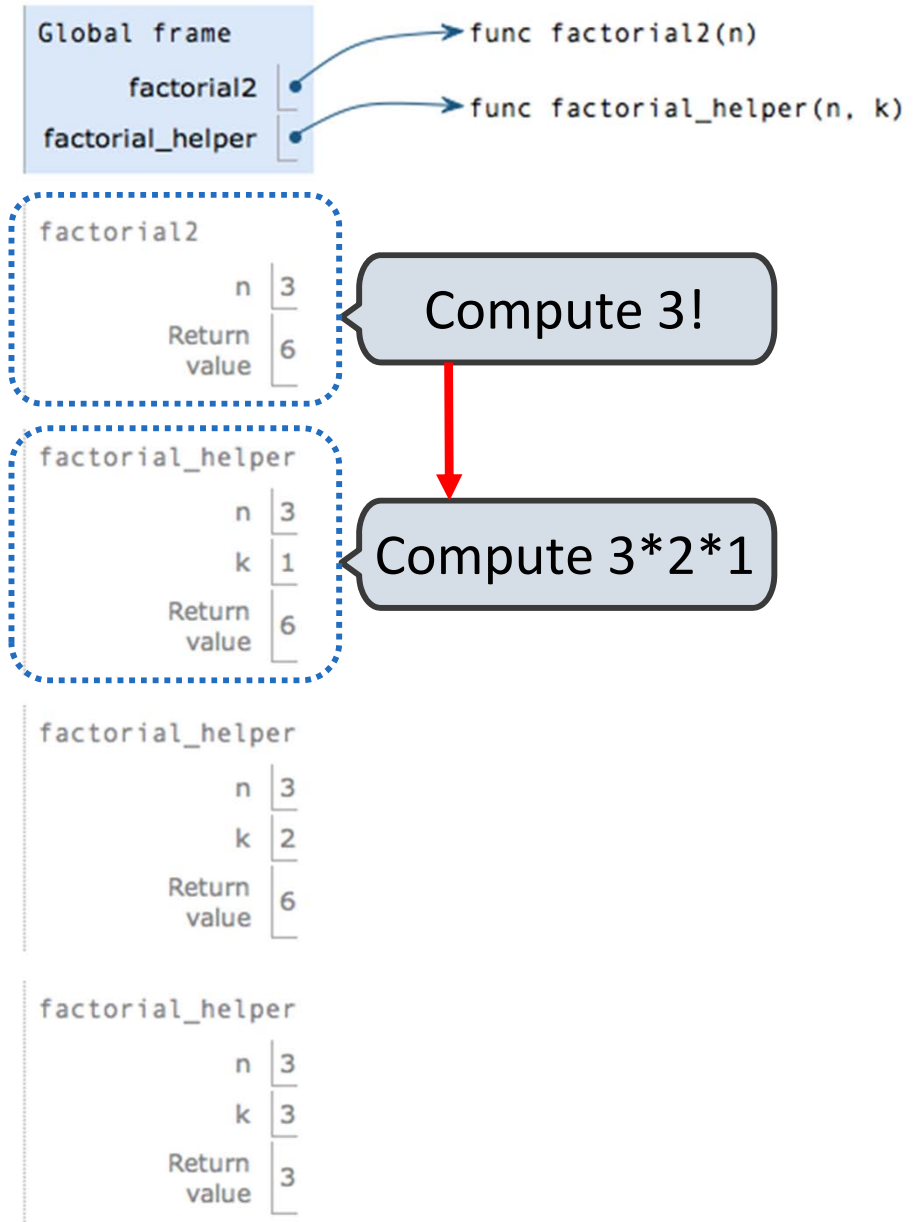
Example: <http://goo.gl/6zz0z>

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



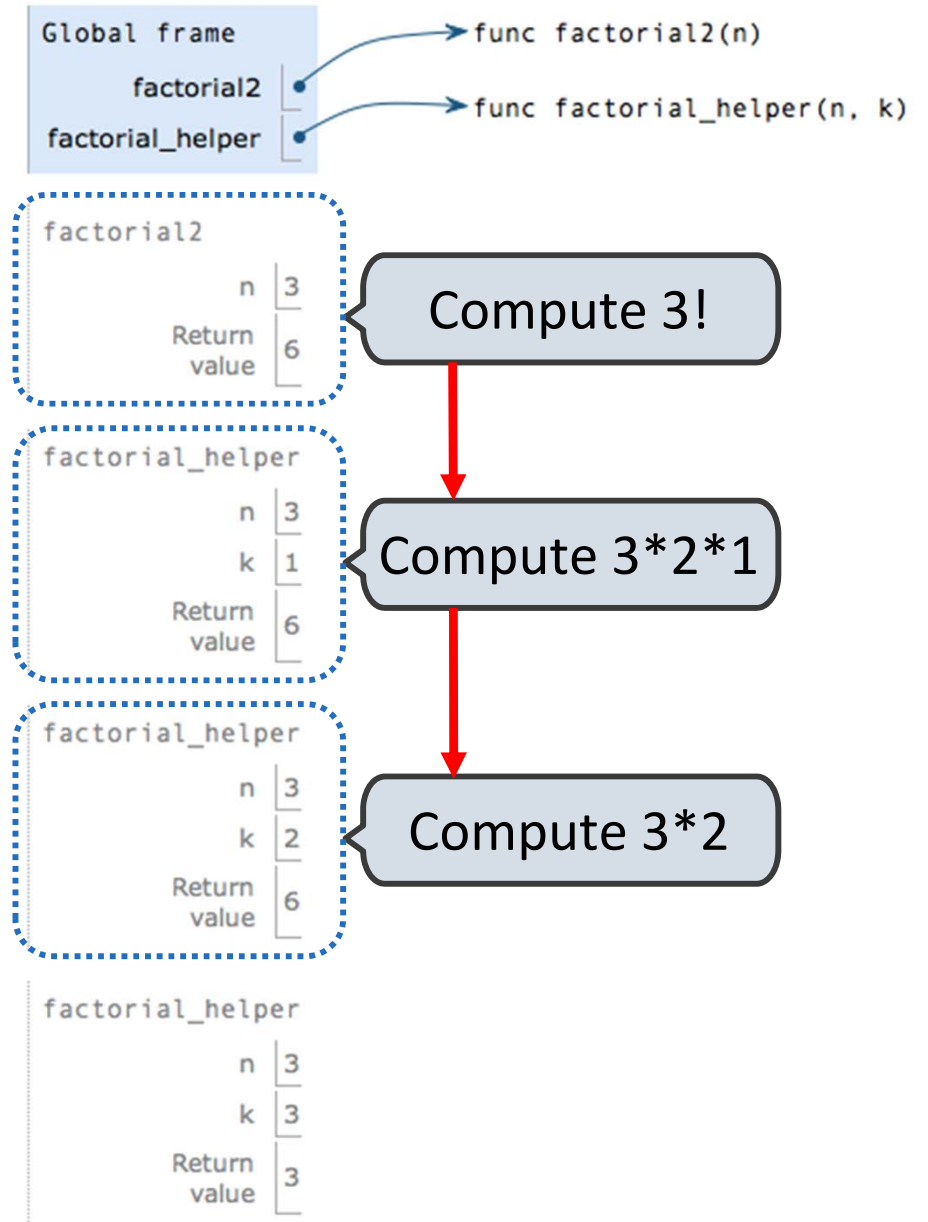
Example: <http://goo.gl/6zz0z>

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



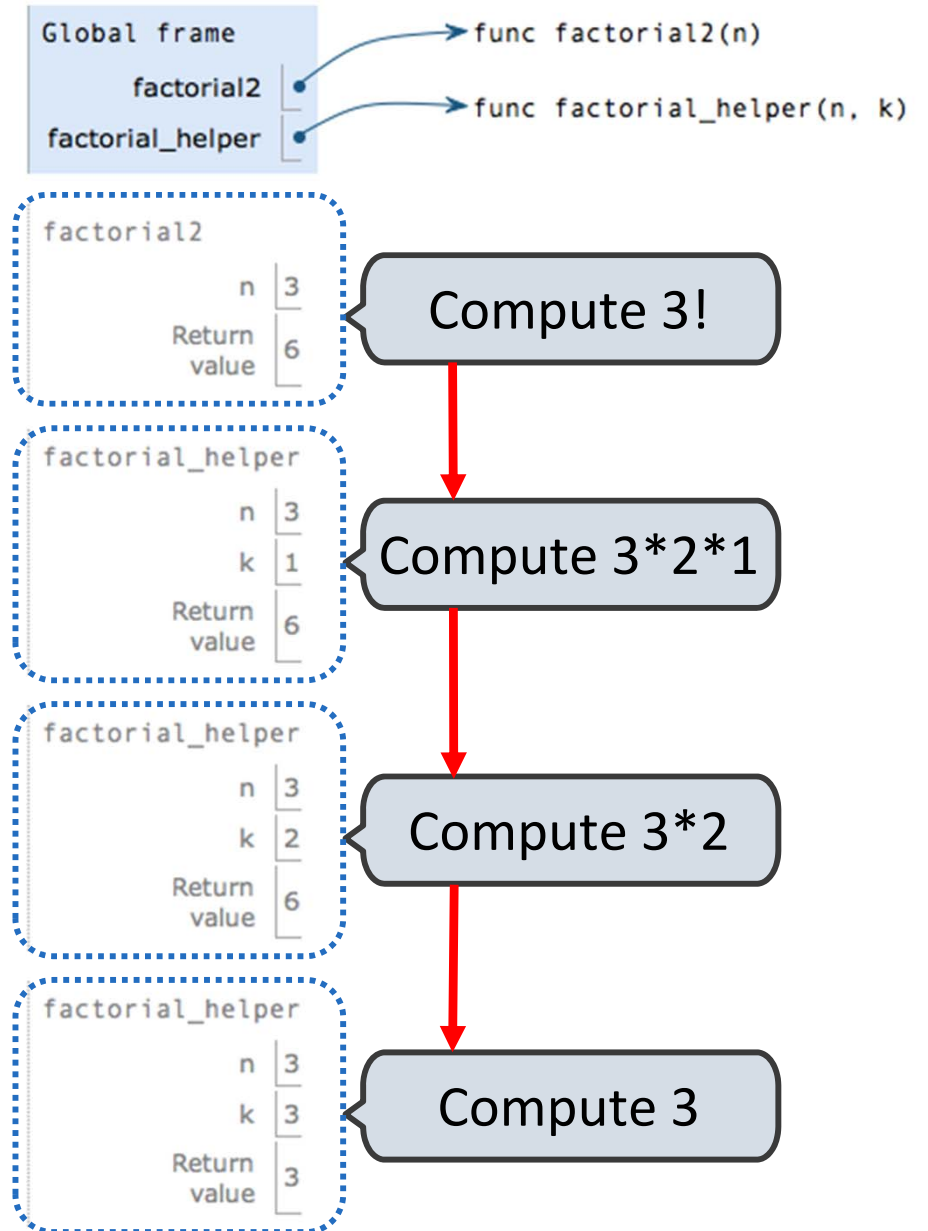
Example: <http://goo.gl/6zz0z>

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



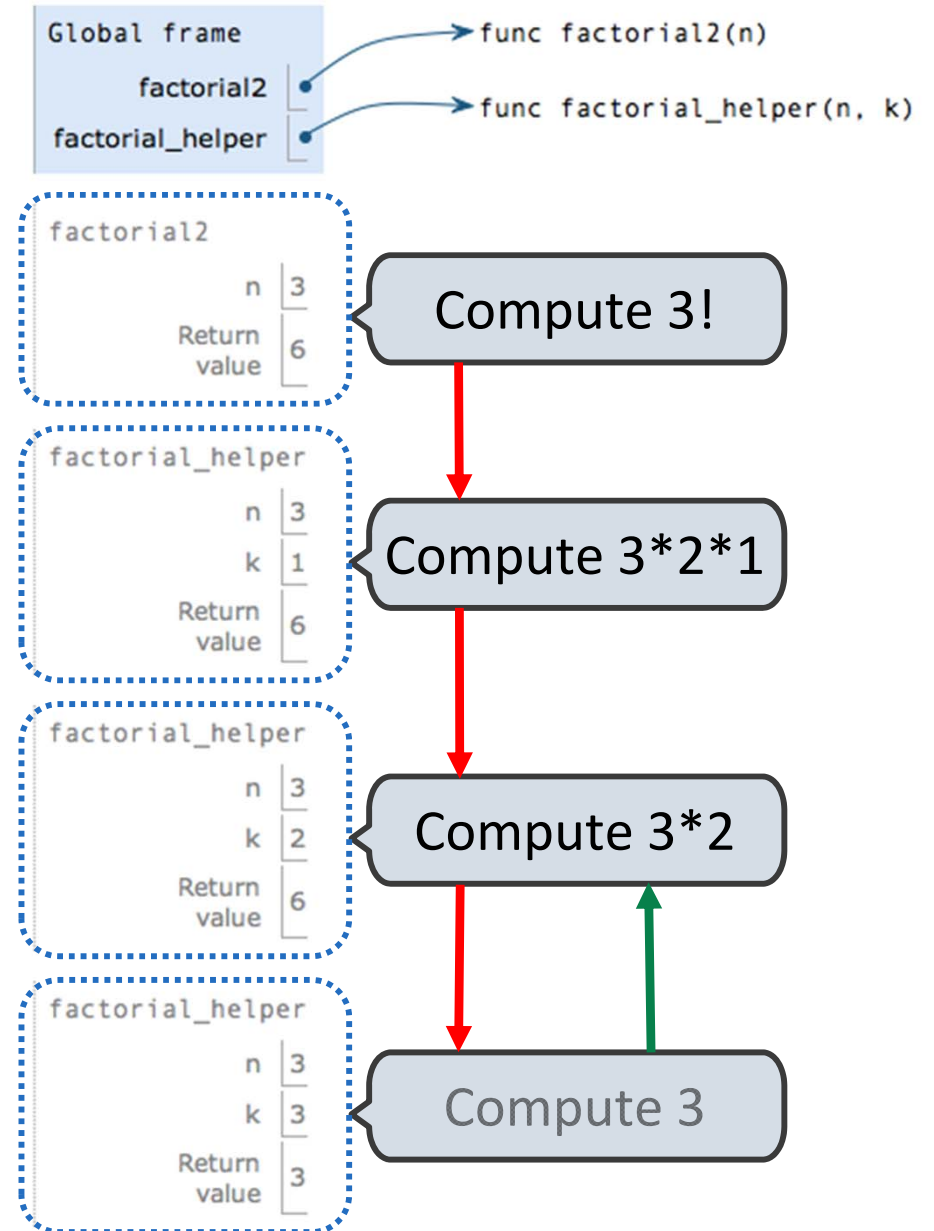
Example: <http://goo.gl/6zz0z>

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



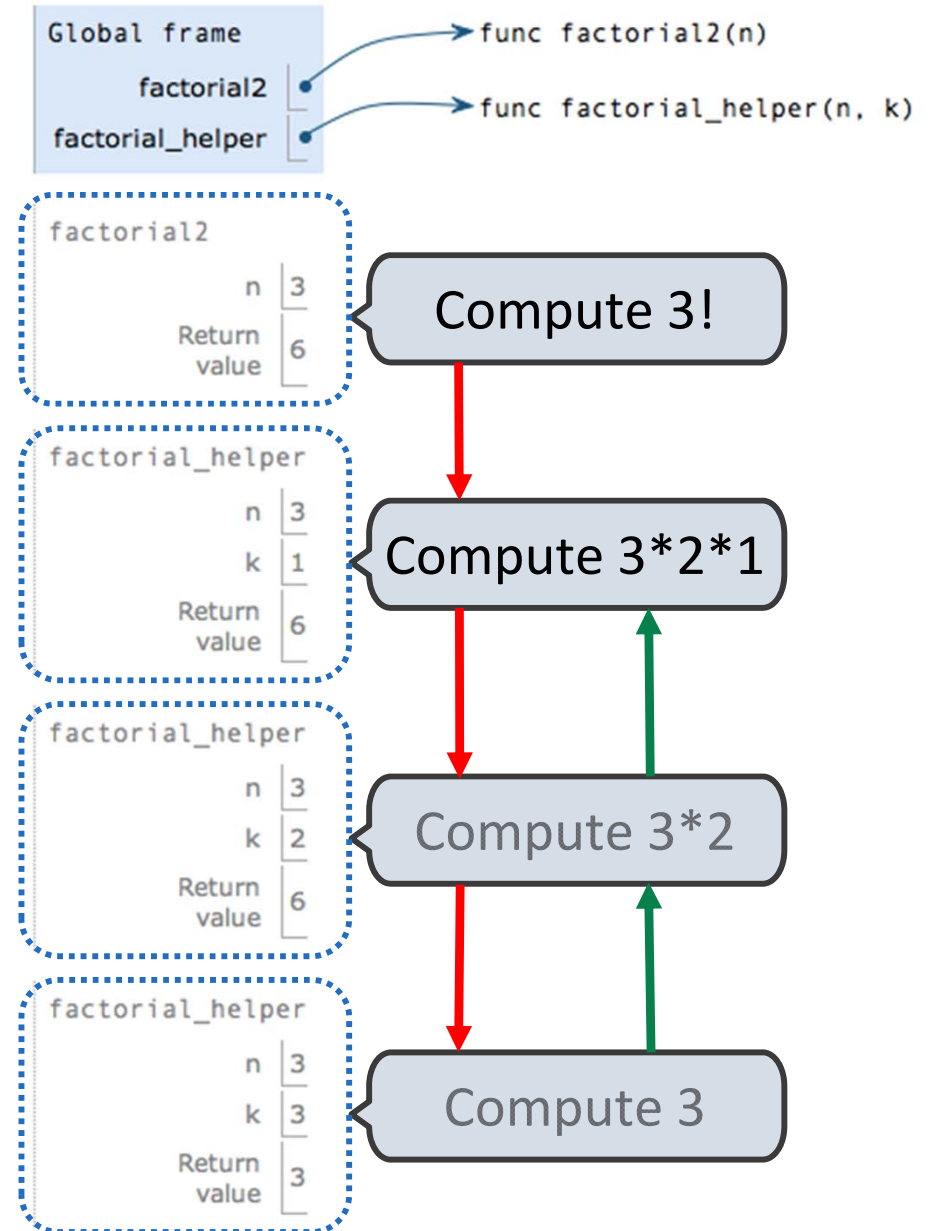
Example: <http://goo.gl/6zz0z>

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



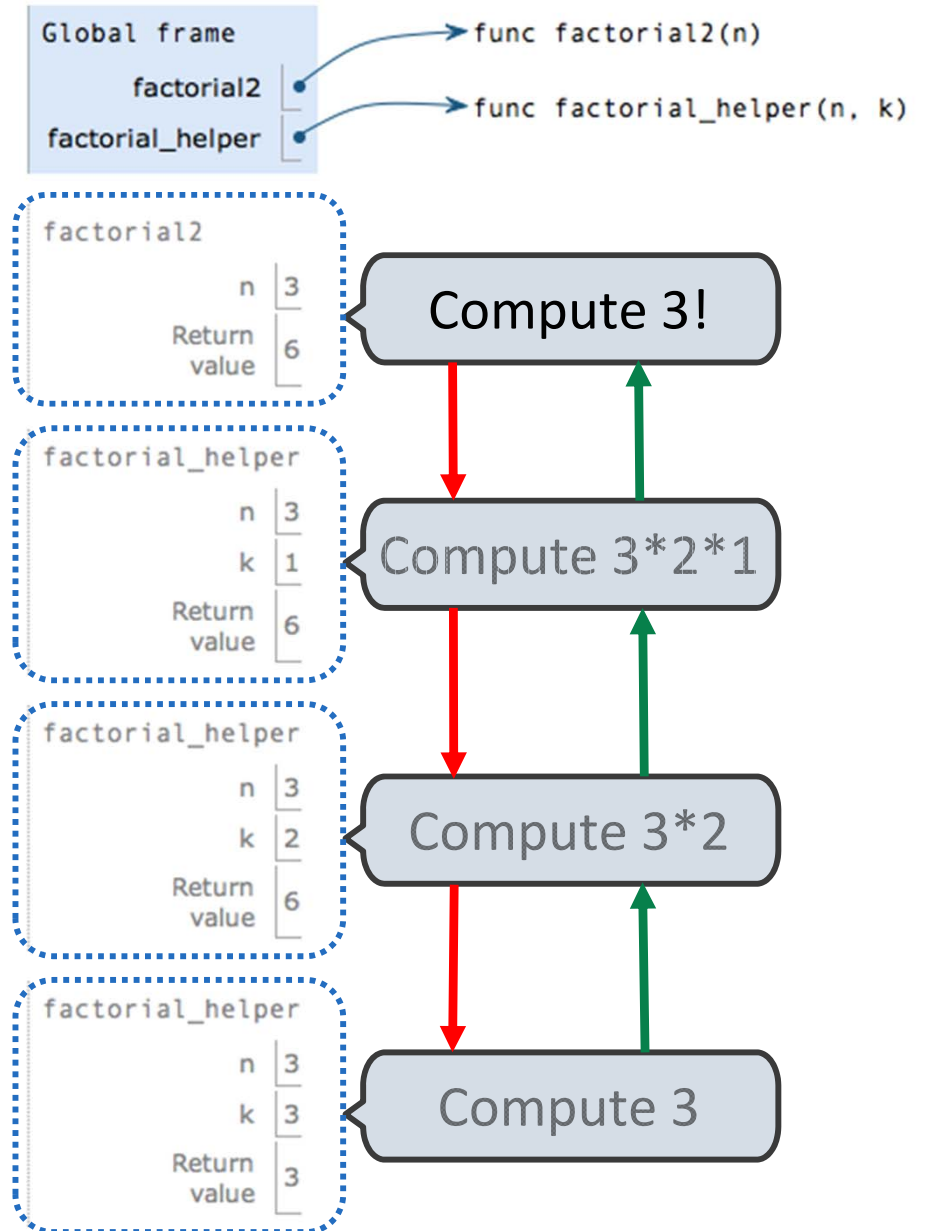
Example: <http://goo.gl/6zz0z>

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



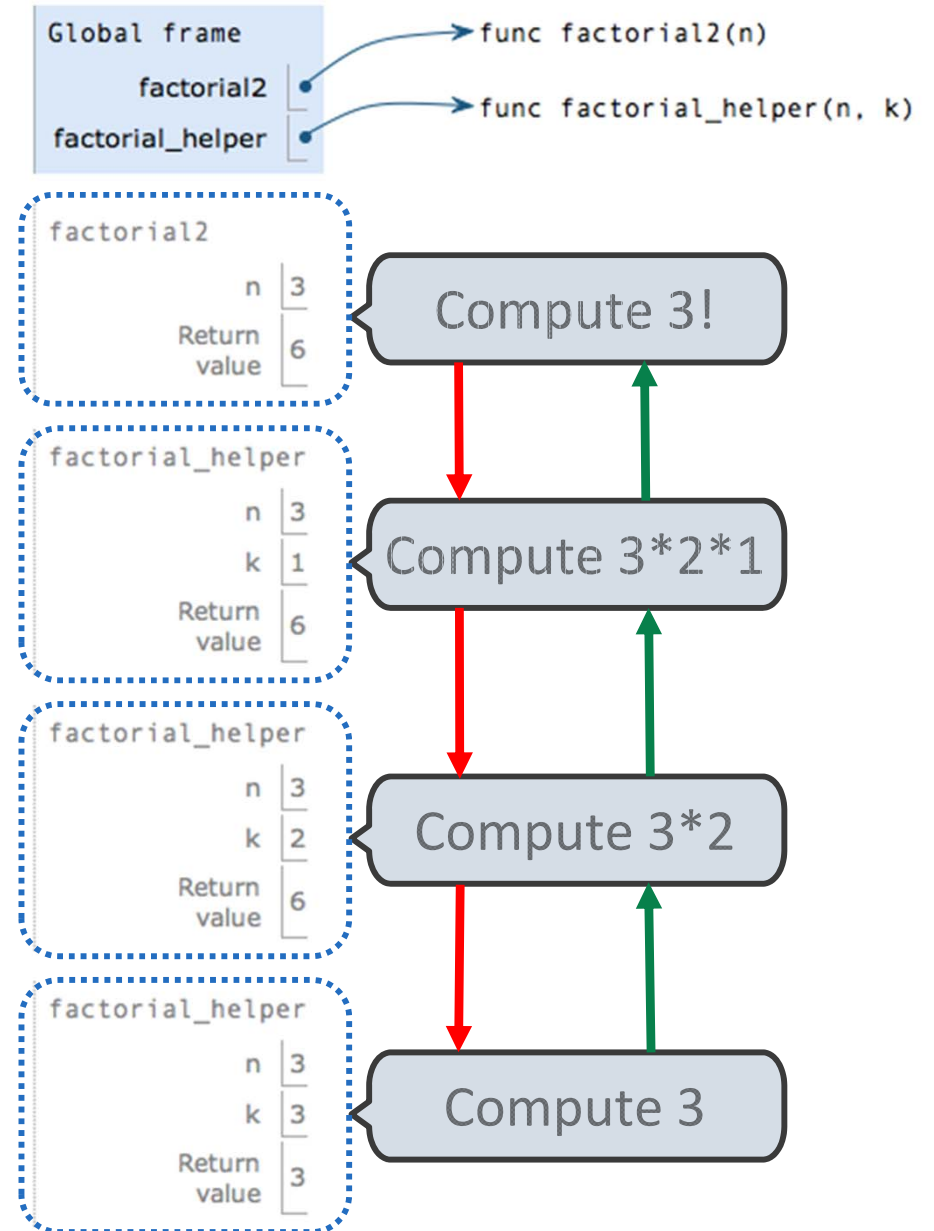
Example: <http://goo.gl/6zz0z>

Reverse Environment Diagram



Here is how the reversed computation evolves

```
1 def factorial2(n):  
2     return factorial_helper(n, 1)  
3  
4 def factorial_helper(n, k):  
5     if k >= n:  
6         return k  
7     return k * factorial_helper(n, k + 1)  
8  
9 factorial2(3)
```



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

Fibonacci Sequence



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

Fibonacci Sequence



The Fibonacci sequence is defined as

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

Fibonacci Sequence



The Fibonacci sequence is defined as

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

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_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

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

Example: <http://goo.gl/DZbRG>

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)
```

Example: <http://goo.gl/DZbRG>

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)
```

Example: <http://goo.gl/DZbRG>

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!

Tree recursion



Tree recursion



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

Tree recursion



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

This is called *tree recursion*

Tree recursion



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

This is called *tree recursion*

