CS134 Lecture 19:
Recursion

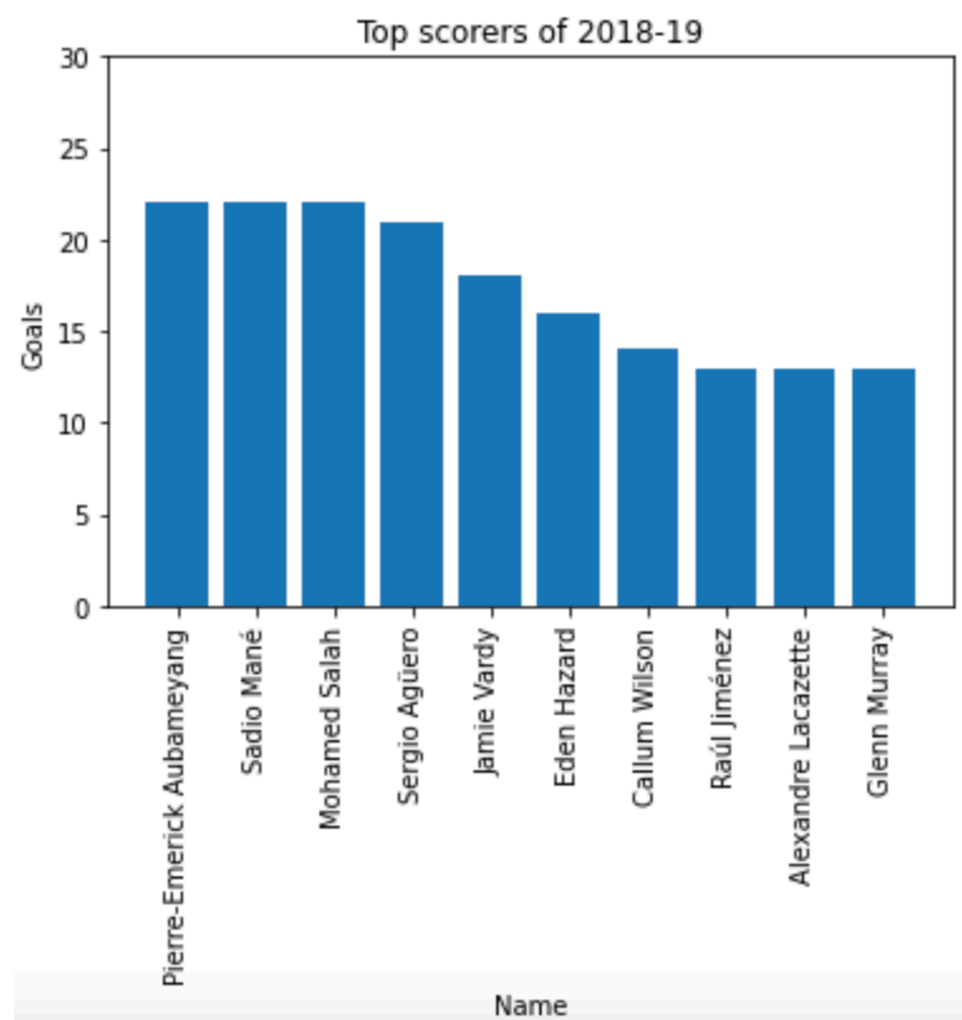
Announcements & Logistics

- **Lab 6 due Wed/Thurs at 10 pm**
  - Uses dictionaries, plotting, CSV files
- **HW 6** will be out today, due Mon at 10pm
- Lab 7, 8, and 9 are **partner labs**
  - Fill out google form sent by Lida by **noon tomorrow (Thursday)**!
  - Pair programming is an important skill as well as a vehicle for learning
- Pick up your **graded midterm exam** at the end of class
  - Will use last few mins of lecture to discuss the midterm

Do You Have Any Questions?

Last Time

- Worked through an example involving CSVs, dictionaries, and sets
- Discussed plotting with matplotlib
    - Python is pretty useful for data processing and visualization!



Today's Plan

Intro To Recursion

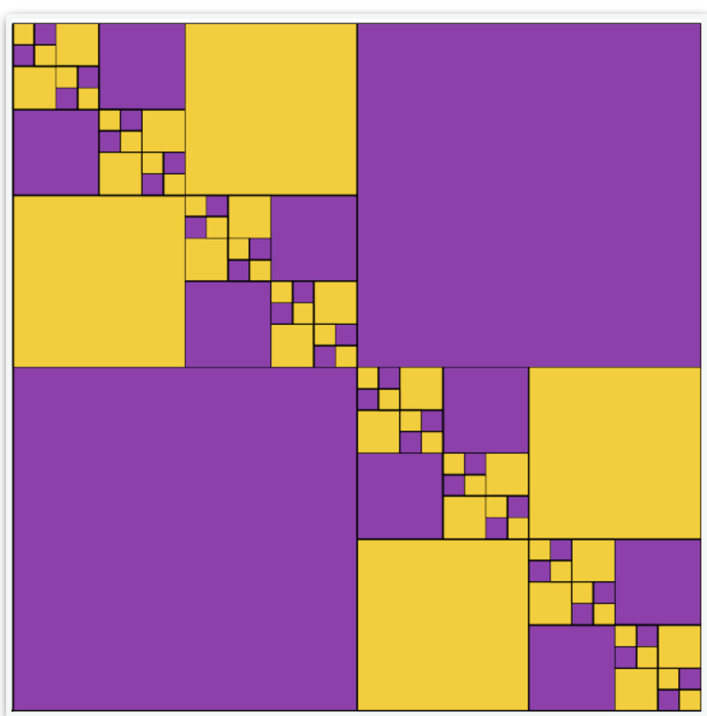
- Discuss what we mean by the term **recursion**
- Practice translating recursive ideas into recursive programs
- Examining the relationship between **recursive** and **iterative** programs
  - That is, how do recursive ideas relate to the iterative ideas (for loops, while loops) we've covered so far?

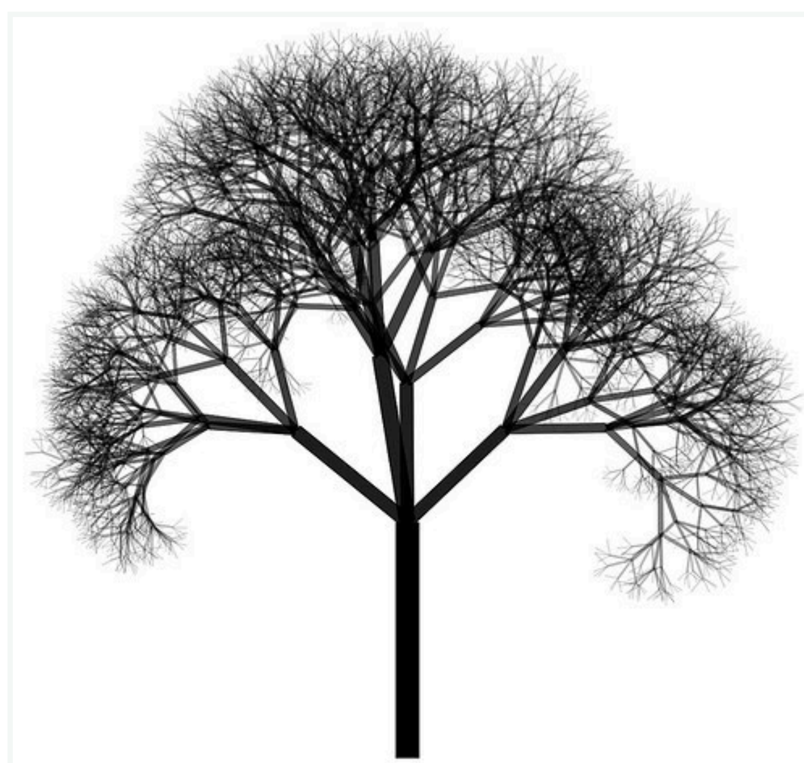
Where are We Going?

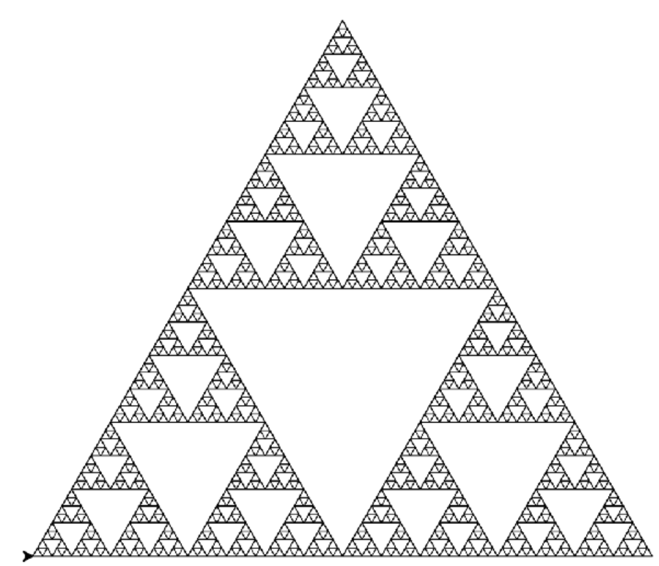
- First half of CS134: learned some **fundamental programming concepts**
    - Functions, conditionals, loops, data types
    - Built-in data structures and operations
- Looking ahead to the second half: more emphasis on **algorithmic** and **conceptual** topics: more "computational thinking"
    - **Recursion** (~1 week)
    - Defining our own **data types** using **classes and objects** (~2 weeks)
        - Object oriented programming topics
    - Continue developing our intuition regarding efficient vs inefficient code

Why Learn About Recursion?

- Recursion is an important problem solving paradigm
  - An alternative to iteration for repeatedly performing a task
  - Process that lets us "divide, conquer, combine"
  - Useful to build and maintain data structures (like trees and lists)
- Provides a different lens to view the world
  - If you love procrastination — recursion is just the thing for you!







So What Is Recursion?

- We will explore recursion by first seeing some examples in action
- Let's revisit a familiar function: count_occurrences(elem, lst)
  - Goal is to return the number of times elem appears inside list lst

```
def count_occurrences(elem, lst) :
    count = 0
    for item in lst :
        if item == elem :
            count = count + 1
    return count
```

  - This function is **iterative**: we iterate through the list using a for loop, and compare elem against each item in the list

So What Is Recursion?

- One of the keys to thinking recursively breaking down the problem:
  - What is the smallest version of the problem that we can *immediately* solve?
  - For larger versions of the problem, is there a small step we can take that brings us closer to the smallest version of the problem?
- Let's answer these questions for count_occurrences(elem, lst)
  - How many times does elem appear in an empty list?

```
def count_occurrences(elem, lst) :
   # smallest list we know the answer to is empty list!
   if len(lst) == 0:
       return 0
```

So What Is Recursion?

- How many times does elem appear in an empty list?

```
def count_occurrences(elem, lst) :
    # smallest list we know the answer to is empty list!
    if len(lst) == 0:
        return 0
```

- How many times does elem appear in a larger list?
  - We don't know yet! But we do know that the list has at least one element in it, otherwise we would have returned 0…
  - **Idea**: let's break the problem into two smaller problems
    - Is the first item in the list equal to elem?
    - How many times does elem appear in the rest of the list?

So What Is Recursion?

- **Idea**: let's break the problem into two smaller problems
  - Is the first item in the list equal to elem?
  - How many times does elem appear in the rest of the list?

```
def count_occurrences(elem, lst) :
    # smallest list we know the answer to is empty list!
    if len(lst) == 0:
        return 0

    # Is the first item in the list equal to elem?
    first = 1 if elem == lst[0] else 0

    # How many times does elem appear in the rest of the list?
    rest = count_occurrences(elem, lst[1:])

    # combine our results
    return first + rest
```

So What Is Recursion?

- Surprisingly, this function works!
- Some observations:
  - Some paths through the function call the same function again
    - This is what makes the function recursive
  - Other paths through the function (the smallest case that we can solve immediately) simply return the answer
    - This is called a base case. Ever recursive function must have at least one base case!
  - It is important that our recursive calls move us closer to our base case(s), otherwise we may get stuck in an infinite loop!
- Now let's dive into the principles of recursive problem solving more formally to get a better feeling for what is going on…

Recursive Approach to Problem Solving

- A recursive approach to problem solving has two main parts:
  - **Base case(s).** When the problem is **so small**, we solve it directly, without having to reduce it any further (this is when we stop)
  - **Recursive step.** Does the following things:
    - Performs an action that contributes to the solution (take one step)
    - **Reduces** the problem to a smaller version of the same problem, and calls the function on this **smaller subproblem** (break the problem down into a slightly smaller problem + one step)
- The recursive step is a form of "wishful thinking": assume the unfolding of the ***recursion*** will take care of the smaller problem by eventually reducing it to the base case
- In CS136/256, this form of wishful thinking will be introduced more formally as the *inductive hypothesis*

Understanding Recursive Functions

- Let's review a simple recursive function that gives us some intermediate feedback through `print` statements.
- Write a recursive function that prints integers from `n` down to `1`
- Recursive definition of countdown:
  - **Base case:** `n = 1`, `print(n)`

    Print and stop
  - **Recursive rule:** `print(n), call count_down(n-1)`

    Perform one step

    Reduce the problem (or make the problem "smaller")

Understanding Recursive Functions

- Recursive definition of countdown:
  - Base case: `n = 1, print(n)`
  - Recursive rule: `print(n), count_down(n-1)`

```
def count_down(n):
    '''Prints numbers from n down to 1'''
    if n == 1:  # Base case
        print(n)
    else: # Recursive case: n > 1:
        print(n)
        count_down(n-1)
```

```
>>> result = count_down(5)
5
4
3
2
1
```

Understanding Recursive Functions

- Recursive functions seem to be able to reproduce looping behavior without writing any loops at all
- To understand what happens behind the scenes when a function calls itself, let's review what happens when a function calls another function
- Conceptually we understand function calls through the **function frame model**

Most of the examples we're looking at today are easily written iteratively, but we'll be looking at problems on Friday where that may not be the case!

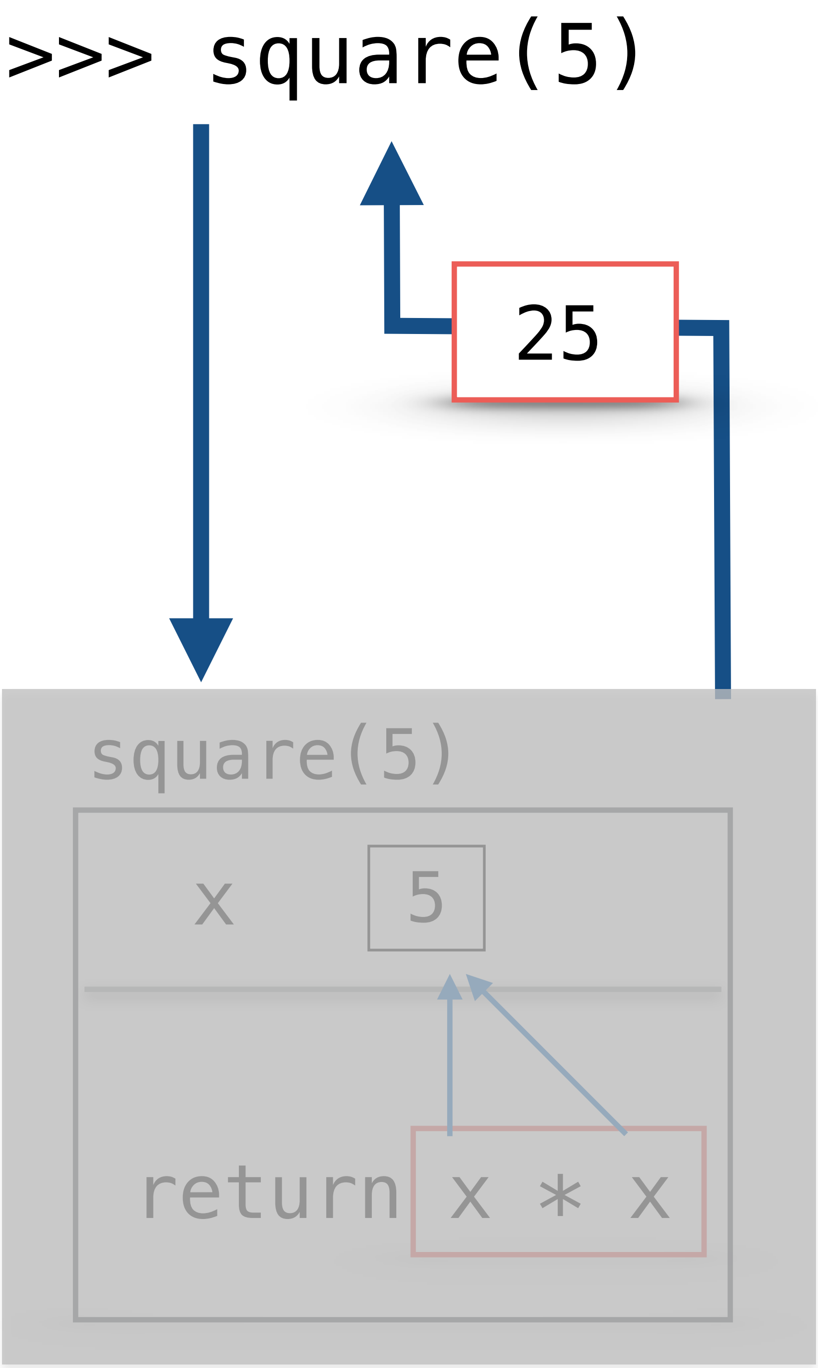
Review:
Function Frame Model

Review: Function Frame Model

- Consider a simple function **`square`**
- What happens when **`square(5)`** is invoked?

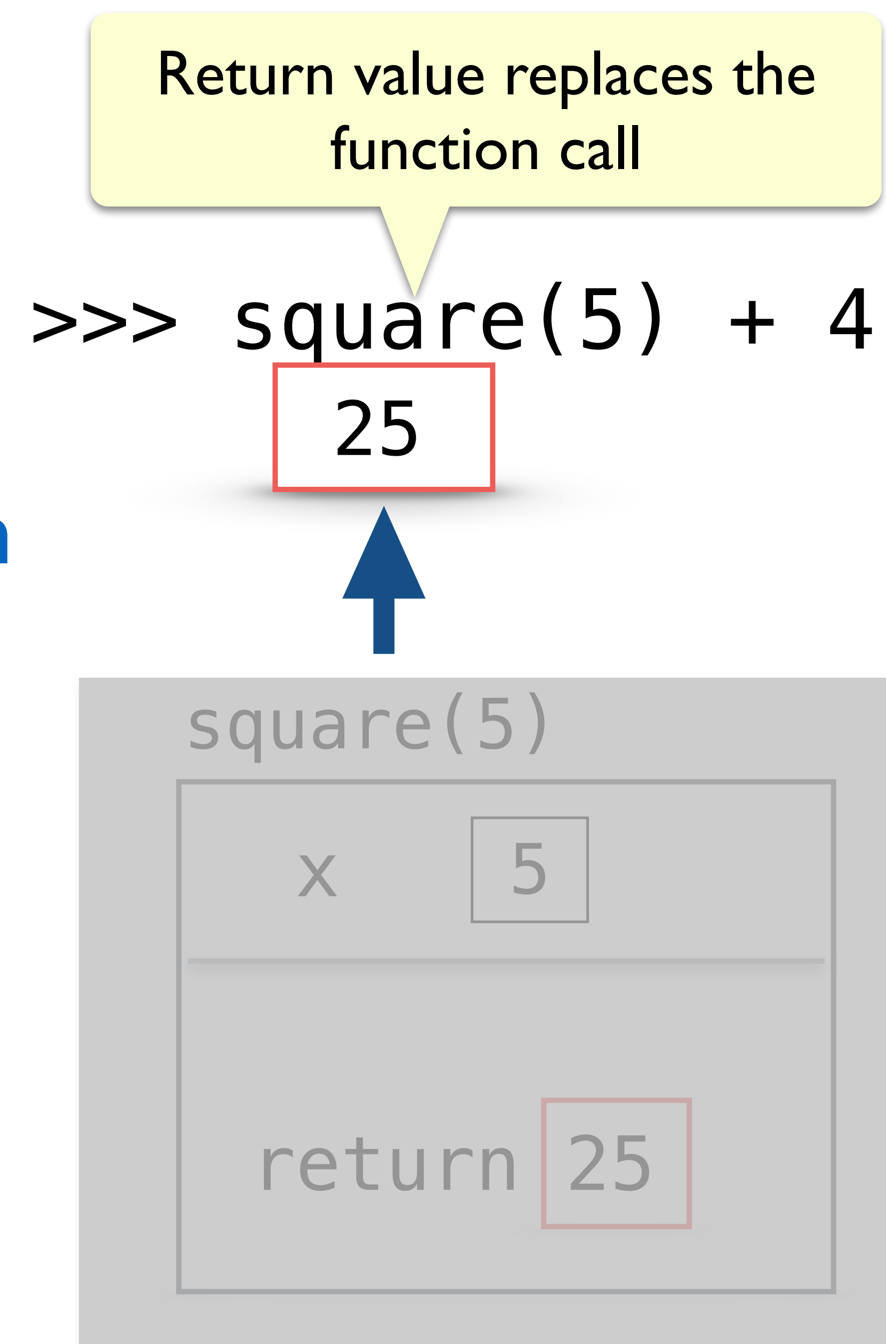
```
def square(x):
  return x*x
```

Review:
Function Frame Model



Summary:
Function Frame Model

- When we **return** from a function frame "control flow" goes back to where the function call was made
- Function frame (and the local variables inside it) **are destroyed after the return**
- If a function does not have an explicit return statement, it returns **None** after all statements in the body are executed



Review:
Function Frame Model

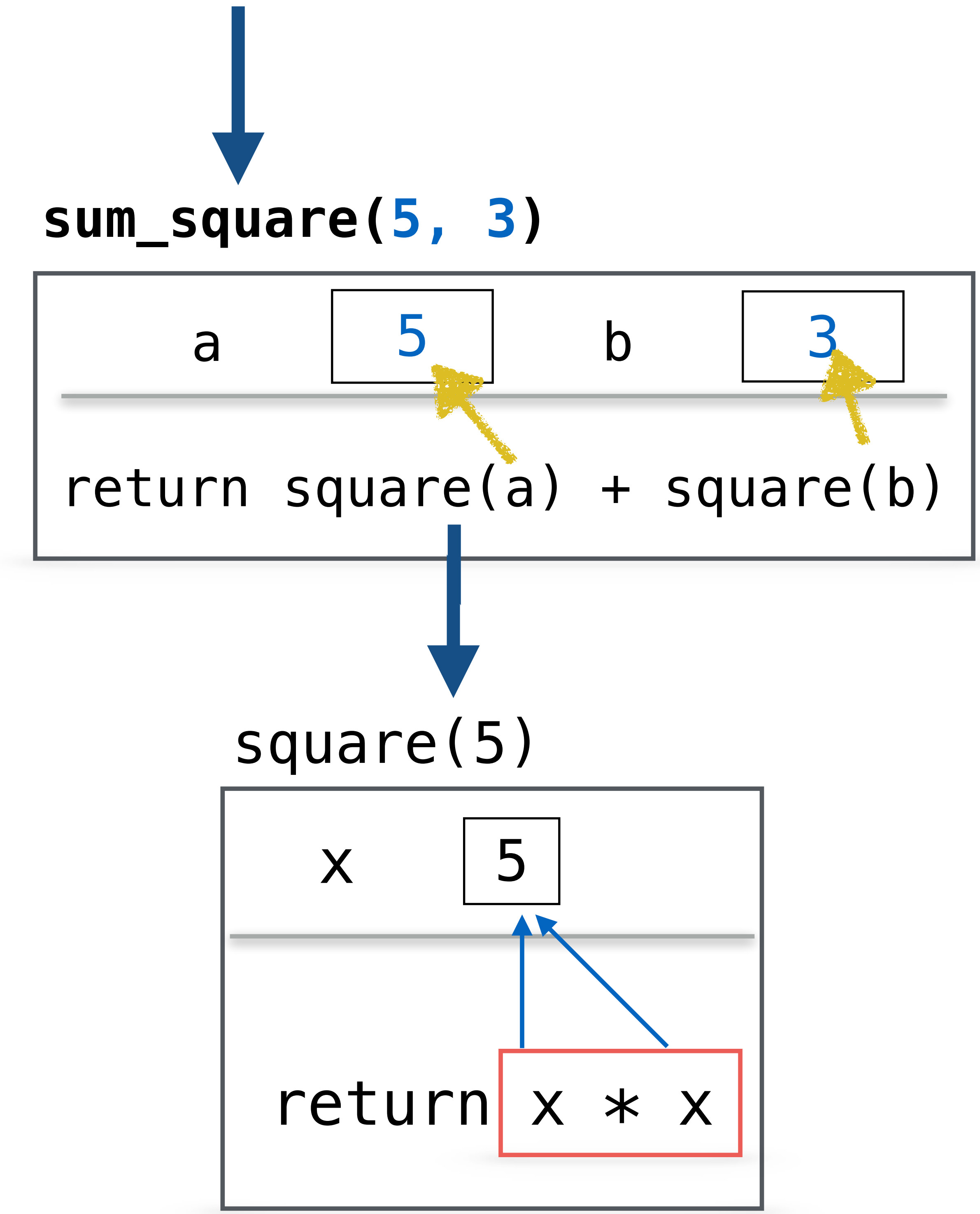
- How about functions that call other functions?

```
def sum_square(a, b):
  return square(a) + square(b)
```

- What happens when we call `sum_square(5, 3)`?

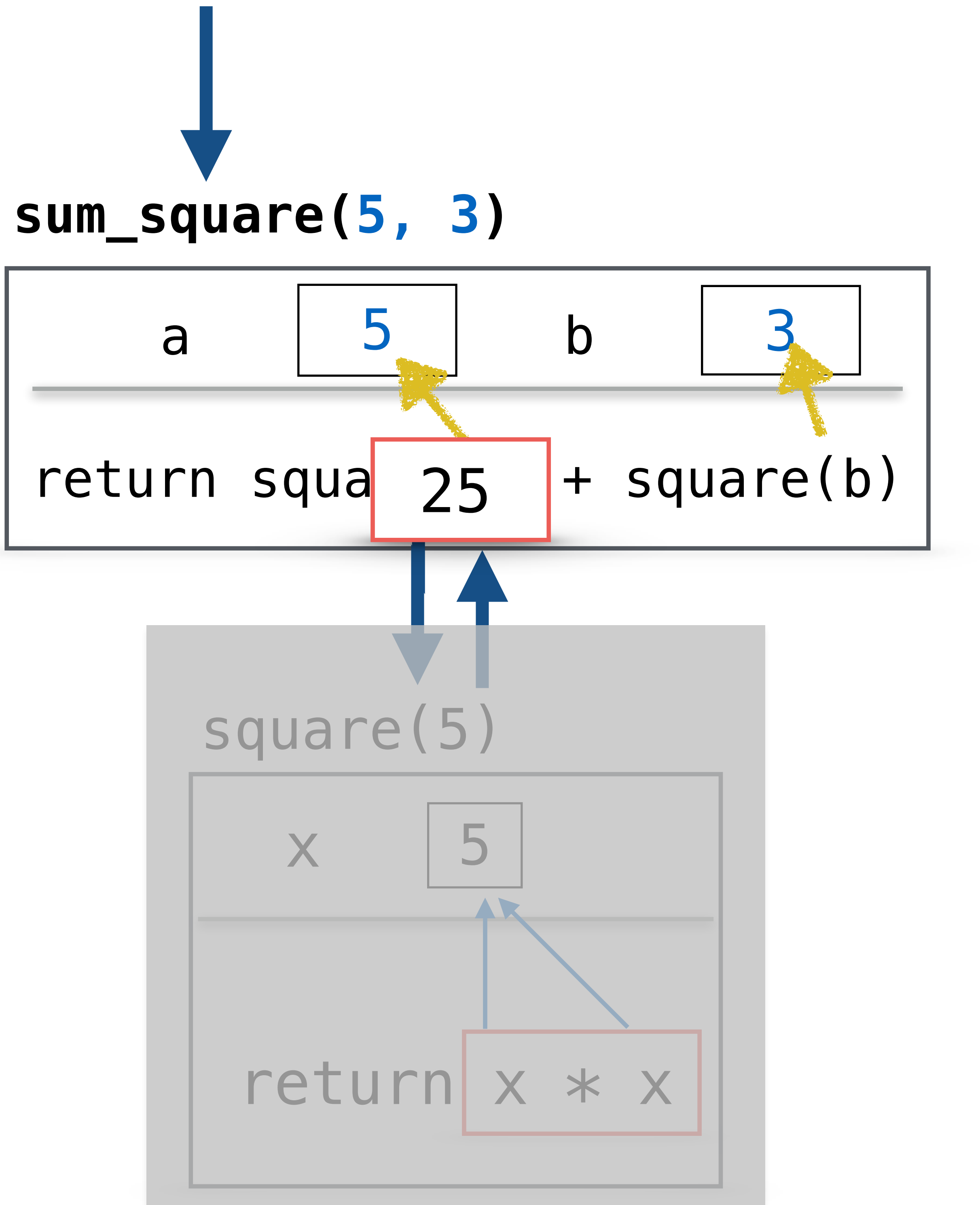
```
def sum_square(a, b):
  return square(a) + square(b)
```

```
>>> sum_square(5,3)
```



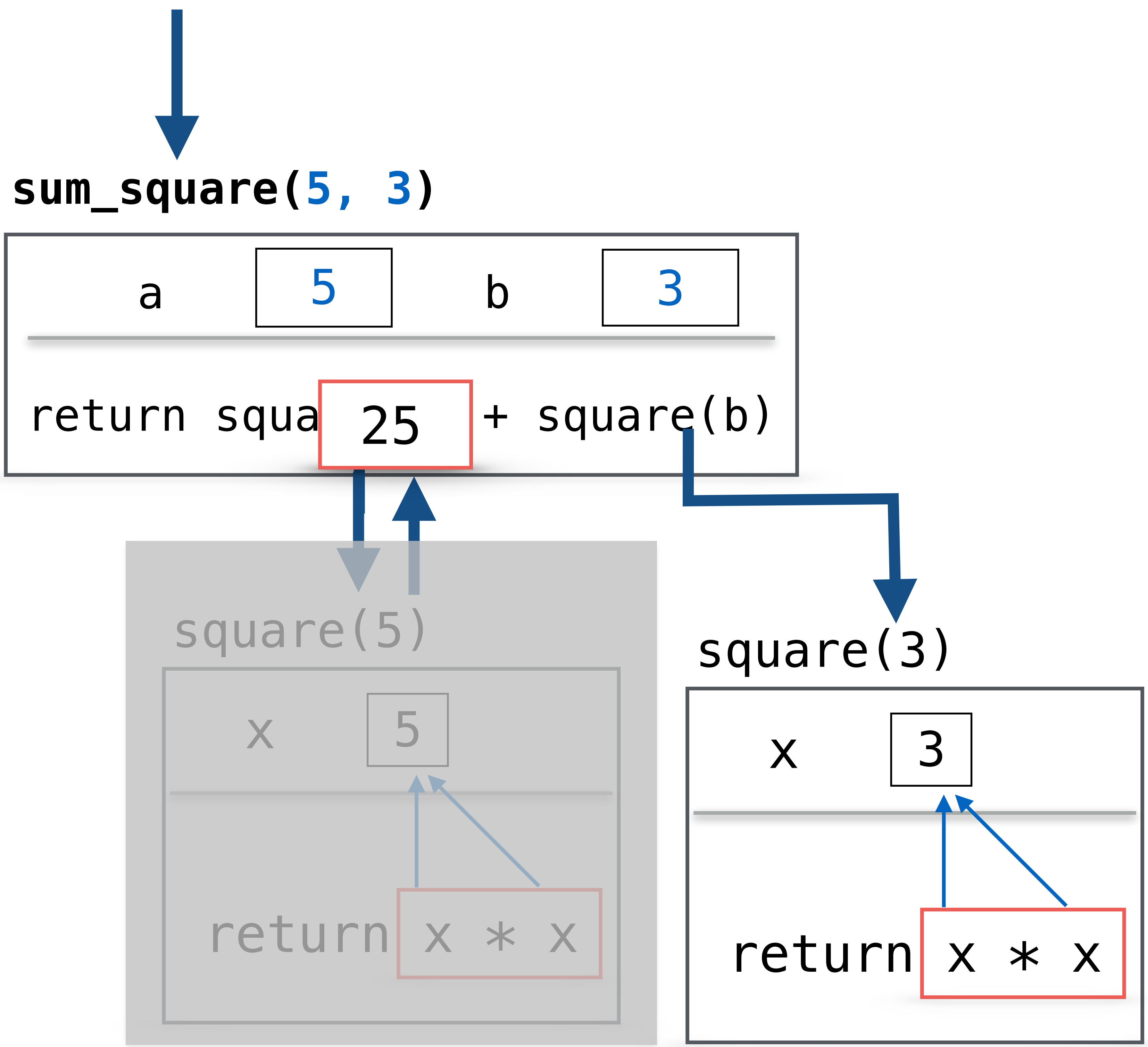
```
def sum_square(a, b):
   return square(a) + square(b)
```

```
>>> sum_square(5,3)
```



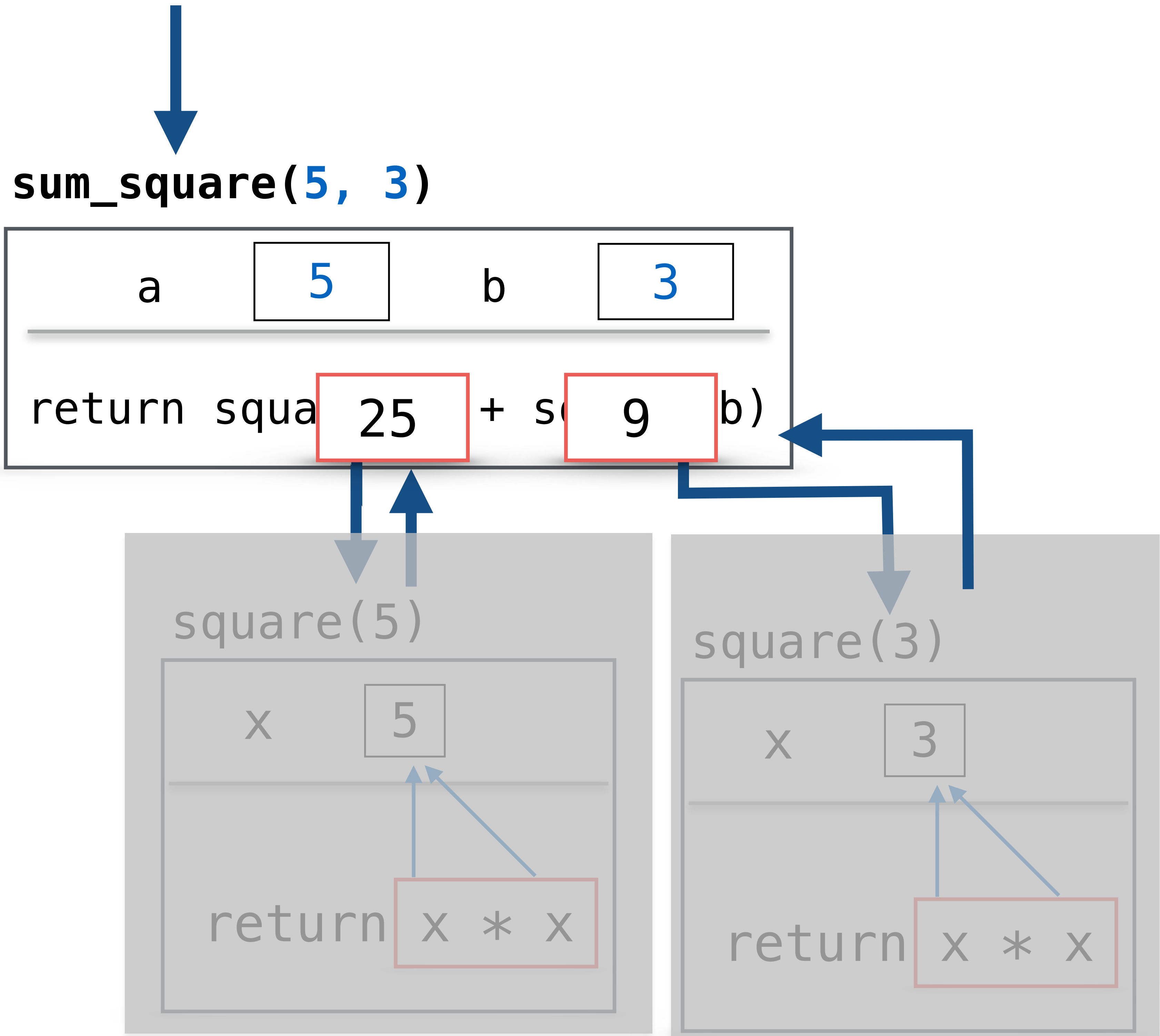
```
def sum_square(a, b):
   return square(a) + square(b)
```

```
>>> sum_square(5,3)
```

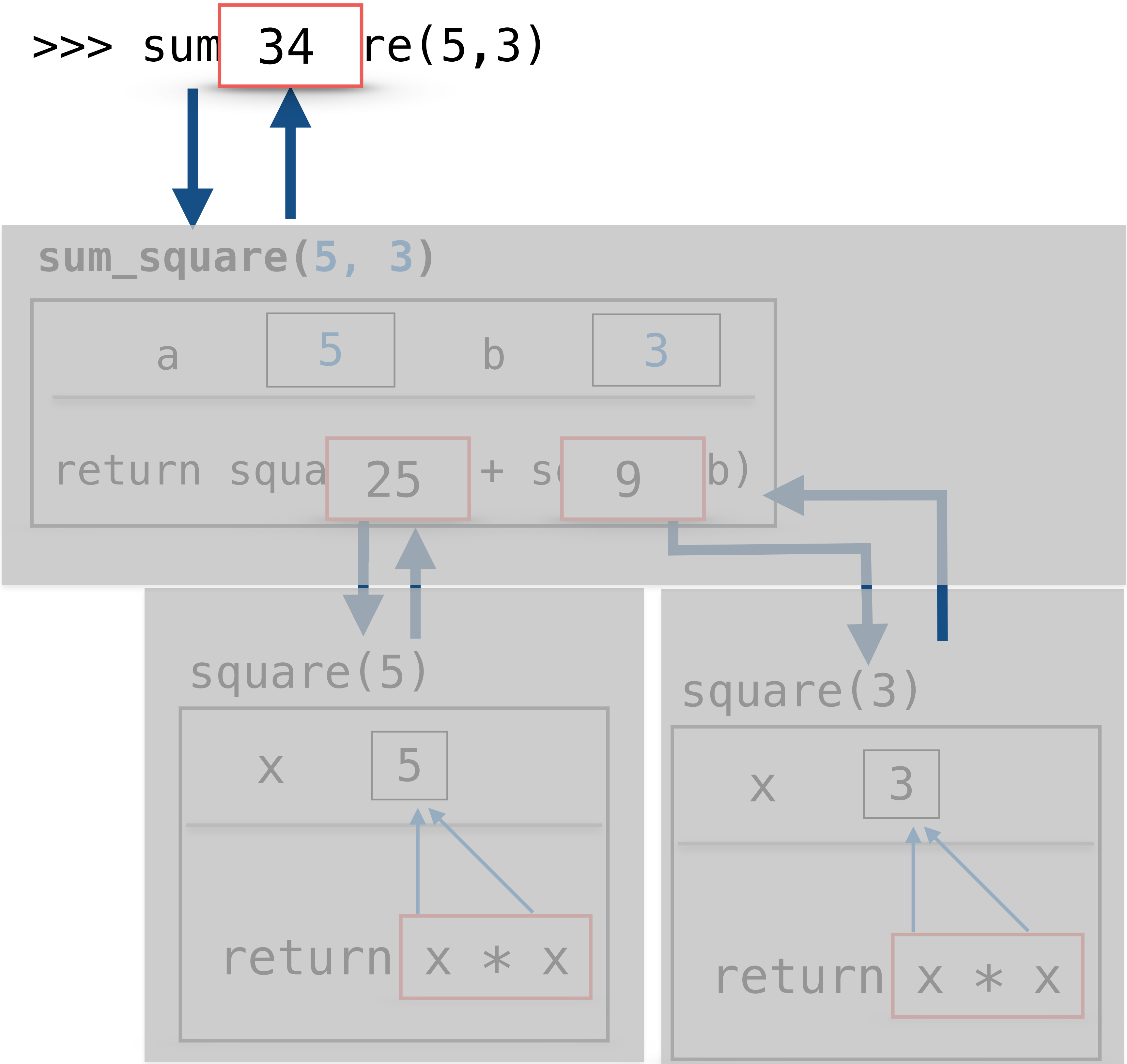


```
def sum_square(a, b):
    return square(a) + square(b)
```

```
>>> sum_square(5,3)
```



```
def sum_square(a, b):
    return square(a) + square(b)
```

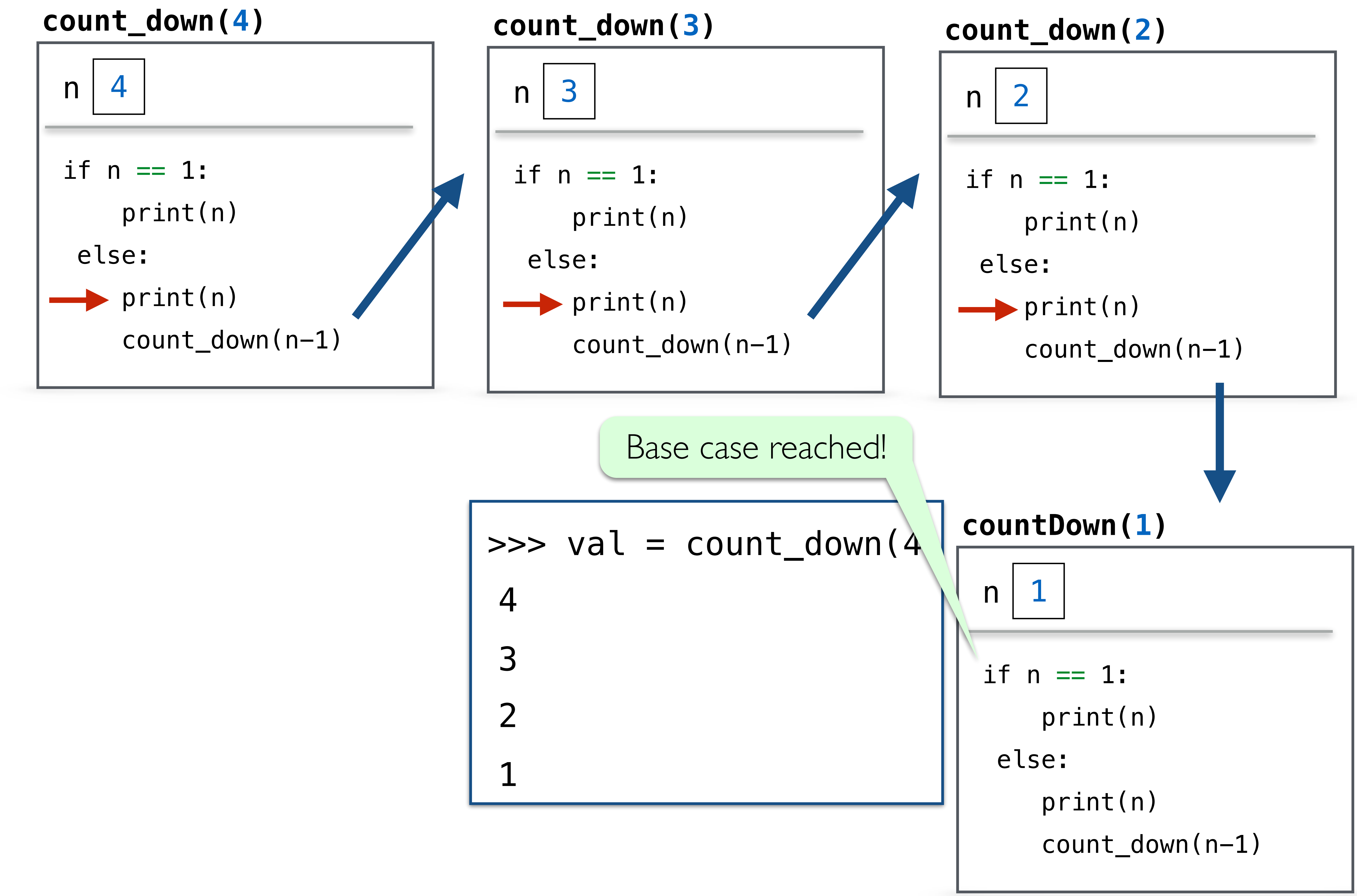


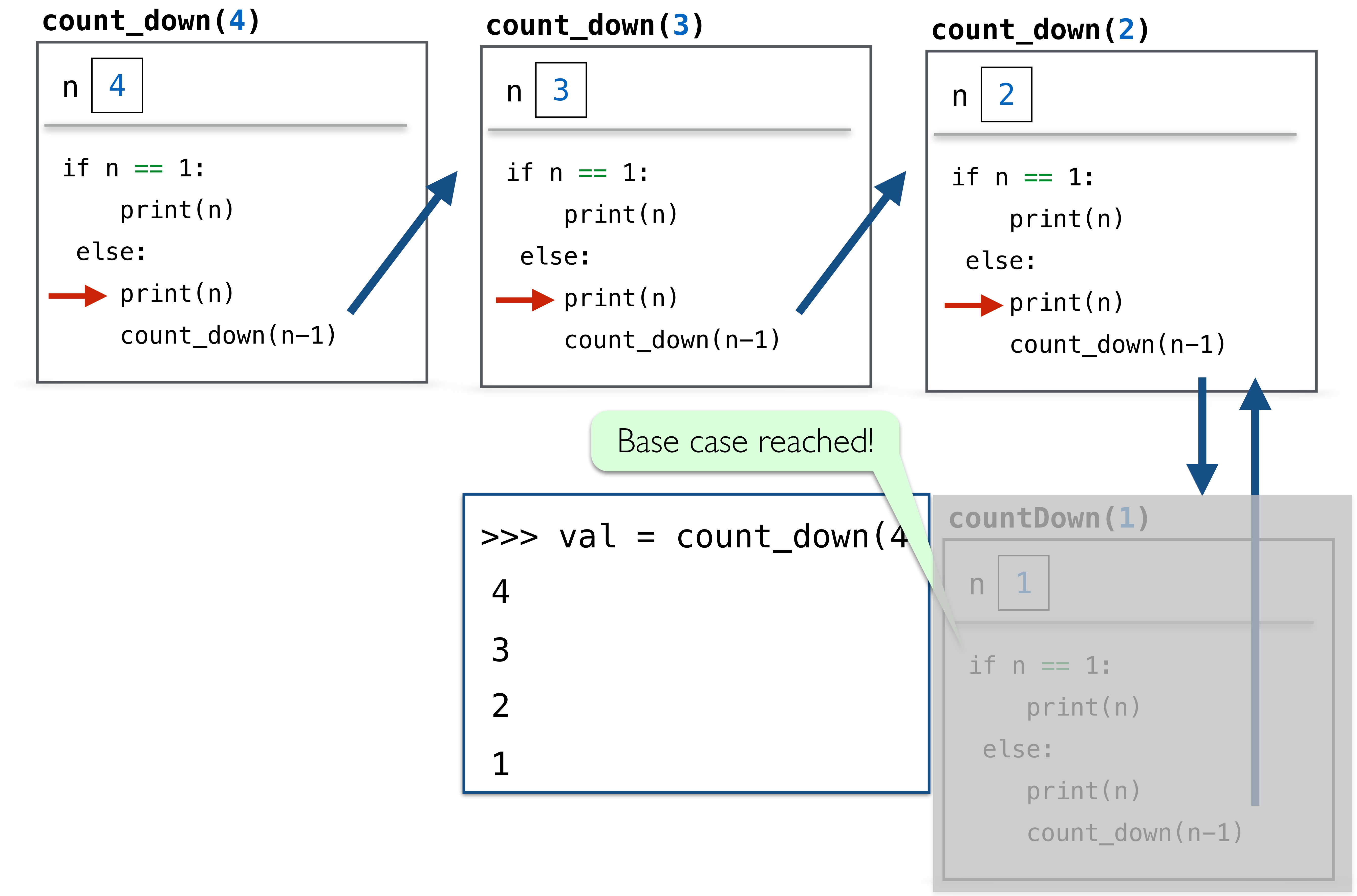
Function Frame Model to Understand `count_down`

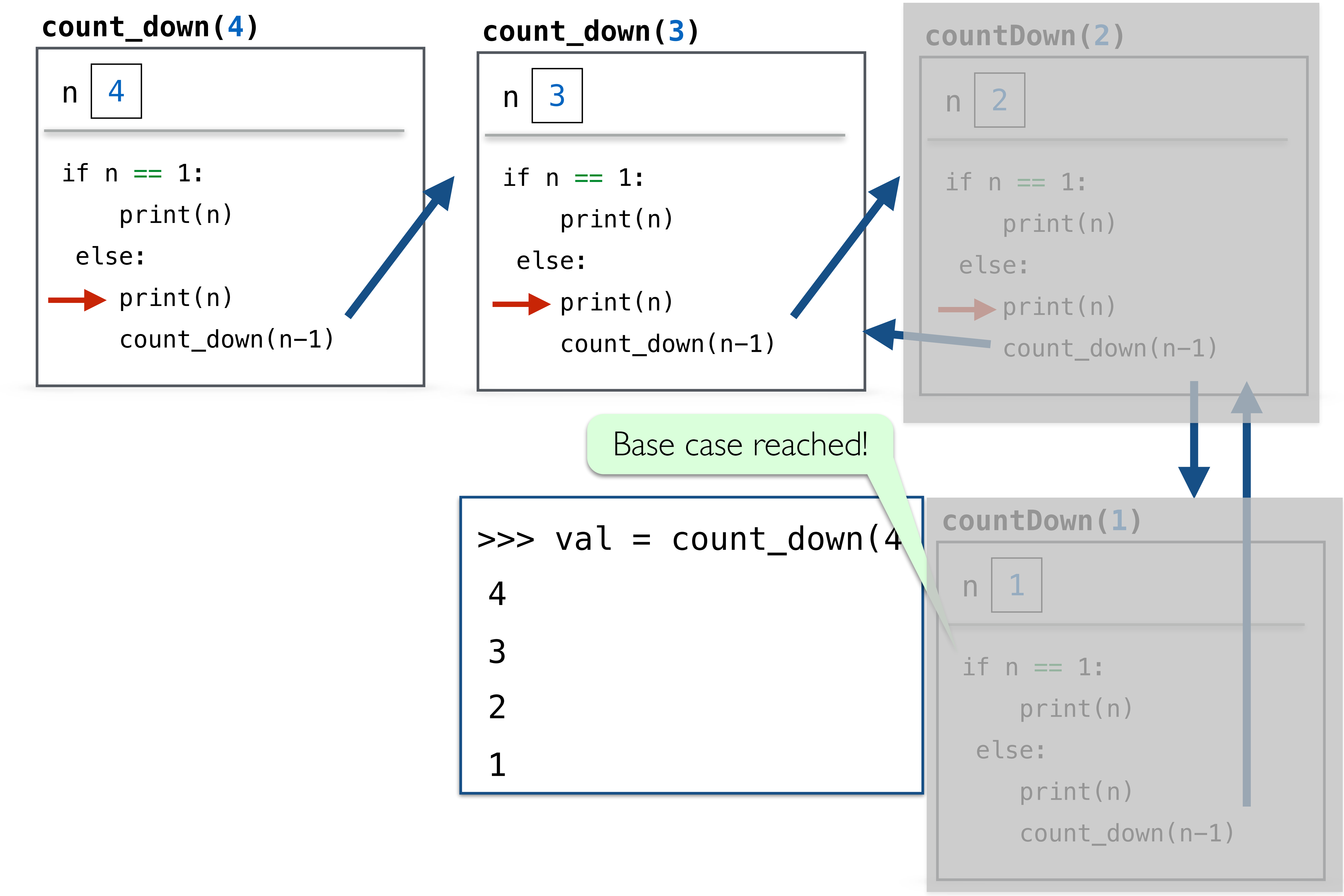
```python
def count_down(n):
    '''Prints ints from n down to 1'''
    if n == 1:
        print(n)
    else:
        print(n)
        count_down(n-1)
```

```
>>> val = count_down(5)
5
4
3
2
1
```

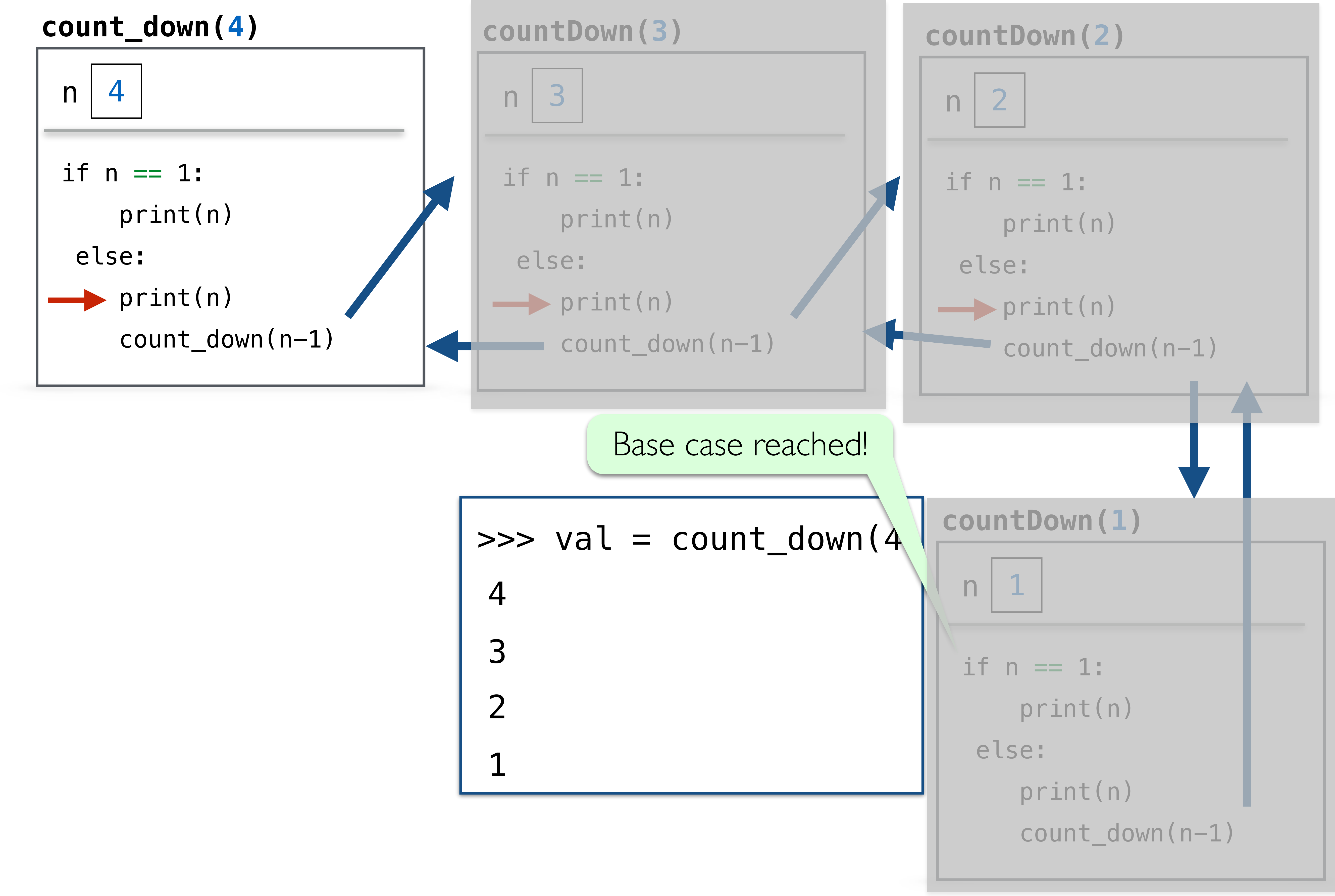
```
>>> val = count_down(4)
4
3
2
1
```

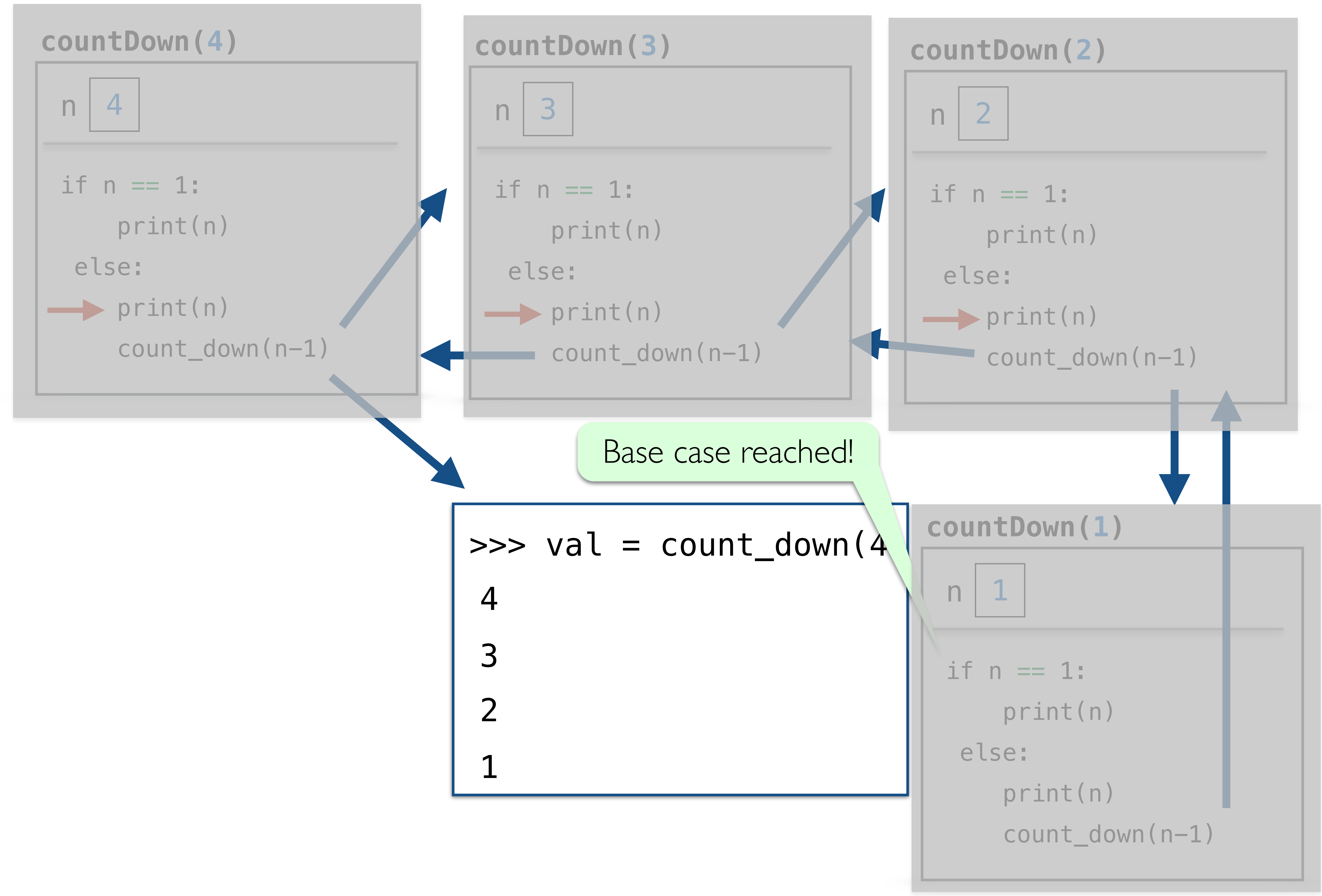

count_down(4)
n 4
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
count_down(3)
n 3
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
count_down(2)
n 2
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
Base case reached!
>>> val = count_down(4
4
3
2
1
countDown(1)
n 1
if n == 1:
print(n)
else:
print(n)
count_down(n-1)


count_down(4)
n 4
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
count_down(3)
n 3
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
count_down(2)
n 2
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
Base case reached!
>>> val = count_down(4
4
3
2
1
countDown(1)
n 1
if n == 1:
print(n)
else:
print(n)
count_down(n-1)



count_down(4)
n 4
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
count_down(3)
n 3
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
countDown(2)
n 2
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
Base case reached!
>>> val = count_down(4
4
3
2
1
countDown(1)
n 1
if n == 1:
print(n)
else:
print(n)
count_down(n-1)


count_down(4)
n 4
if n == 1:
print(n)
else:
print(n)
count_down(n−1)
countDown(3)
n 3
if n == 1:
print(n)
else:
print(n)
count_down(n−1)
countDown(2)
n 2
if n == 1:
print(n)
else:
print(n)
count_down(n−1)
Base case reached!
>>> val = count_down(4
4
3
2
1
countDown(1)
n 1
if n == 1:
print(n)
else:
print(n)
count_down(n−1)


countDown(4)
n 4
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
countDown(3)
n 3
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
countDown(2)
n 2
if n == 1:
print(n)
else:
print(n)
count_down(n-1)
Base case reached!
>>> val = count_down(4
4
3
2
1
countDown(1)
n 1
if n == 1:
print(n)
else:
print(n)
count_down(n-1)

TADA!

- Recursive functions may seem like magic at first glance, but they follow from the principles that we've been building all semester.
- It often takes several exposures to recursion before it "clicks", so we'll keep revisiting recursion in the coming lectures
  - Drawing pictures and practicing are two tools that can help
  - Our next lab is a partner lab so you can bounce your ideas off of a classmate and work though recursion stumbles