

CS 134:  
Recursion

# Announcements & Logistics

- **Lab 6 is today/tomorrow, due Wed/Thurs 10 pm**
  - Analyzes supreme court data
  - Uses dictionaries, tuples, sorting with lambda, plotting, CSV files
  - Review relevant lecture material before lab!
- **HW 6** will be posted on Wednesday
- **Labs 4 and 5** will be returned today
- **Midterm** will be returned at the end of class
  - Avg: 82%
  - Please review your exam and come see us with questions!
  - Solutions available on Glow

**Do You Have Any Questions?**

# Last Time: Data Structure Review

- Wrapped up dictionaries and sets and discussed plotting with matplotlib
- Data structure discussion: which to use when
  - List/tuples: when order matters, dictionaries/sets: when order does not matter
- Tuples or lists?
  - Do we need to **add/remove items dynamically**? If yes, use **lists** (they are mutable!); if data stays same (no changes), use **tuples** (more space efficient)
- Lists vs dictionaries?
  - Dictionaries/sets have huge performance benefits over a list as they store "hashes" of elements and thus support **fast look ups/ insertions/ deletes**
  - This is why dictionaries are also referred to as **hash tables** in other programming languages
- If you want to learn about the implementation and trade off various data structures: take **CSCI 136!**

# Where are We Going?

- First half of CSI 34: learning all the **fundamentals of programming**
  - Functions, conditionals, loops, data types
  - Built-in data structures and methods, sorting, plotting
- Looking ahead to the second half: shift in the course to more **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
    - Building our own data types: **linked lists**
  - **How does sorting work**/ what happens under the hood when Python is sorting?
  - Understanding the basics behind efficient vs inefficient code



# Today's Plan: Intro To Recursion

- What is **recursion**?
- Translating recursive ideas into recursive programs
- Examining the relation 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

# Recursion In Art and Pop Culture

- You're already familiar with the idea of recursion, whether you've referred to it by that name or not!
- The Droste effect was one of the first explicit uses of recursion in an advertising medium in 1904
- The cocoa tin shows an image of a woman holding a platter with a tin that has an image of the same woman holding platter with a tin that has an image of...



# Recursion In Art and Pop Culture

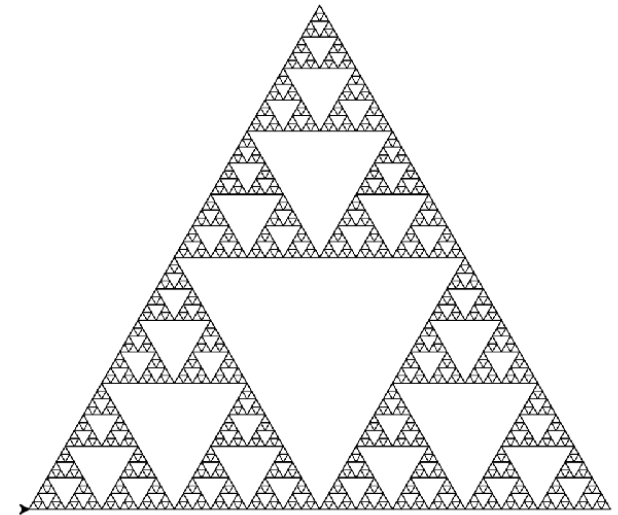
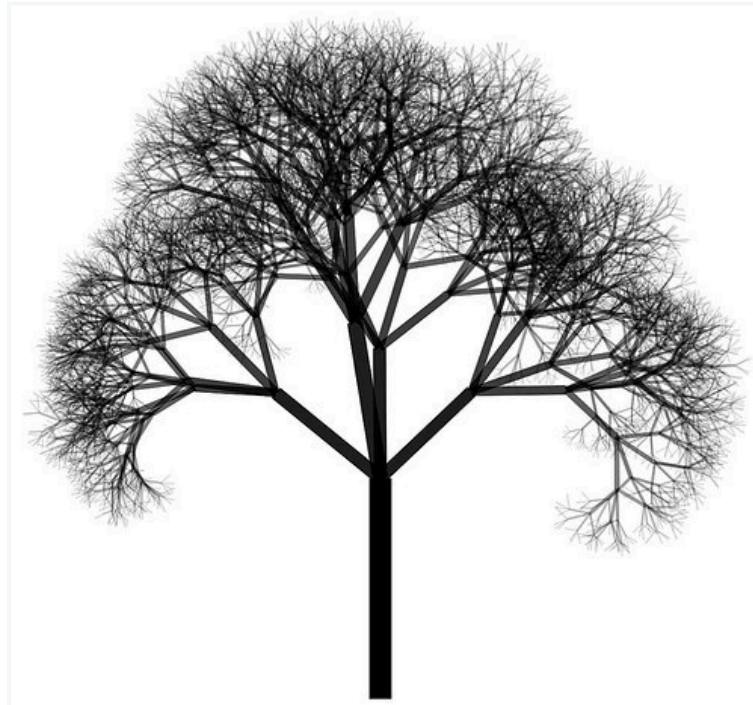
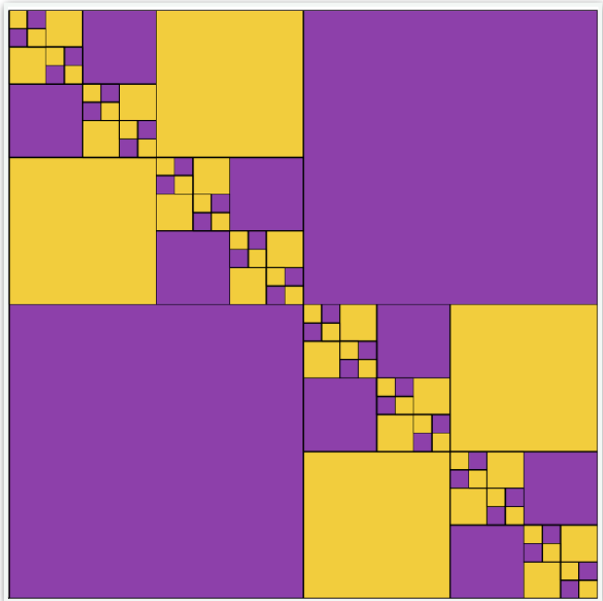
- Computer scientists were of course writing nerdy poems about recursion long before it was cool.

Great fleas have little fleas upon their backs to bite 'em,  
And little fleas have lesser fleas, and so ad infinitum.  
And the great fleas themselves, in turn, have greater fleas to go on;  
While these again have greater still, and greater still, and so on.

— *Siphonaptera, A Budget Of Paradoxes*  
by Augustus De Morgan (1874)

# Why Learn About Recursion?

- Recursion is an important problem solving paradigm that can not only lead to *elegant* code, it can also be used to do really cool things.
- By the end of the week, you'll be able to use recursion to draw these beautiful pictures



# So What Is Recursion?

- The easiest way to understand recursion is to first see examples of it
- Let's start by examining a familiar recursive definition in mathematics
- The set of natural numbers can be defined as follows:
  - $0$  is a natural number
  - If  $n$  is a natural number, then  $n+1$  is a natural number
- Building blocks of a recursive idea:
  1. **Base case(s)**: E.g.,  $0$  is a natural number
  2. **Recursive rule(s)**: E.g., if  $n$  is a natural number, then  $n+1$  is a natural number

# Exercise: Forming Base Case & Recursive Rules

- How would you define the concept of exponentiation  $a^n$  as a base case and a recursive rule (assuming  $n \geq 0$ )
- A recursive definition:
  - **Base case:**  $a^0 = 1$
  - **Recursive rule:**  $a^n = a * a^{n-1}$

# Exercise: Forming Base Case & Recursive Rules

- Similarly, how would you define the concept of factorial  $n!$  as a base case and a recursive rule (assuming  $n \geq 0$ )
- A recursive definition:
  - **Base case:**  $0! = 1$
  - **Recursive rule:**  $n! = n * (n-1)!$

# Exercise: Forming Base Case & Recursive Rules

- Let's examine a more complicated series known as the Fibonacci sequence.
- The Fibonacci sequence is a series of numbers that starts with **0** and **1**, and where each successive number is the sum of the two preceding ones
  - **0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ..**
- A recursive definition:
  - **Base cases:**  $F_0 = 0$  and  $F_1 = 1$
  - **Recursive rule:**  $F_n = F_{n-1} + F_{n-2}$



# Translating Recursive Ideas To Programs

- The beauty of recursion is that once you've written down your recursive idea, the programming part comes relatively easy
- Ideally, you spend more time with pen and paper and front-load all your thinking into coming up with an appropriate base case and recursive rule
- Once you have these two ingredients, the implementation of recursive programs is fairly formulaic

```
def recursiveProgram(inputs):  
    # if inputs correspond to base case apply base case rules  
    # else apply recursive rule
```

# Translating Recursive Ideas To Programs

- Recursive definition for  $a^n$ :
  - **Base case:**  $a^0 = 1$
  - **Recursive rule:**  $a^n = a * a^{n-1}$

```
def power(a, n):  
    """  
    Returns a^n. Assumes n >= 0.  
    """  
    if n == 0:  
        return 1  
    else:  
        return a * power(a, n-1)
```

```
print(power(5, 0))  
print(power(5, 4))
```

1

625

# Translating Recursive Ideas To Programs

- Recursive definition for Fibonacci:
  - **Base cases:**  $F_0 = 0$ ,  $F_1 = 1$
  - **Recursion:**  $F_n = F_{n-1} + F_{n-2}$

```
def fibonacci(n):  
    """  
    Returns nth Fibonacci number  
    """  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

```
print(fibonacci(5))  
print(fibonacci(6))  
print(fibonacci(7))
```

```
5  
8  
13
```

# Recursive Functions

- We have seen many examples of functions calling other functions
- A **recursive function** is a function that **calls itself**
- Similar to recursive definitions, all recursive functions consist of one or more base cases and a set of recursive rules that successively simplify the problem **until we approach one of the base cases**
- It's important that the recursive rule eventually takes you towards one of the base cases, else we end up with the recursion equivalent of an infinite loop
- We will compare recursive implementations to iterative implementations in the next lecture, but for now let's see what infinite recursion looks like and take a deeper look into how recursion works

# Infinite Recursion

- Recursive definition for  $a^n$ :
  - **Base case:**  $a^0 = 1$
  - **Recursive rule:**  $a^n = a * a^{n-1}$

```
def infinitePower(a, n):  
    """  
    Returns a^n  
    """  
    if n == 0:  
        return 1  
    else:  
        return a * infinitePower(a, n)
```

```
print(infinitePower(5, 4))
```

- This gives us the message **RecursionError: maximum depth exceeded in comparison** — notice we are no longer simplifying the problem in our recursive rule.
- What does this error mean?
- So far, we've simply believed in the magic of recursion (sometimes folks even explicitly make reference to the existence of a recursion fairy) but let's take a closer look at what goes on in recursive function calls.

# 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:**  $n = 0$ , do nothing
  - **Recursive rule:** `print(n), call countdown(n-1)`

# Understanding Recursive Functions

- Recursive definition of countdown:
  - **Base case:**  $n = 0$ , do nothing
  - **Recursive rule:** `print(n)`, call `countDown(n-1)`

```
def countDown(n):  
    '''Prints numbers from n down to 1'''  
    if n < 1: # Base case  
        pass # Do nothing  
    else: # Recursive case: n >= 1:  
        print(n)  
        countDown(n-1)
```

```
countDown(5)
```

```
5  
4  
3  
2  
1
```

# Side Note: Implicit Base Case

- It is possible to simplify our function by omitting the base case
- The following two versions are **equivalent**
- However, in recursion, **we prefer that you write the base case explicitly** for pedagogical reasons (for now)
  - Bad things happen if you forget the base case.... (we'll see)

```
def countdown(n):  
    '''Version 1'''  
    if n < 1:  
        pass # do nothing  
    else:  
        print(n)  
        countdown(n-1)
```

```
def countdown(n):  
    '''Version 2'''  
    if n > 0:  
        print(n)  
        countdown(n-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**

# Review: Function Frame Model

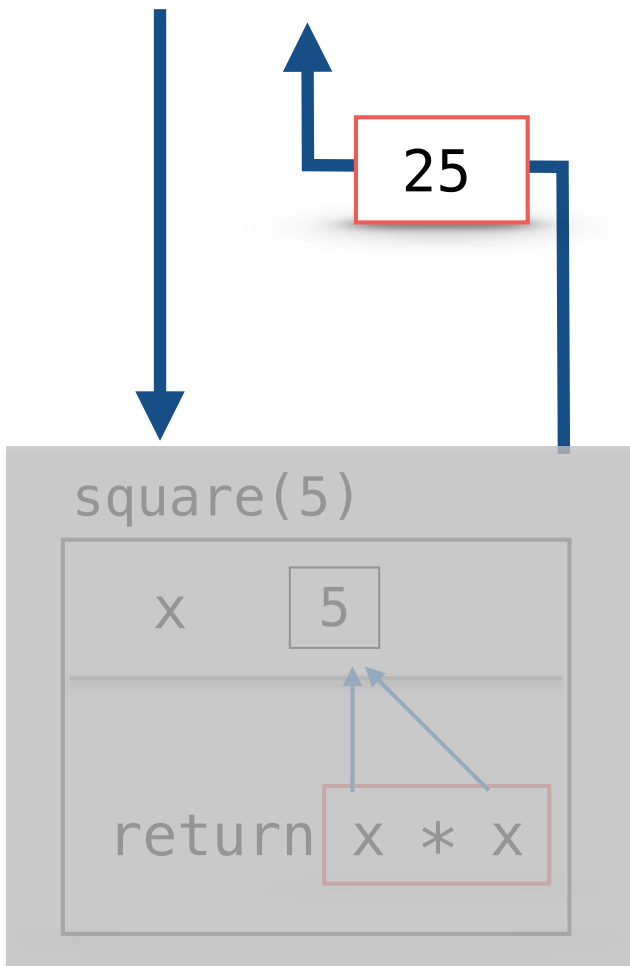
- Consider a simple function `square`
- What happens when `square(5)` is invoked?

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

# Review:

## Function Frame Model

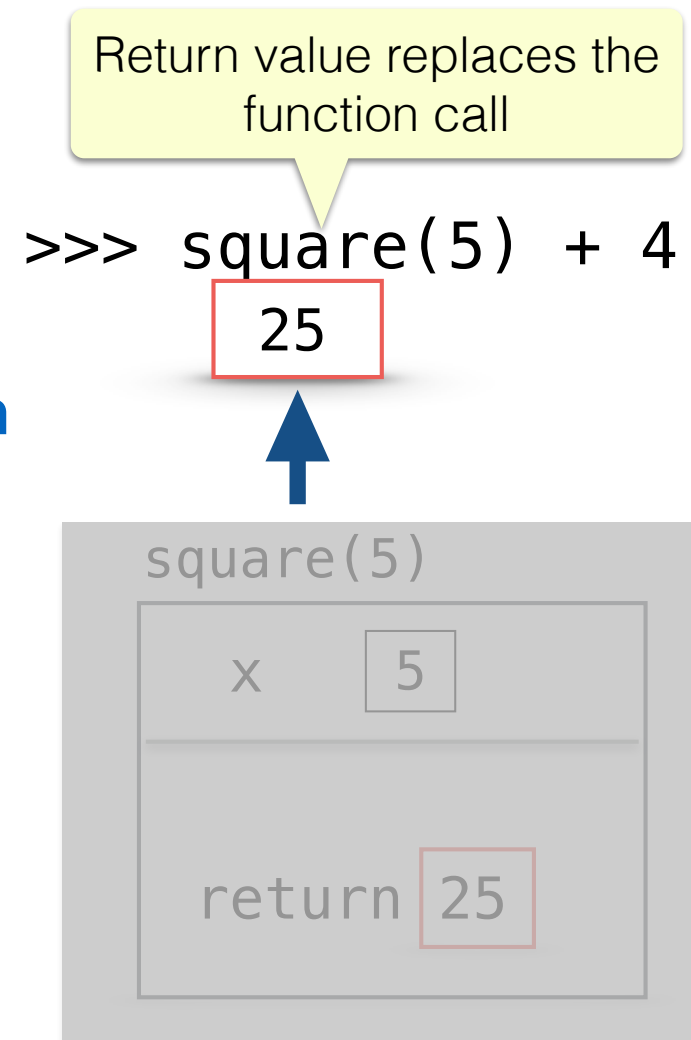
```
>>> square(5)
```



# 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 sumSquare(a, b):  
    return square(a) + square(b)
```

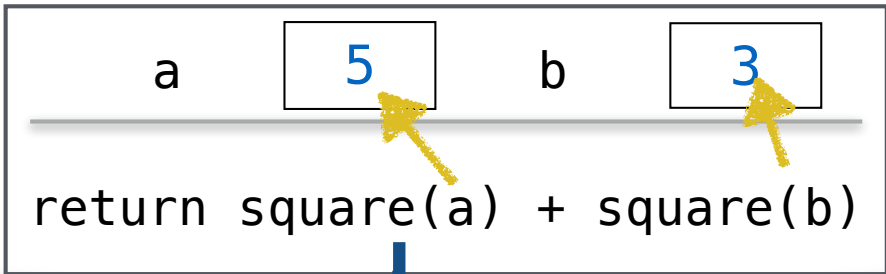
- What happens when we call `sumSquare(5, 3)`?

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

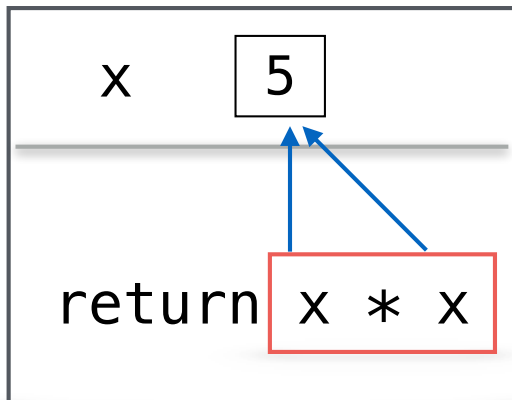
```
>>> sumSquare(5,3)
```



```
sumSquare(5, 3)
```



```
square(5)
```

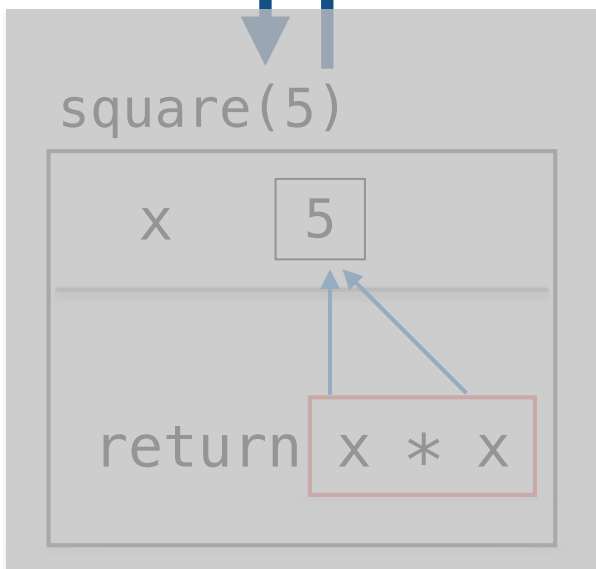
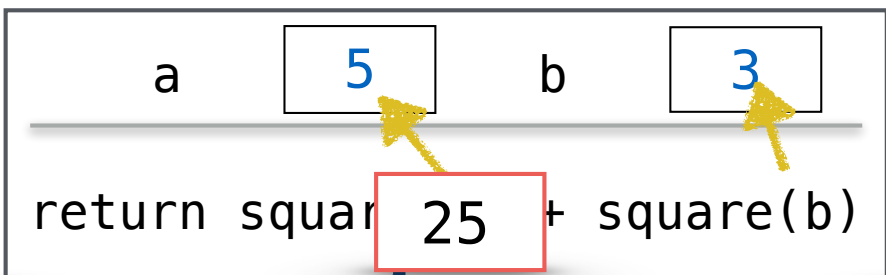


```
def sumSquare(a, b):  
    return square(a) + square(b)
```

>>> sumSquare(5,3)



**sumSquare(5, 3)**

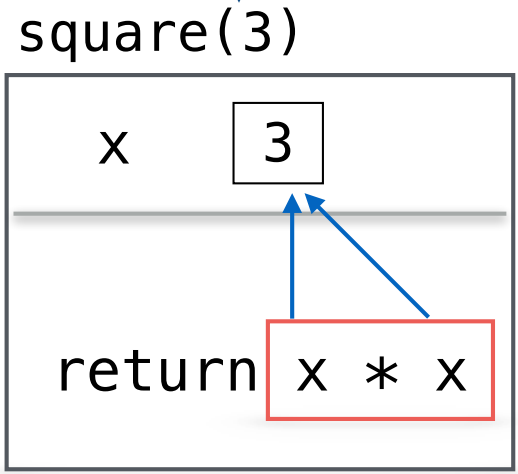
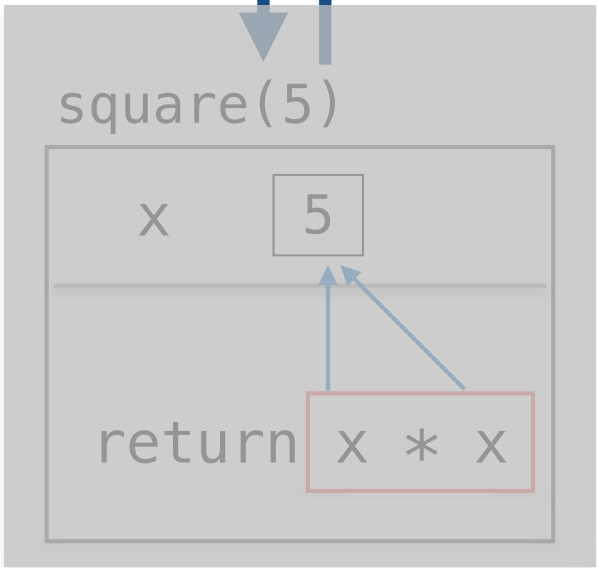
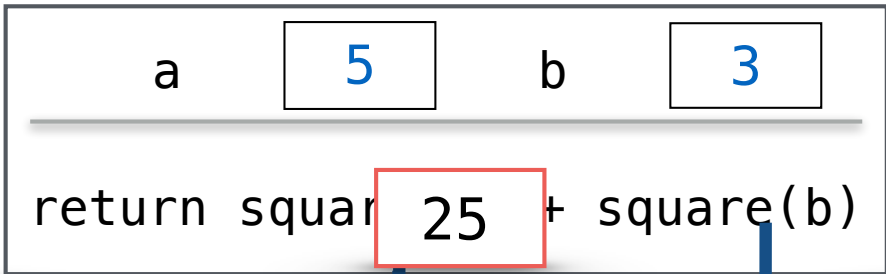


```
def sumSquare(a, b):  
    return square(a) + square(b)
```

```
>>> sumSquare(5,3)
```



**sumSquare(5, 3)**



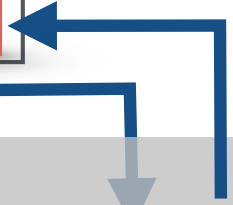
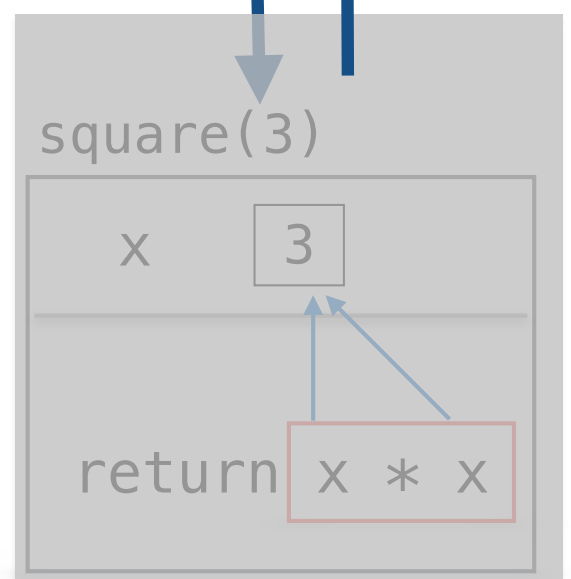
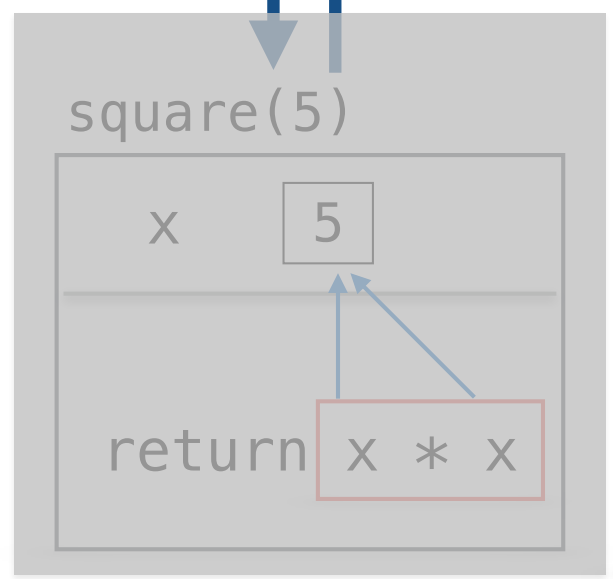
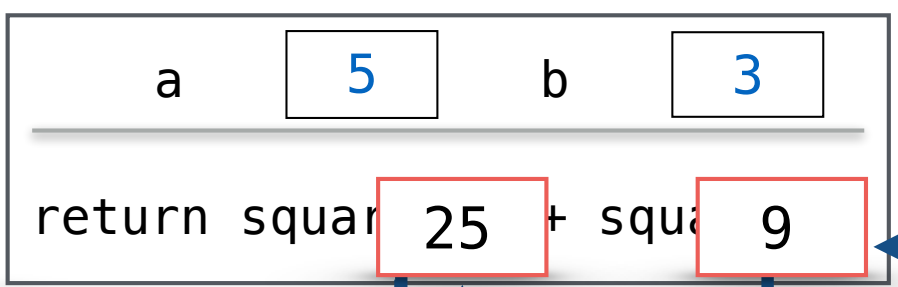


```
def sumSquare(a, b):  
    return square(a) + square(b)
```

```
>>> sumSquare(5,3)
```

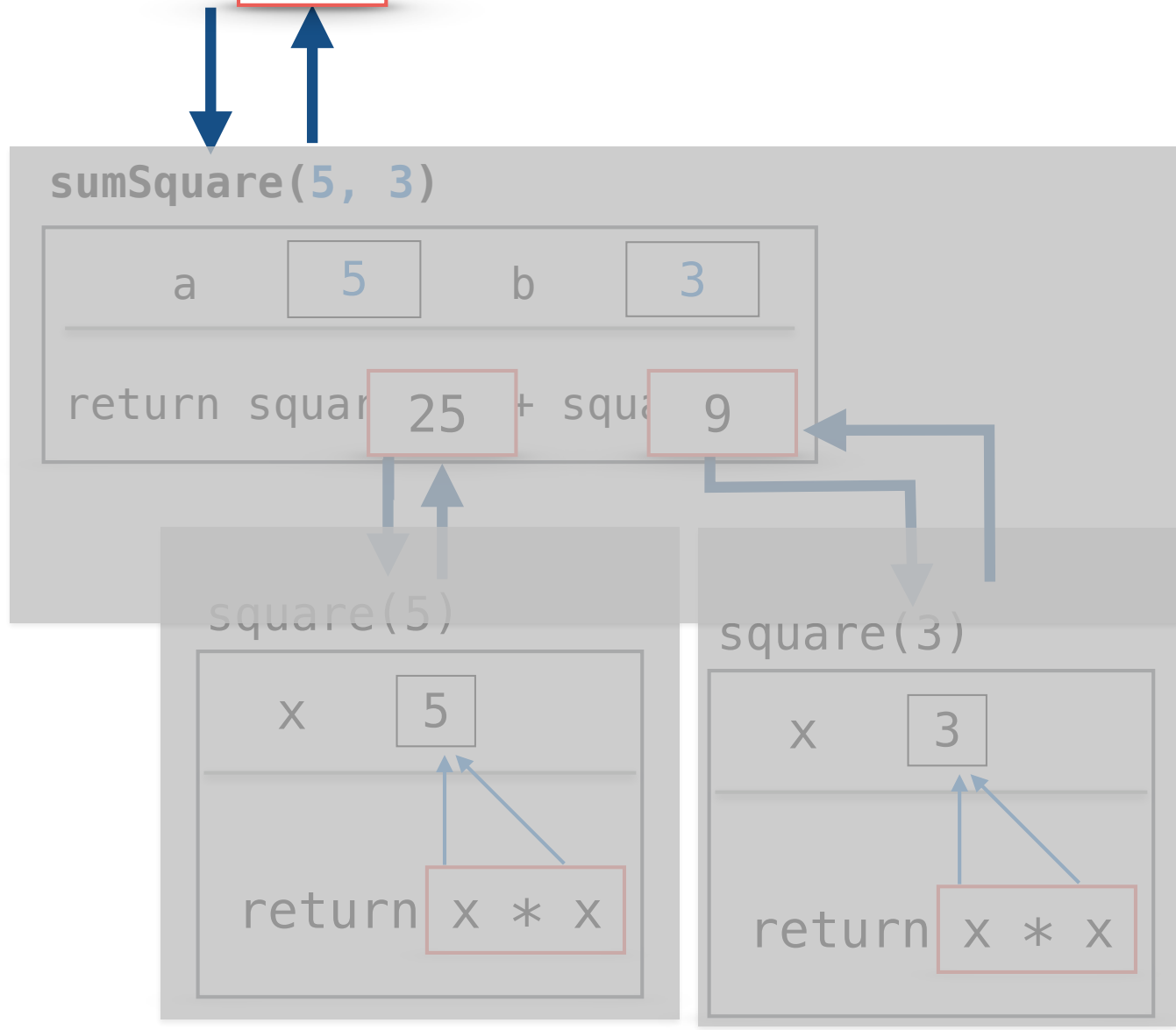


**sumSquare(5, 3)**



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

```
>>> sumSquare(5, 3)
```



Function Frame Model to  
Understand **countDown**

```
def countDown(n):  
    '''Prints ints from n down to 1'''  
    if n < 1:  
        pass # do nothing  
    else:  
        print(n)  
        countDown(n-1)
```

```
>>> countDown(5)
```

```
5  
4  
3  
2  
1
```

```
>>> countDown(4)
```

```
4  
3  
2  
1
```

### countDown(3)

```
n 3
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

### countDown(2)

```
n 2
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

### countDown(1)

```
n 1
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

Base case reached!

```
>>> countDown(3)
```

```
3
2
1
```

### countDown(0)

```
n 0
-----
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)
```

Implicit return

### countDown(3)

```
n 3
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

### countDown(2)

```
n 2
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

### countDown(1)

```
n 1
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

Base case reached!

```
>>> countDown(3)
```

```
3
2
1
```

### countDown(0)

```
n 0
-----
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)
```

Implicit return

### countDown(3)

```
n 3
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

### countDown(2)

```
n 2
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

### countDown(1)

```
n 1
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

Implicit return

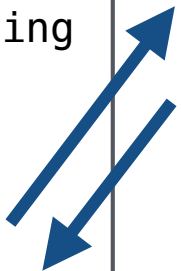
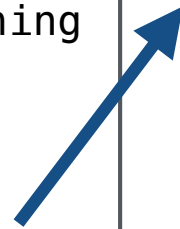
Base case reached!

```
>>> countDown(3)
3
2
1
```

### countDown(0)

```
n 0
-----
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)
```

Implicit return



### countDown(3)

```
n 3
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

### countDown(2)

```
n 2
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

Implicit return

### countDown(1)

```
n 1
-----
if n < 1:
    pass # do nothing
else:
    → print(n)
    countDown(n-1)
```

Implicit return

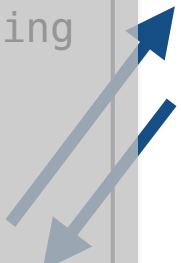
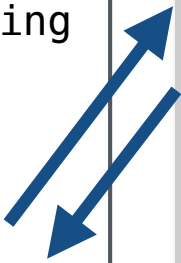
Base case reached!

```
>>> countDown(3)
3
2
1
```

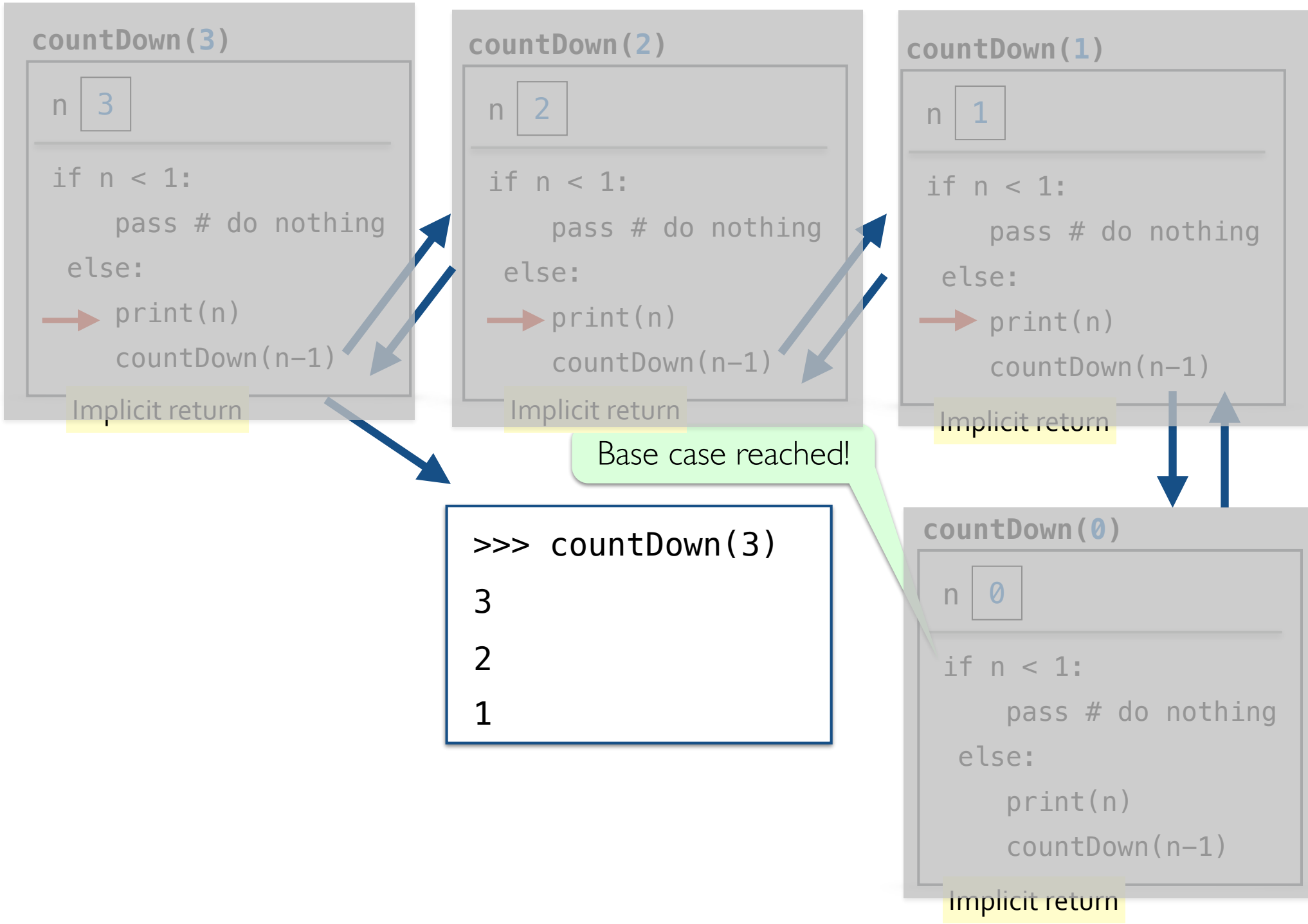
### countDown(0)

```
n 0
-----
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)
```

Implicit return









More Recursion: **countUp**

# countUp(n)

- Write a recursive function that prints integers from **1** up to **n**
- Recursive definition of countUp:
  - **Base case:**  $n = 0$ , do nothing
  - **Recursive rule:** call `countUp(n-1)`, `print(n)`

```
>>> countUp(5)
```

```
1  
2  
3  
4  
5
```

```
>>> countUp(4)
```

```
1  
2  
3  
4
```

```
>>> countUp(3)
```

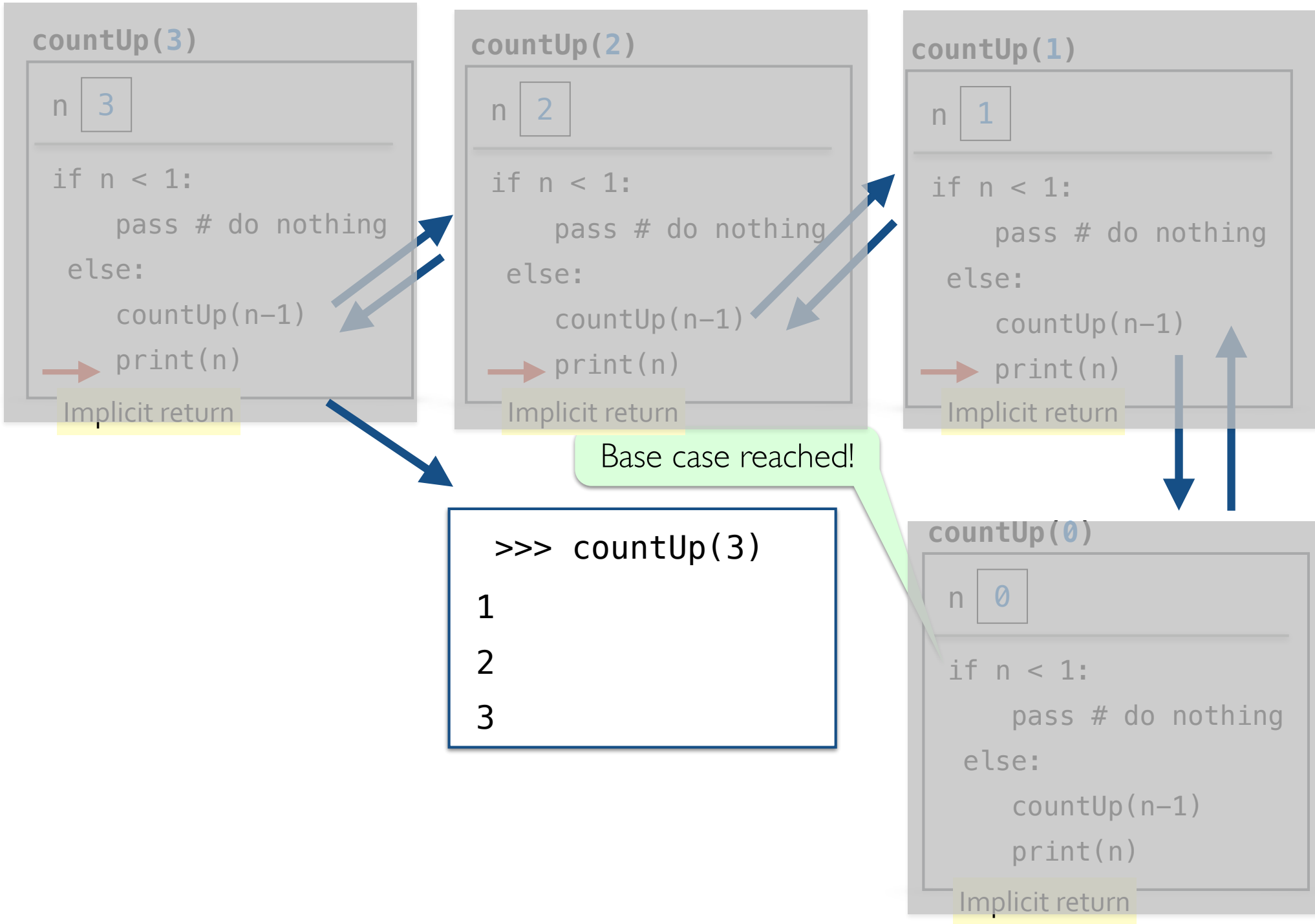
```
1  
2  
3
```

# countUp(n)

- Note that unlike `countDown(n)` we moved our print statement to be **after** the recursive function call
- By printing after the recursive call, the print statement gets executed “on the way back” from recursive calls

```
def countUp(n):  
    '''Prints out integers from 1 up to n'''  
    if n < 1:  
        pass # do nothing  
    else:  
        countUp(n-1)  
        print(n)
```

# Function Frame Model to Understand `countUp`



# Helpful Exercises

- It may be helpful to revisit the **power** and **fibonacci** functions we implemented earlier in the lecture and try to build function frame models for them
- You can also implement these functions in the website below, and visualize the execution (the program builds the function frame models for you step-by-step.)
  - <https://pythontutor.com/visualize.html#mode=edit>
- Remember to also review the examples in the Jupyter notebook — some of them examples involve printing pretty patterns!



Recursion GOTCHAs!

# GOTCHA # 1

- If the problem that you are solving recursively **is not getting smaller**, that is, you are not getting closer to the base case --- **infinite recursion!**
- Never reaches the base case


```
def countdownGotcha(n):  
    '''Prints ints from n down to 1'''  
    if n < 1:  
        pass # do nothing  
    else:  
        print(n)  
        countdownGotcha(n)
```

Subproblem not getting smaller!

# GOTCHA #2

- Missing base case/ unreachable base case--- another way to cause **infinite recursion!**

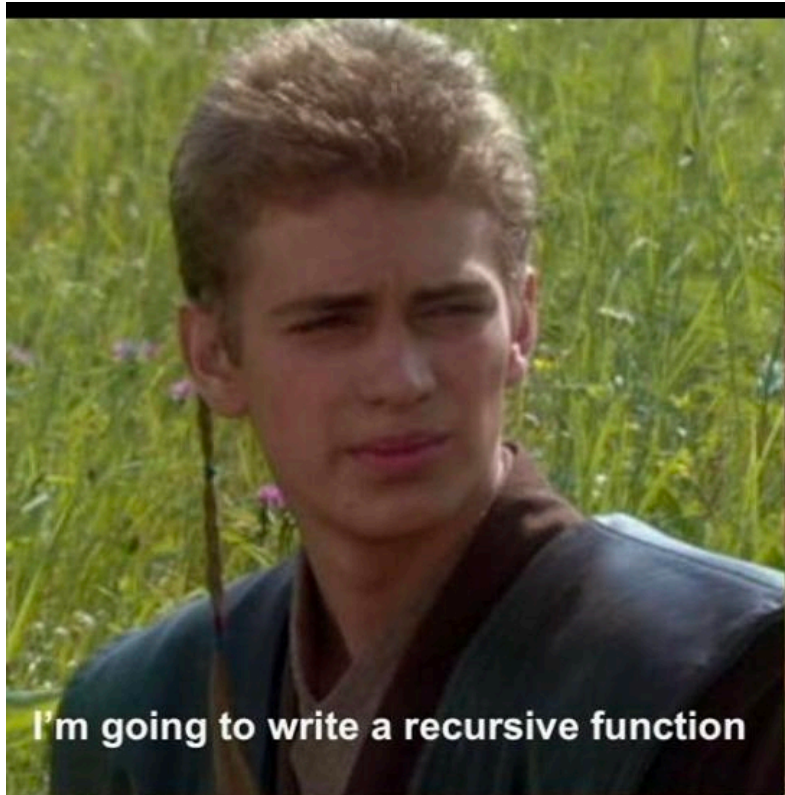
```
def printHalvesGotcha(n):  
    if n > 0:  
        print(n)  
        printHalvesGotcha(n/2)
```



Always true!

# "Maximum recursion depth exceeded"

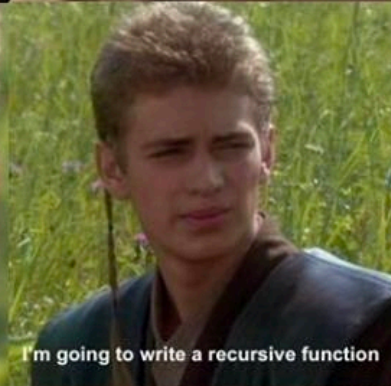
- In practice, the infinite recursion examples will terminate when Python runs out of resources for creating function call frames, leads to a "maximum recursion depth exceeded" error message



I'm going to write a recursive function



With a base case, right?



I'm going to write a recursive function



With a base case, right?



I'm going to write a recursive function



With a base case, right?

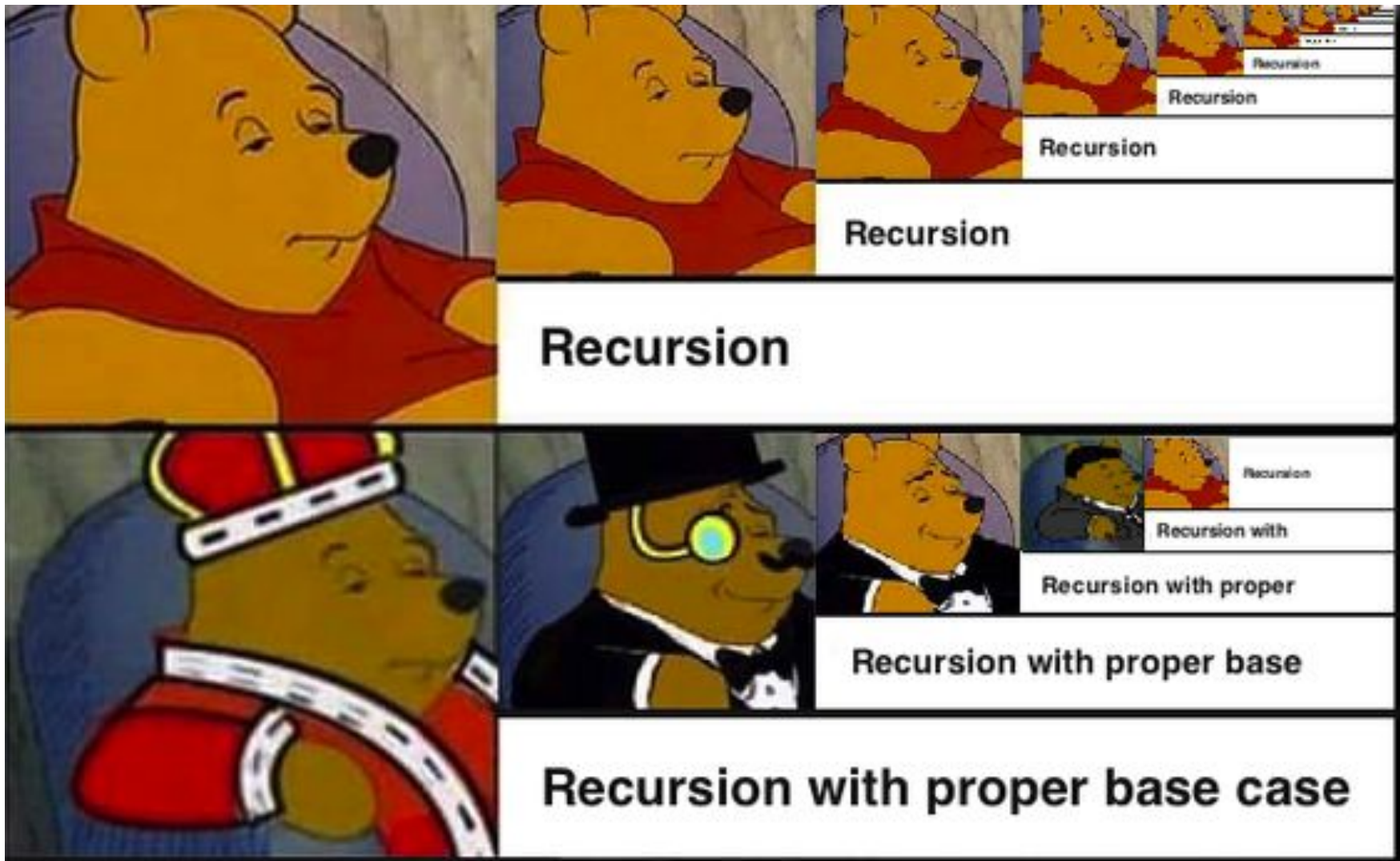


I'm going to write a recursive function



With a base case, right?





# Next Lecture

- Comparing iterative and recursive programs
- Intro to **turtle** module and graphical recursion

