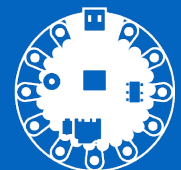
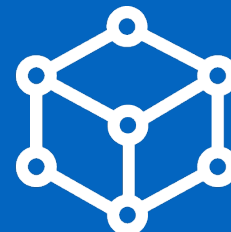
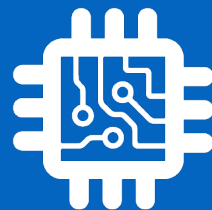
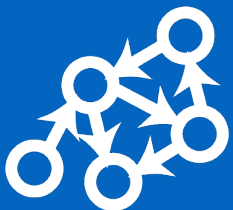


# CSI 34: Recursion



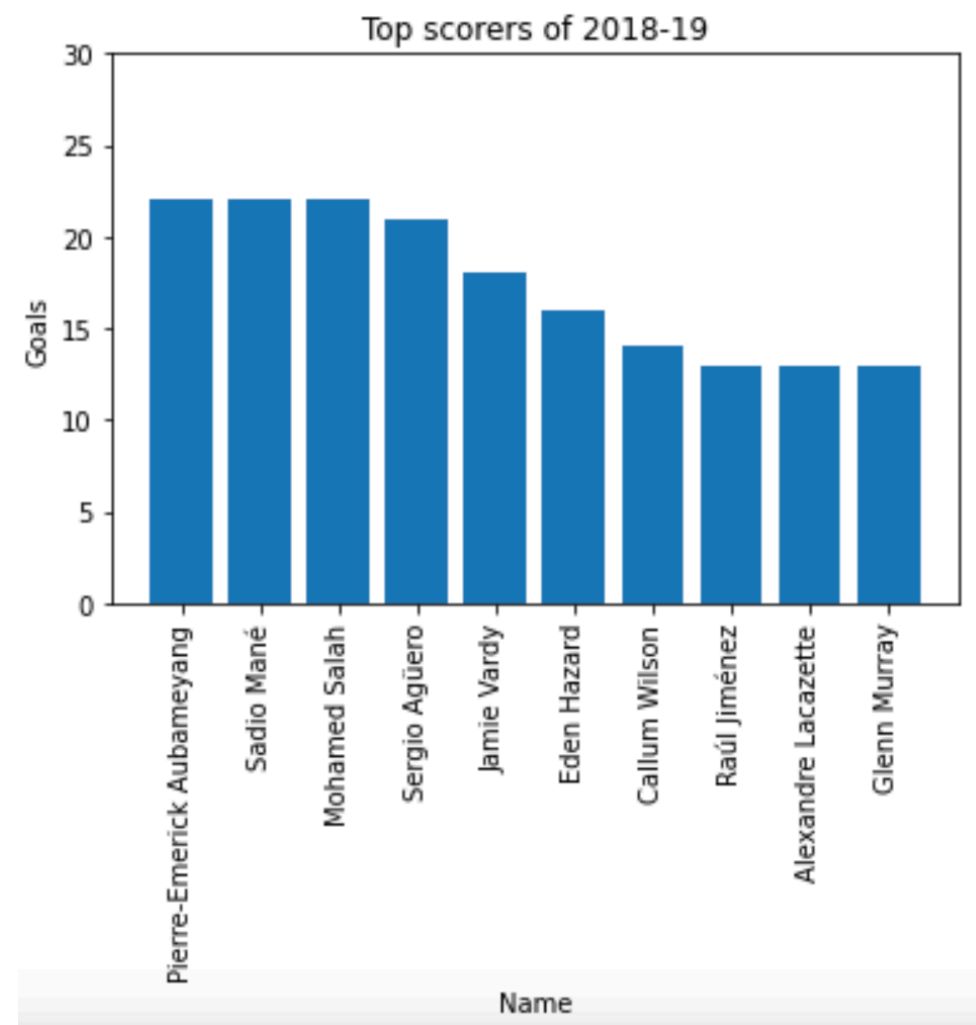
# Announcements & Logistics

- **No HW due tonight :)**
- **Lab 6 due Wed/Thurs at 10 pm**
  - Uses dictionaries, plotting, CSV files

**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
  - Start developing our intuition regarding efficient vs inefficient code

# 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

- You're already familiar with it by that name
- The Droste effect is a recursive advertising method
- The cocoa tin shown here has an image of a woman holding a tray with a cup of cocoa and a tin of Droste cocoa powder

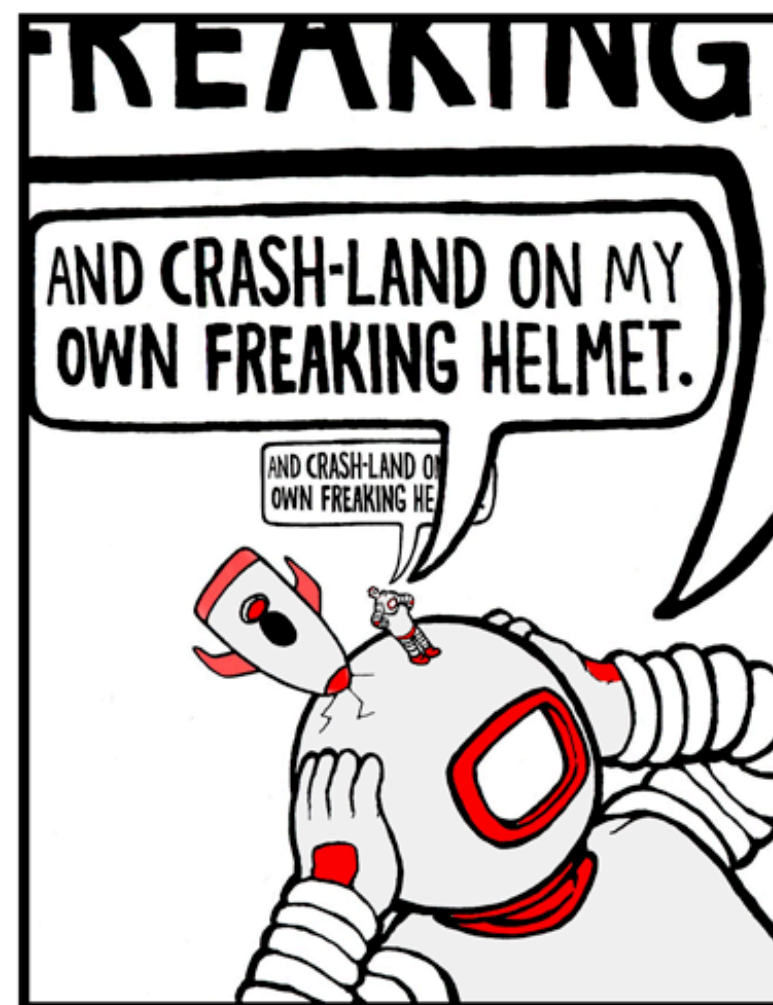


you've referred to

rsion in an

r with a tin that  
that has an image

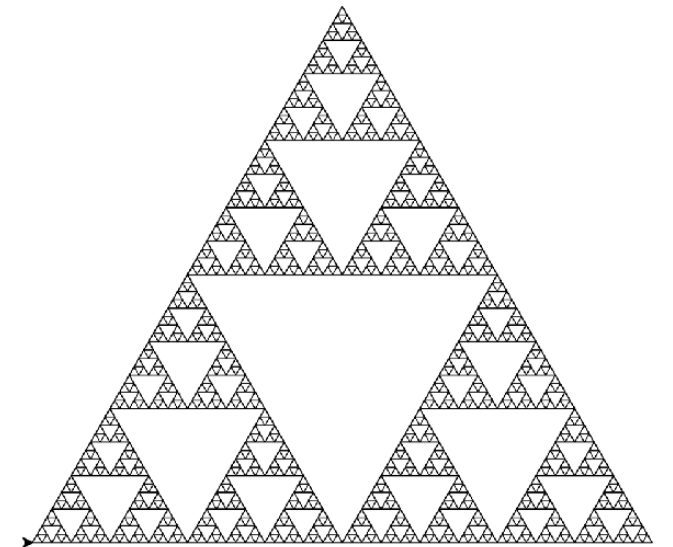
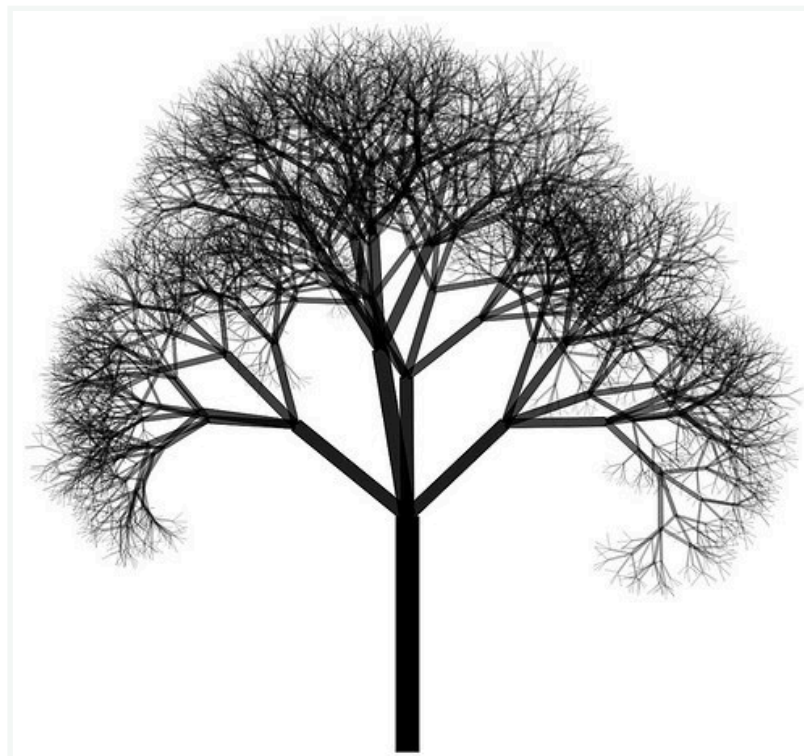
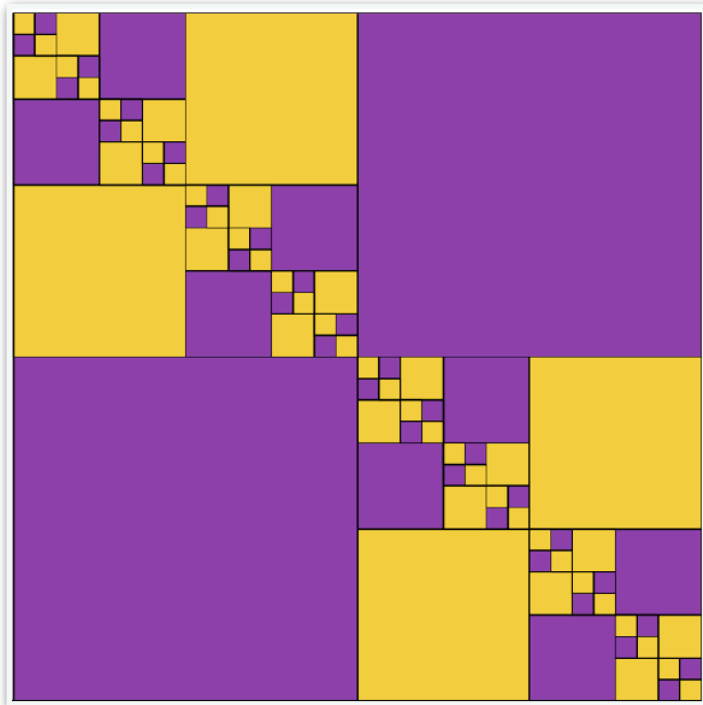






# Why Learn About Recursion?

- 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) :  
    # accumulate 1 on each match  
    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 = 0  
    if elem == lst[0]:  
        first = 1  
  
    # 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?

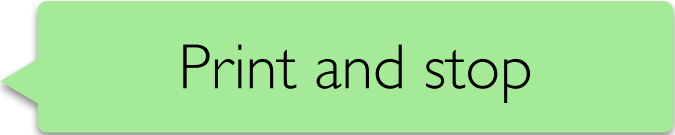

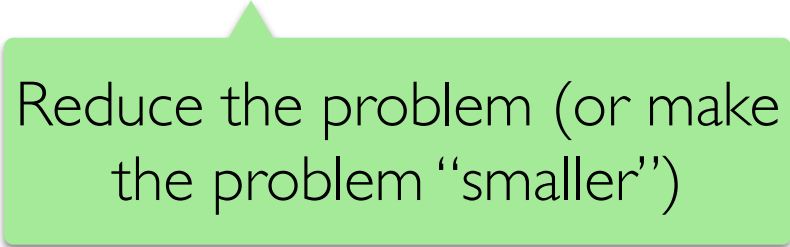
- Surprisingly, this function works!
- Some observations:
  - Some paths through the function **call the same function again**
    - This is what makes the function recursive
  - Other paths through the function (the smallest case that we can solve immediately) simply return the answer
    - This is called a **base case**. Every 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)` 
  - **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)
```

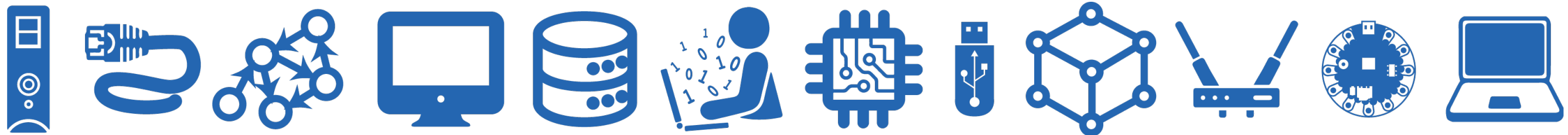
```
5  
4  
3  
2  
1
```

# 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 later where that may not be the case!

# Review: Function Frame Model



# Review: Function Frame Model

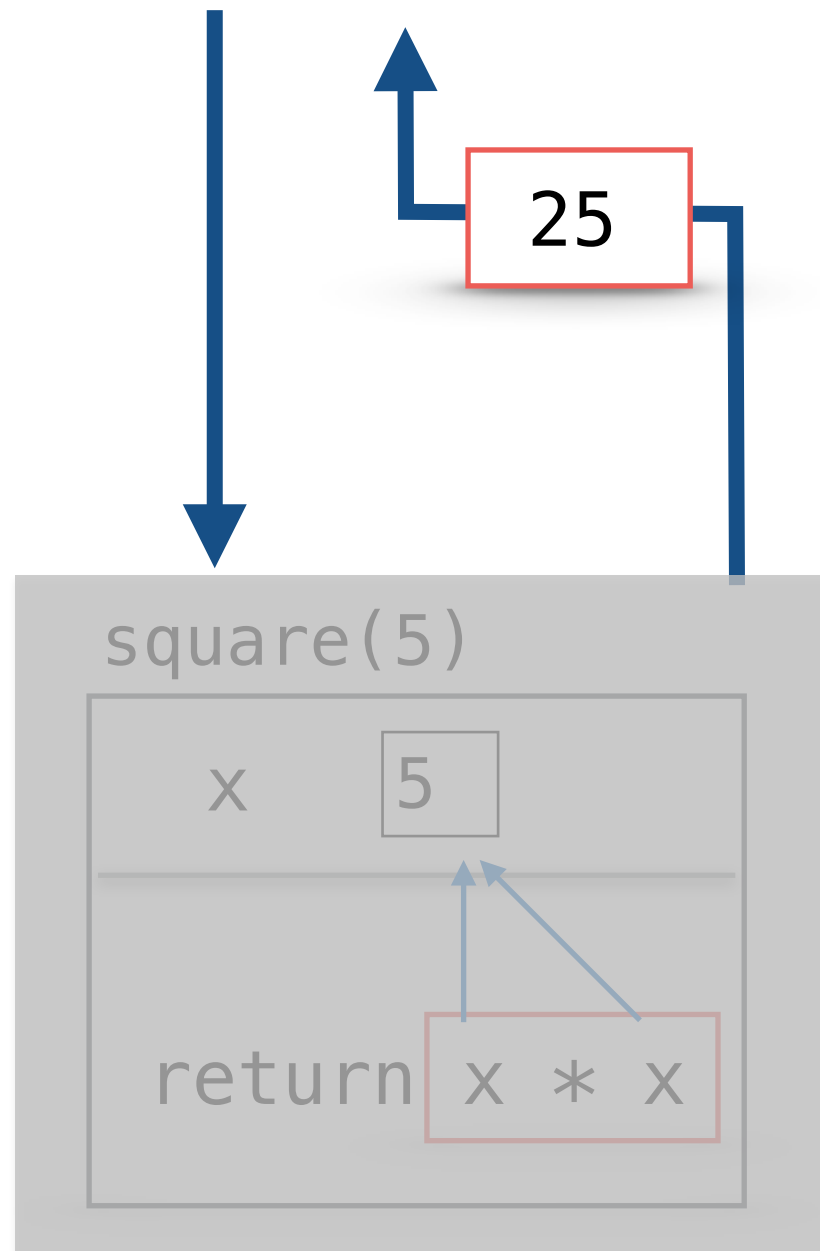
- Consider a simple function **square**
- What happens when **square(5)** is invoked?

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



# Review: Function Frame Model

>>> square(5)



## Summary:

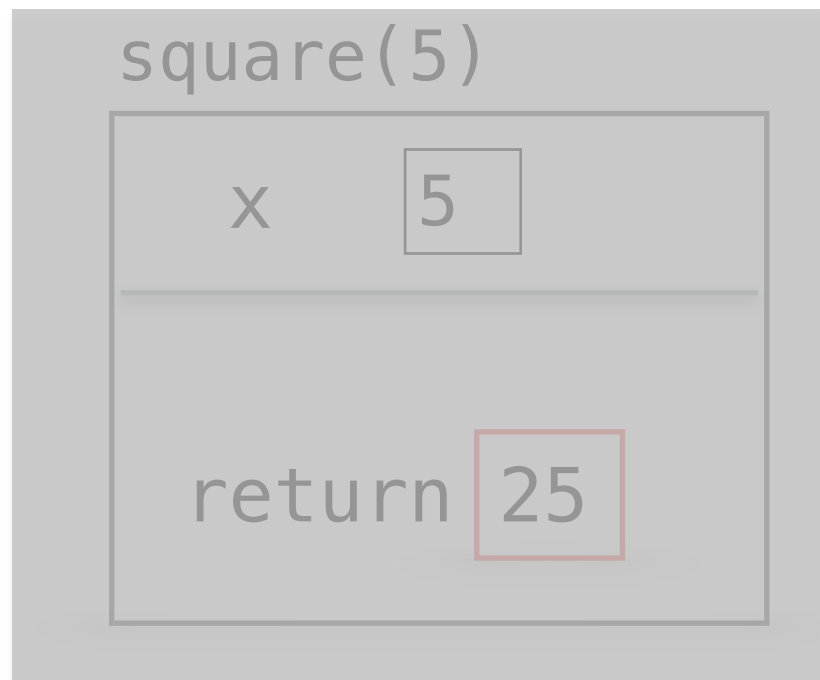
### Function Frame Model

What happens with a complex expression?

Return value replaces the function call

```
>>> square(5) + 4
```

25



- 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)
```

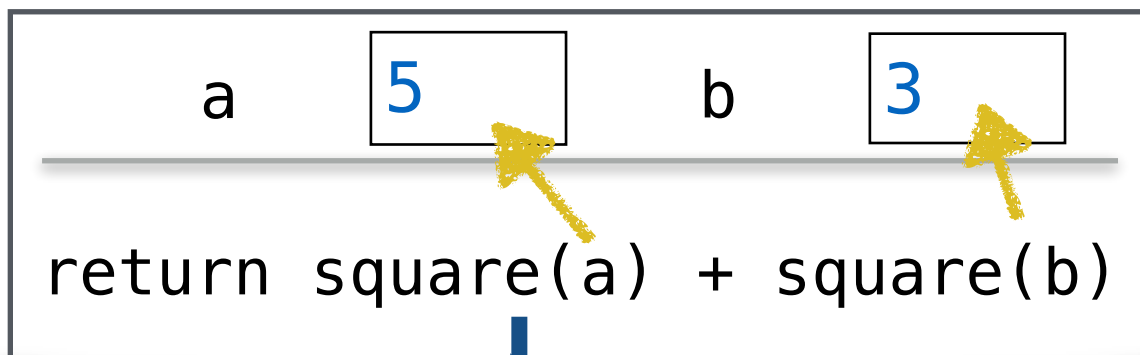
- What happens when we call `sum_square(5, 3)`?

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

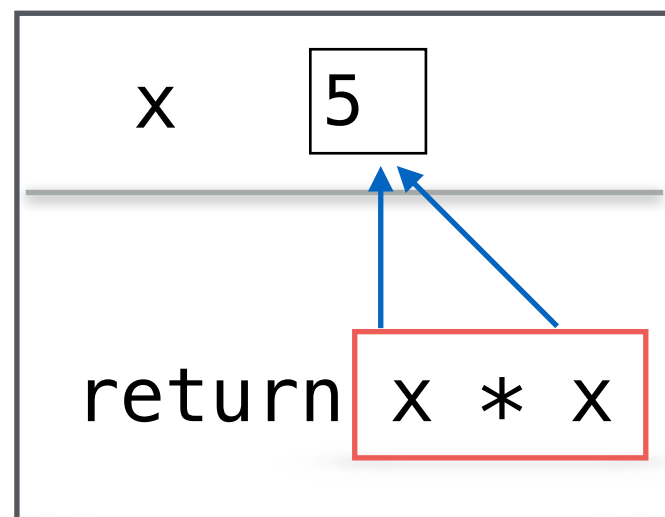
```
>>> sum_square(5,3)
```



**sum\_square(5, 3)**



**square(5)**



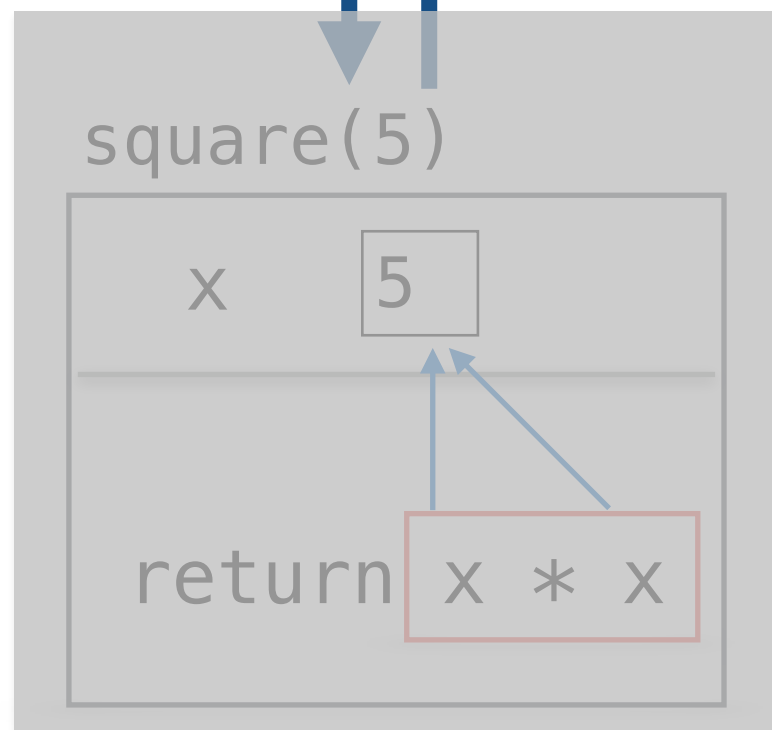
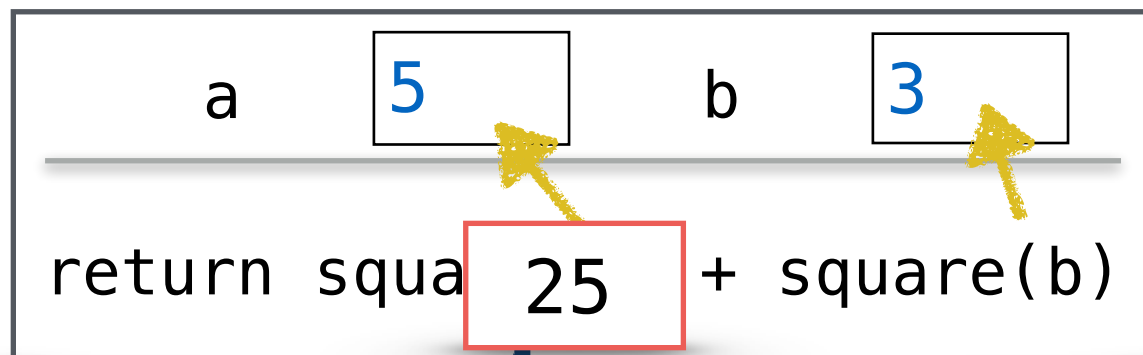


```
def sum_square(a, b):  
    return square(a) + square(b)
```

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



**sum\_square(5, 3)**

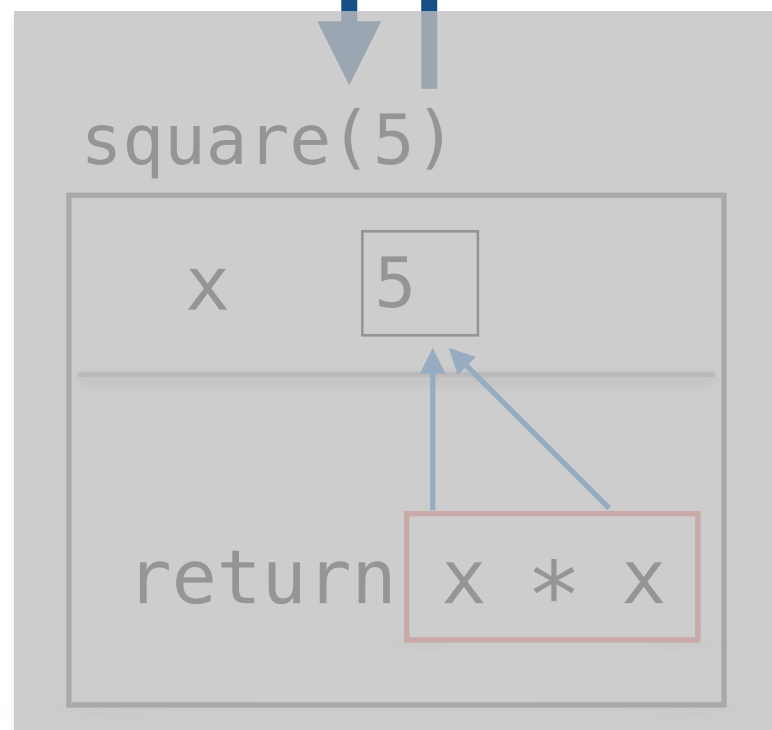
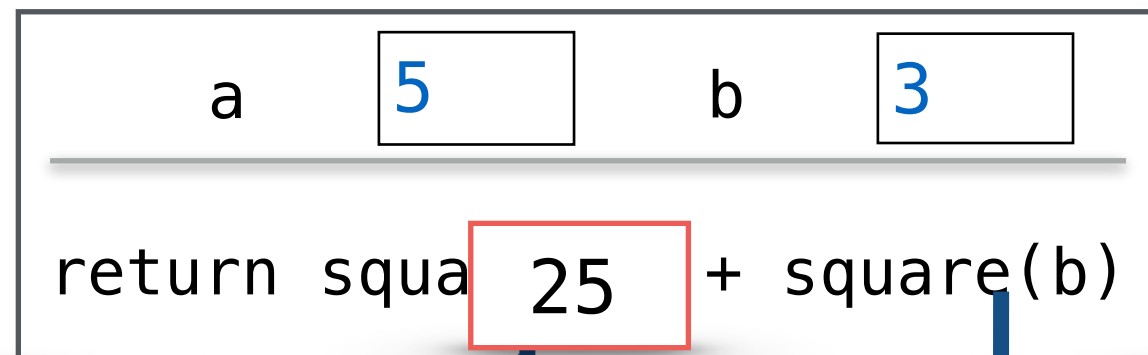


```
def sum_square(a, b):  
    return square(a) + square(b)
```

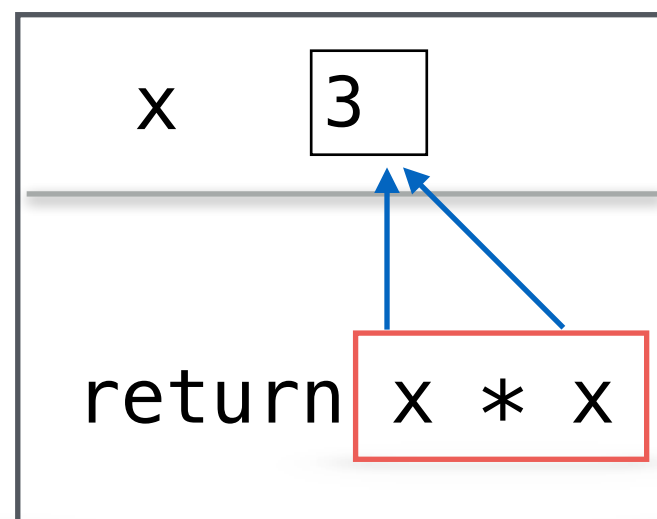
>>> sum\_square(5,3)



**sum\_square(5, 3)**



square(3)

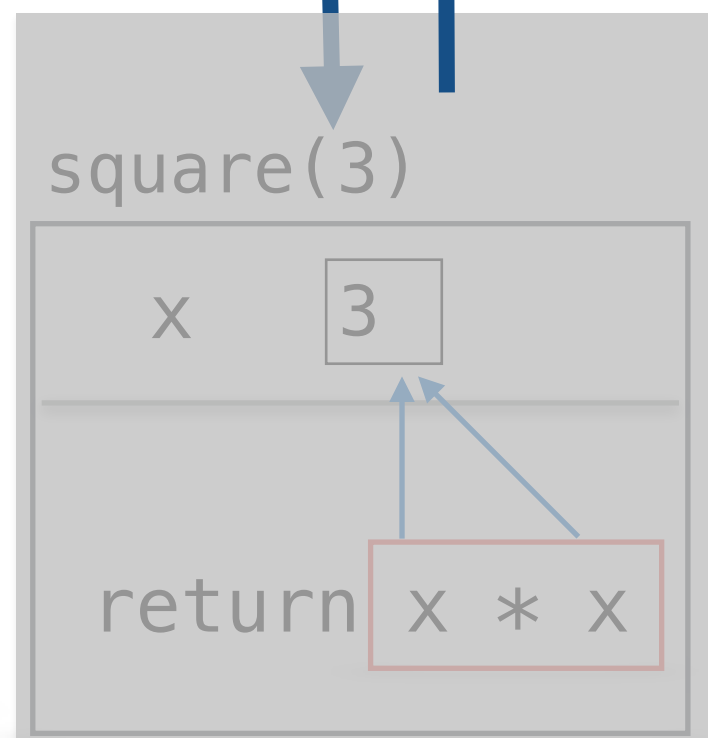
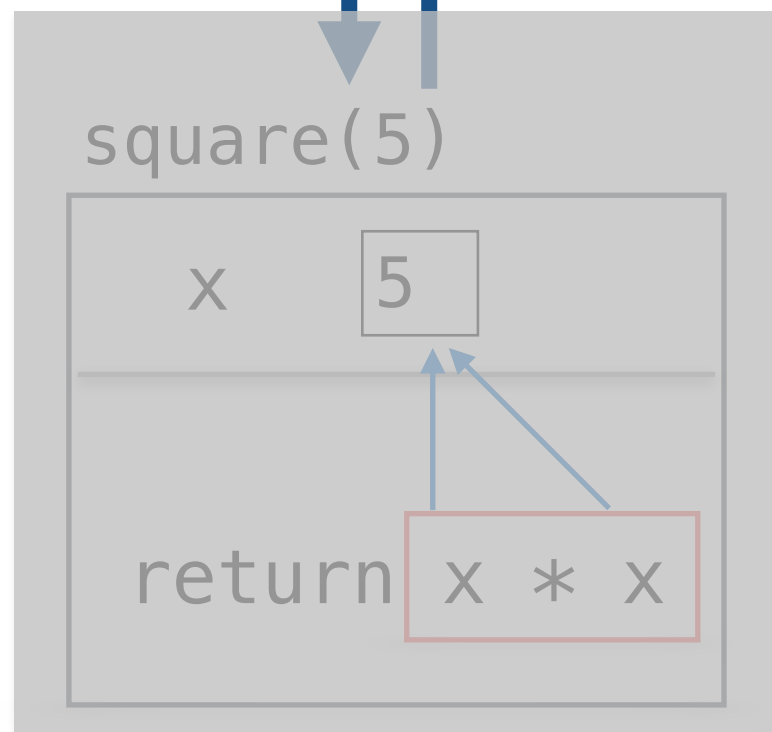
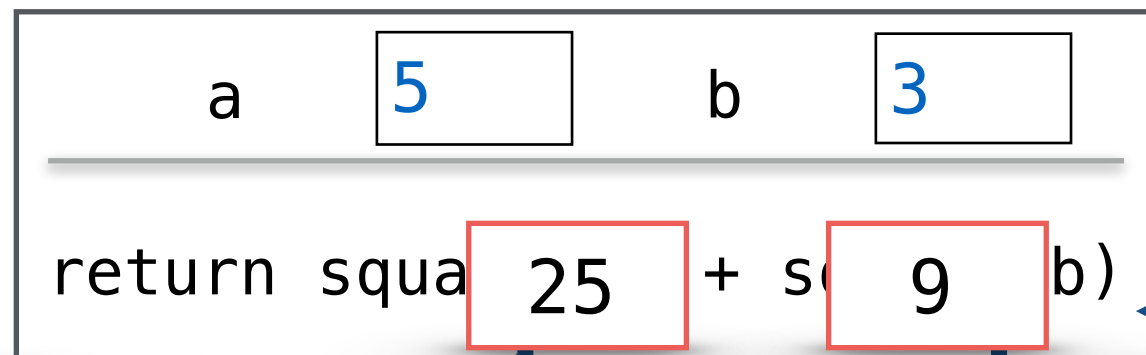


```
def sum_square(a, b):  
    return square(a) + square(b)
```

>>> sum\_square(5,3)



**sum\_square(5, 3)**



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

>>> sum 34 re(5,3)



sum\_square(5, 3)

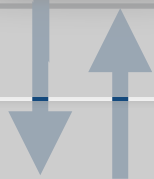
a

5

b

3

return square 25 + square 9 b)



square(5)

x

5

return x \* x

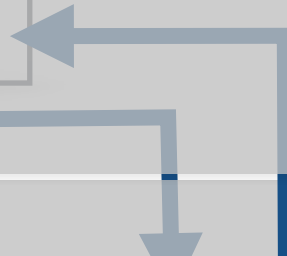
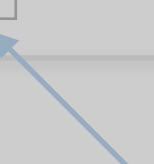


square(3)

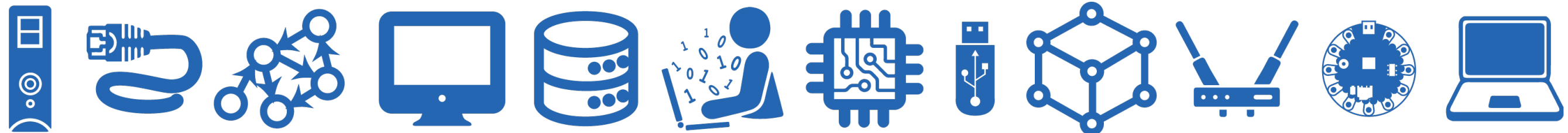
x

3

return x \* x



# Function Frame Model to Understand `count_down`



```
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)
```

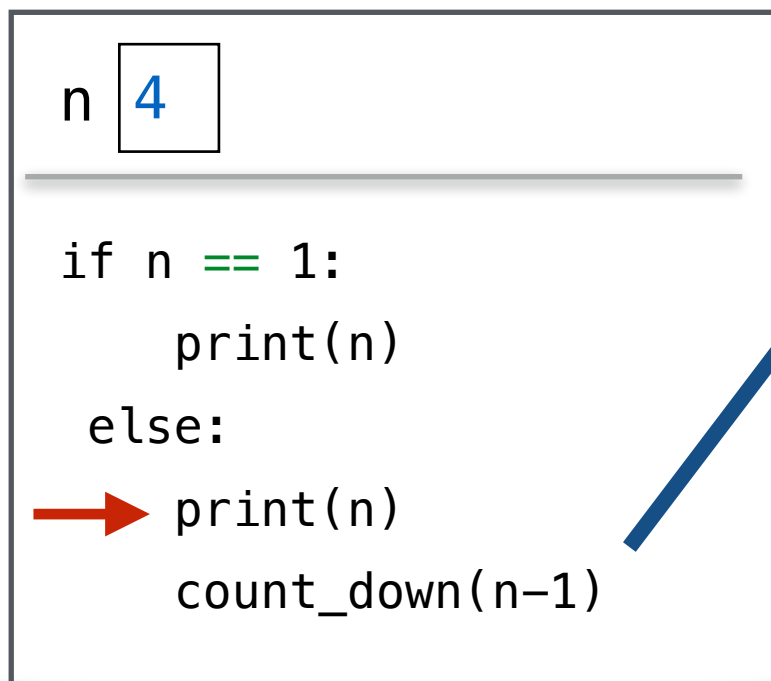
4  
3  
2  
1

**countDown(1)**

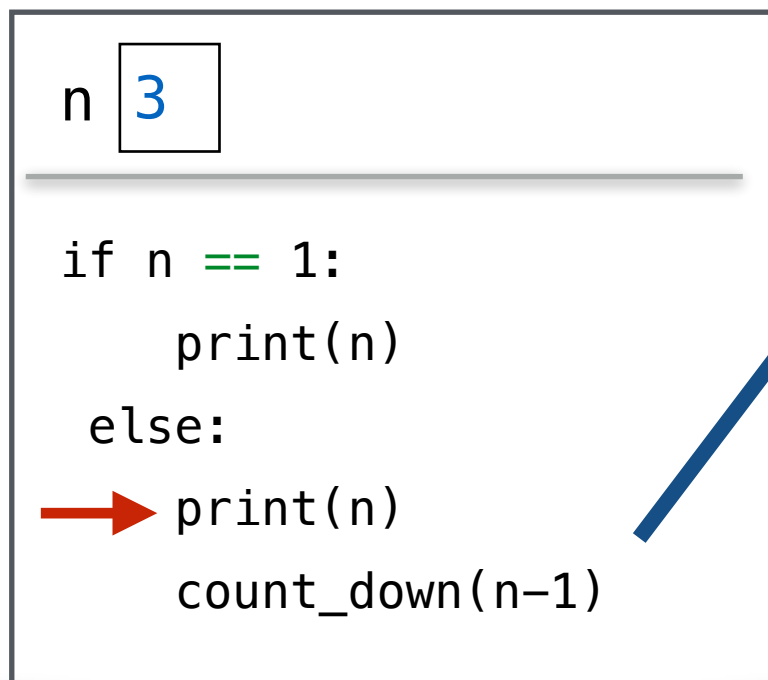
n 1

```
if n == 1:
    print(n)
else:
    print(n)
    count_down(n-1)
```

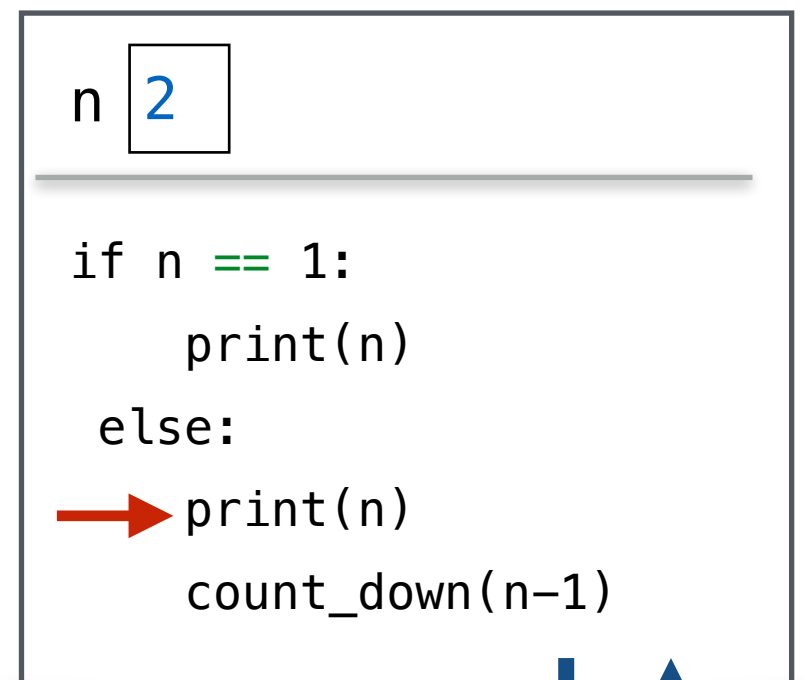
**count\_down(4)**



**count\_down(3)**



**count\_down(2)**

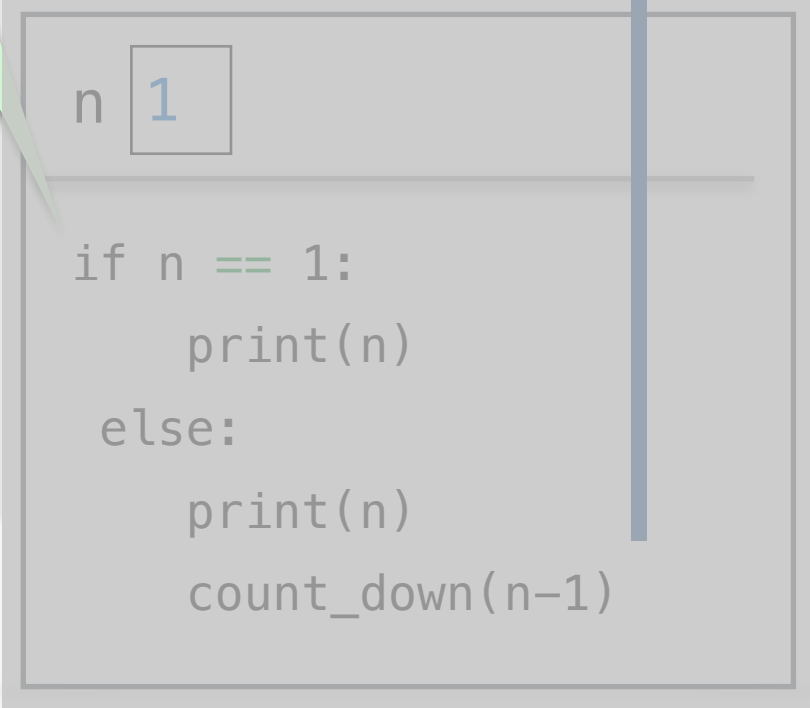


Base case reached!

```
>>> val = count_down(4)
```

```
4
3
2
1
```

**countDown(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

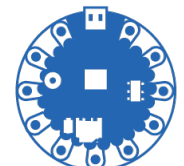
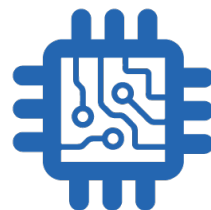
```
if n == 1:
    print(n)
else:
    print(n)
    count_down(n-1)
```

# 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 through recursion stumbles



# More Recursion: `count_up`



# count\_up(n)

- Write a recursive function that prints integers from **1** up to **n**
- Recursive definition of count\_up:
  - **Base case:**  $n = 1$ , `print(n)`
  - **Recursive rule:** `call count_up(n-1), print(n)`

We swapped the order of recursing  
(calling count\_up) and printing

```
>>> count_up(5)
```

```
1  
2  
3  
4  
5
```

```
>>> count_up(4)
```

```
1  
2  
3  
4
```

```
>>> count_up(3)
```

```
1  
2  
3
```

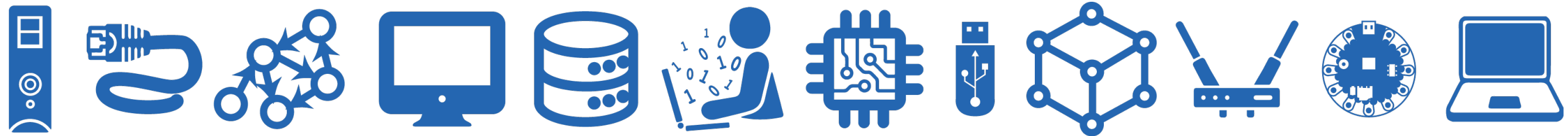
# count\_up(n)

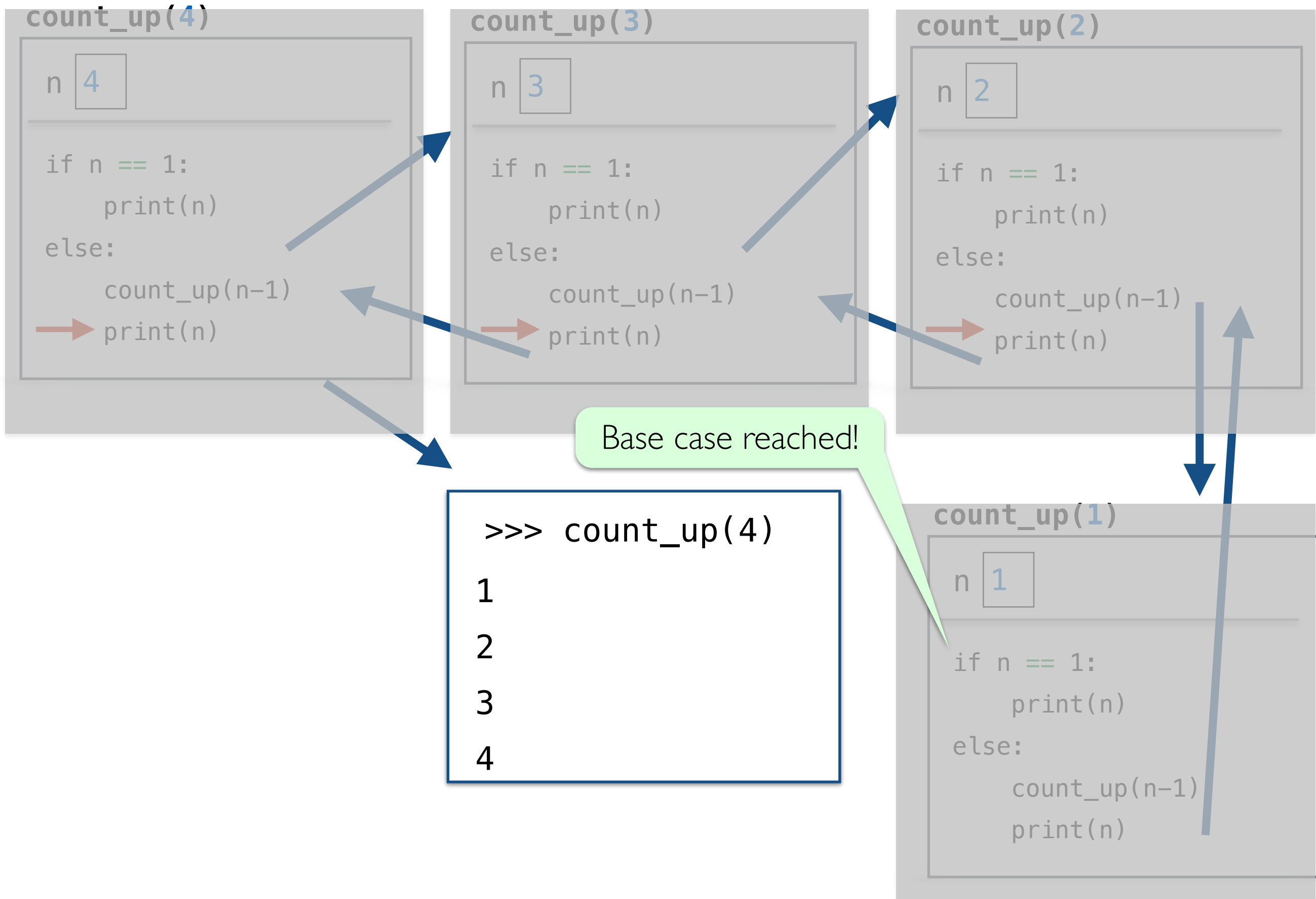
- Note that unlike `count_down(n)` we moved our print statement to be **after** the recursive function call
- By printing **after** the recursive call, the print statement gets executed “on the way back” from recursive calls

```
def count_up(n):  
    '''Prints out integers from 1 up to n'''  
    if n == 1:  
        print(n)  
    else:  
        count_up(n-1)  
        print(n)
```

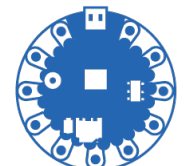
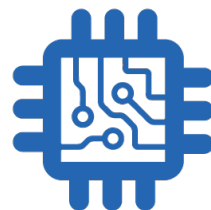
```
>>> count_up(5)  
1  
2  
3  
4  
5
```

# Function Frame Model to Understand `count_up`





# 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 count_down_gotcha(n):  
    '''Prints ints from 1 up to n'''  
    if n == 1: # Base case  
        print(n)  
    else:      # Recursive case  
        print(n)  
        count_down_gotcha(n)
```


Subproblem not getting smaller!



# GOTCHA #2

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

```
def print_halves_gotcha(n):  
    """Prints n, n/2, down to ... 1"""  
    if n > 0:  
        print(n)  
        return print_halves_gotcha(n/2)
```

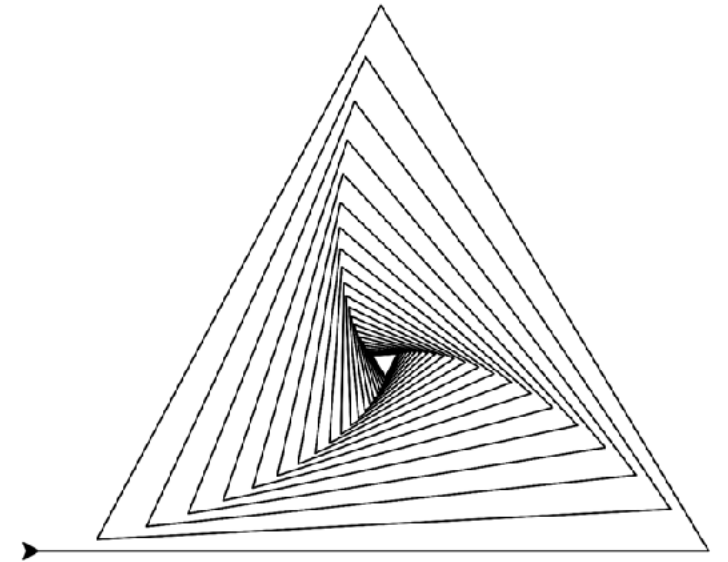
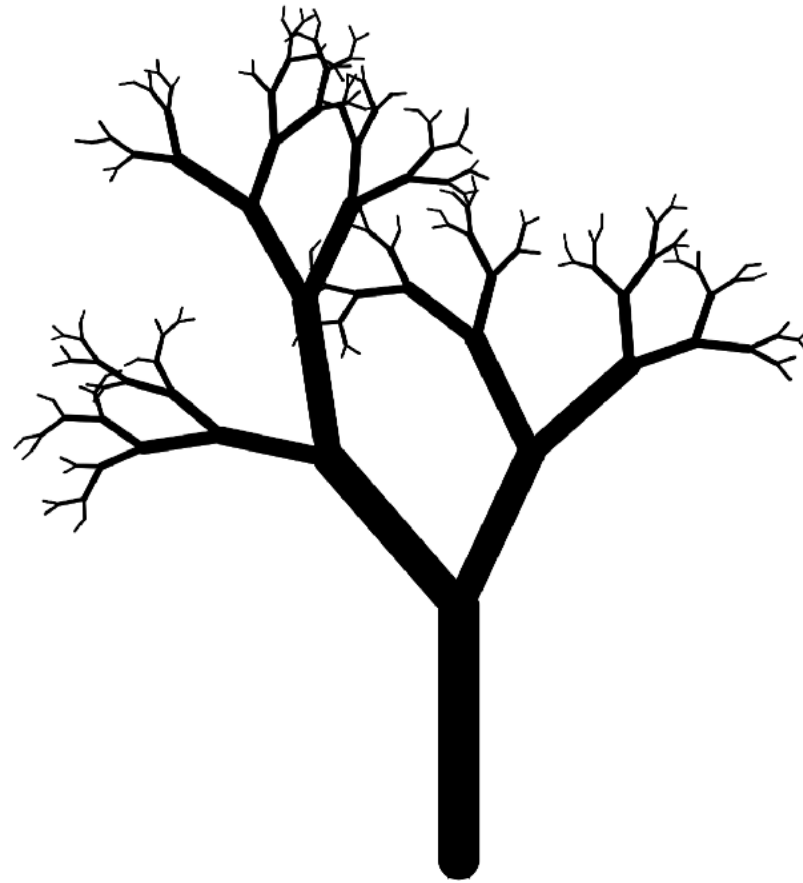
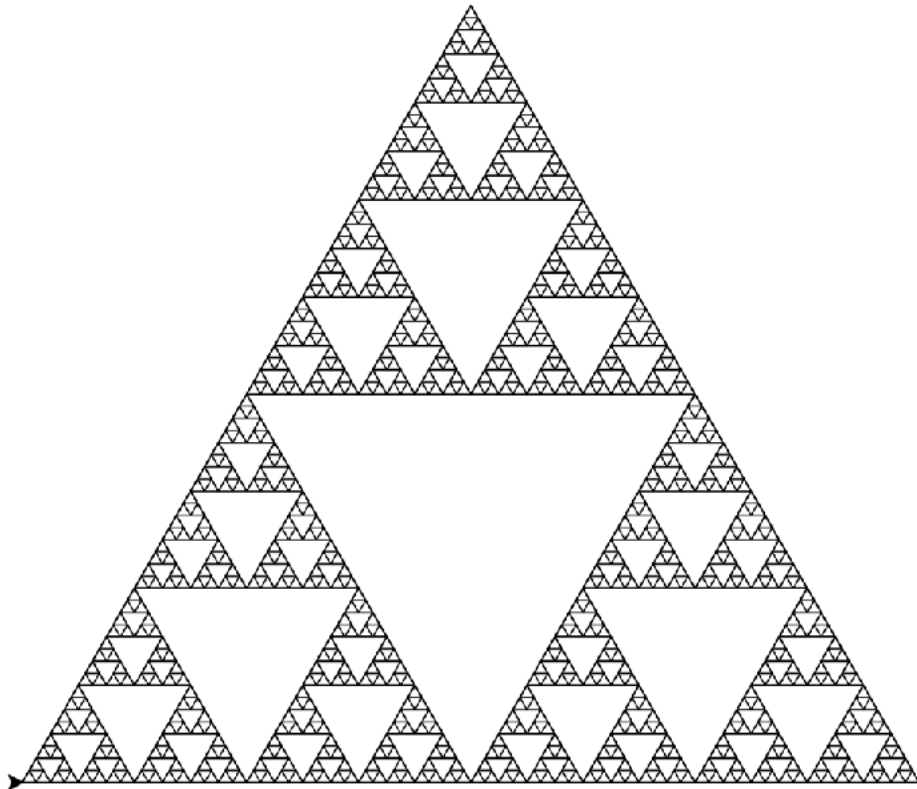


"Maximum recursion  
depth exceeded"

- 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

# Next Lectures

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



# The end!

