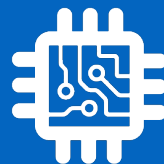


CSI 34: Recursion



Announcements & Logistics

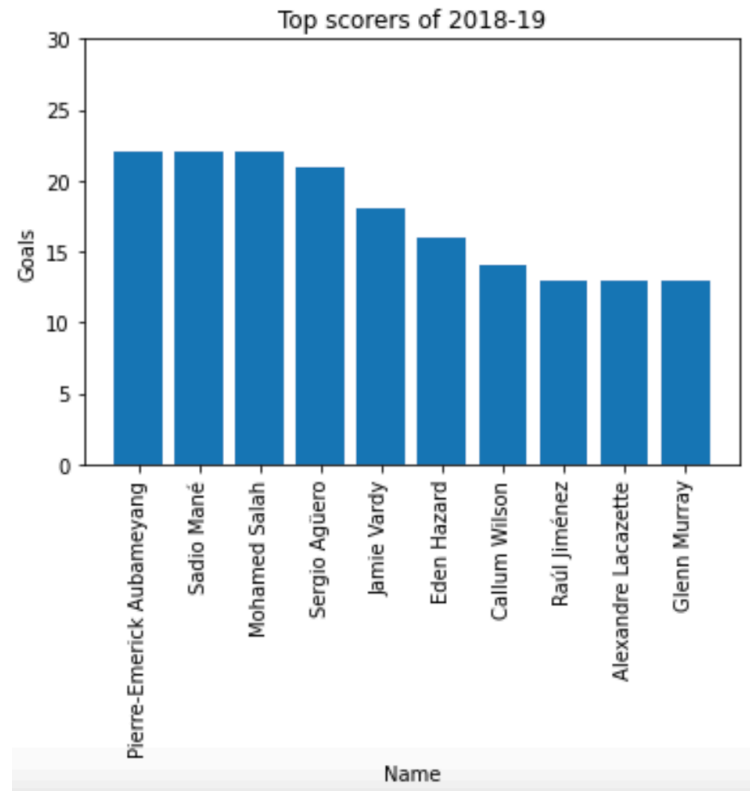
- **Lab 6 due Wed/Thurs at 10 pm**
 - Uses dictionaries, plotting, CSV files
- **HW 6** will be available at noon today, due next Mon at 10pm
- **Lab 5** will be returned today
- Midterms coming soon

- Mac users: Please do not update your Mac right now!
- Thanksgiving week: Optional lab attendance!

Do You Have Any Questions?

Last Time

- Worked through a simple example involving CSVs, dictionaries, and sets
- Discussed plotting with matplotlib
- Python is a powerful language for data processing and visualization



Where are We Going?

- First half of CSI 34: learned the **fundamentals of programming**
 - Functions, conditionals, loops, data types
 - Built-in data structures and methods, sorting, plotting
- 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
 - Building our own data types: **linked lists**
 - **How does sorting really work**/ what happens under the hood when Python is sorting?
 - Continue developing our intuition regarding 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

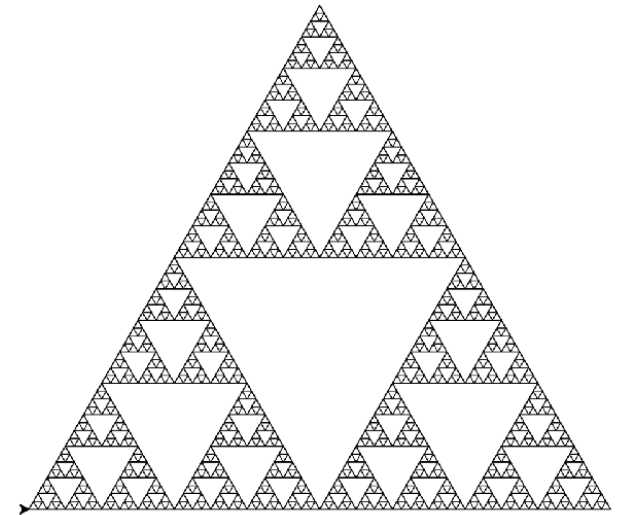
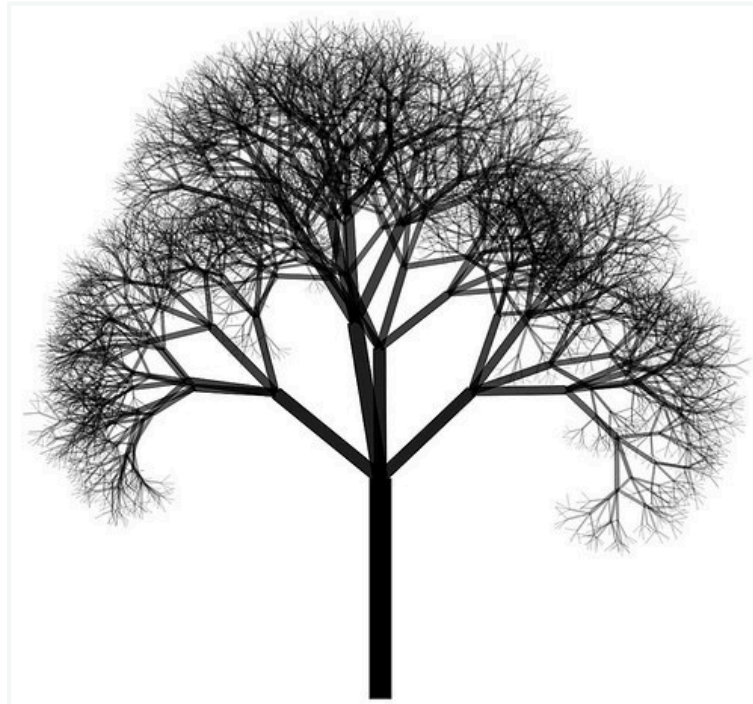
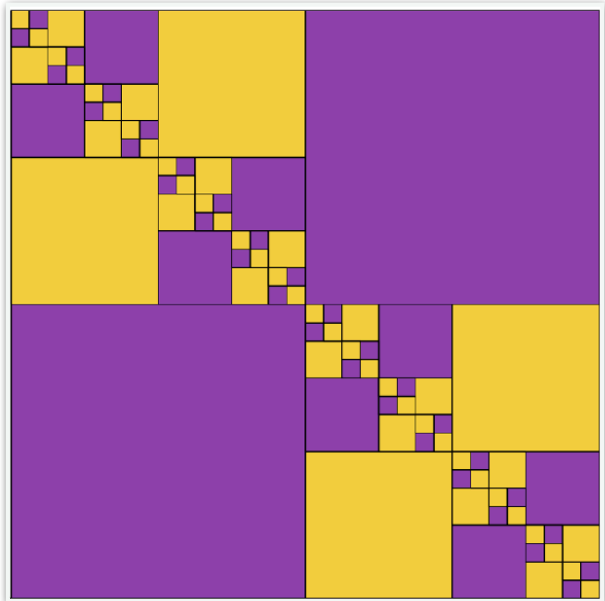
- Computer scientists were of course writing nerdy poems about recursion long before it was cool (and before we had computers).

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 cool things.
- By the end of lab next 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):** 0 is a natural number
 2. **Recursive rule(s):** 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:**
 - **Recursive rule:**

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 is (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**
- Recursive functions consist of one or more **base cases** and a set of **recursive rules** that successively **simplify** (or reduce) the problem **until we reach one of the base cases**
- Recursive rules **must** eventually take you to one of the base cases, else we end up with the recursive equivalent of an infinite loop
- We will compare recursive implementations to iterative implementations soon, but for now let's 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 causes a **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 but let's take a closer look at what goes on in recursive function calls.

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 CSI 36/256, 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:** $n = 0$, return 0

Stop and don't print

- **Recursive rule:** `print(n), return countdown(n-1)`

Perform one step

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

Understanding Recursive Functions

- Recursive definition of countdown:
 - **Base case:** $n = 0$, return 0
 - **Recursive rule:** `print(n)`, return `countDown(n-1)`

```
def countDown(n):  
    '''Prints numbers from n down to 1'''  
    if n < 1: # Base case  
        return 0  
    else: # Recursive case: n >= 1:  
        print(n)  
        return countDown(n-1)
```

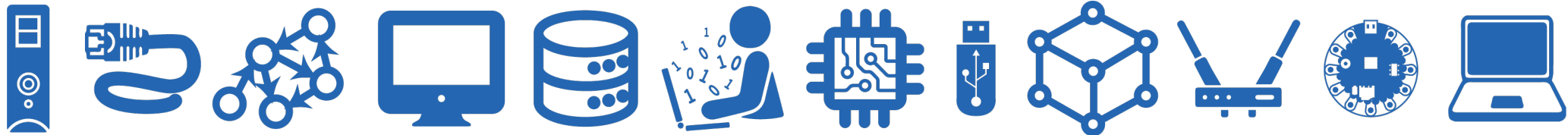
```
result = countDown(5)
```

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**

Review: Function Frame Model



Review: Function Frame Model

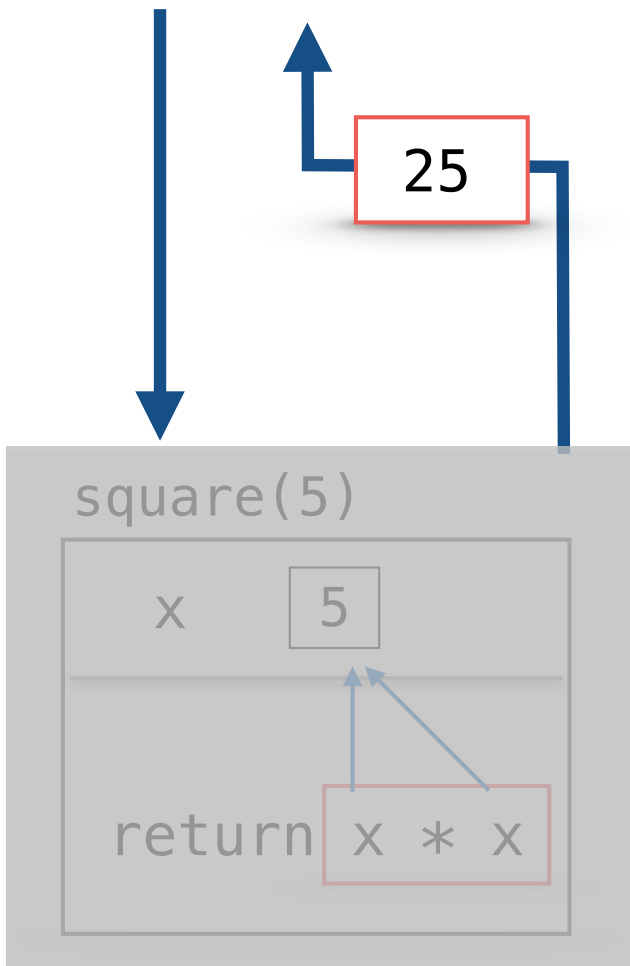
- Consider a simple function `square`
- What happens when `square(5)` is invoked?

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


Review:

Function Frame Model

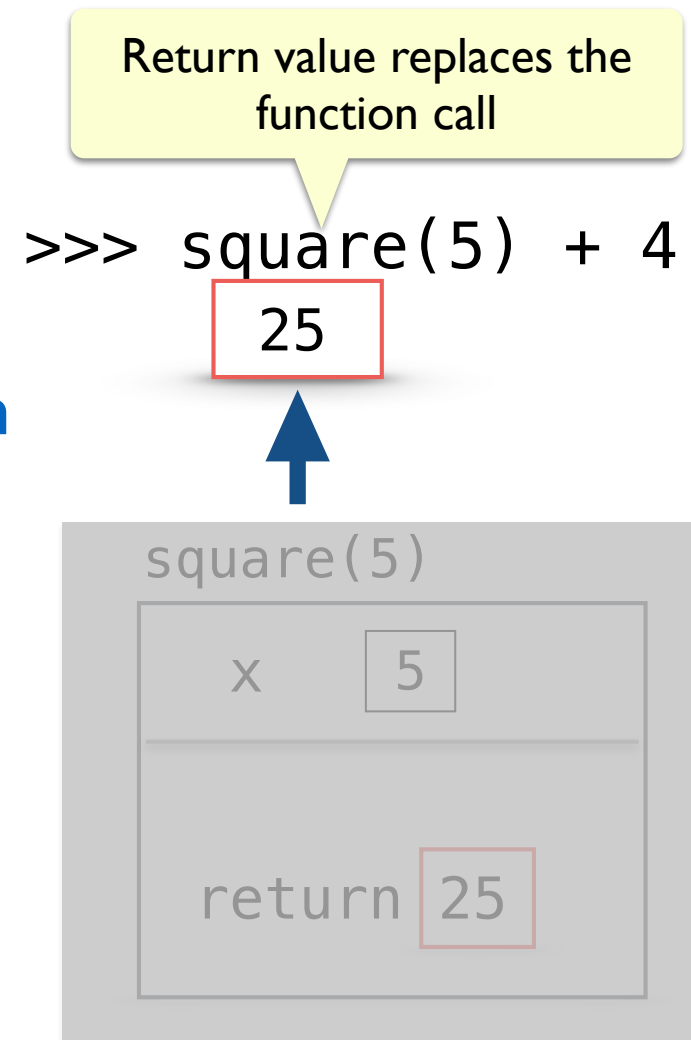
```
>>> square(5)
```



Summary:

Function Frame Model

- 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)
```

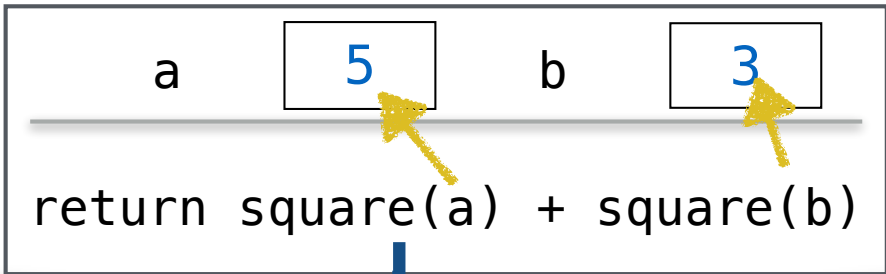
- What happens when we call `sumSquare(5, 3)`?

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

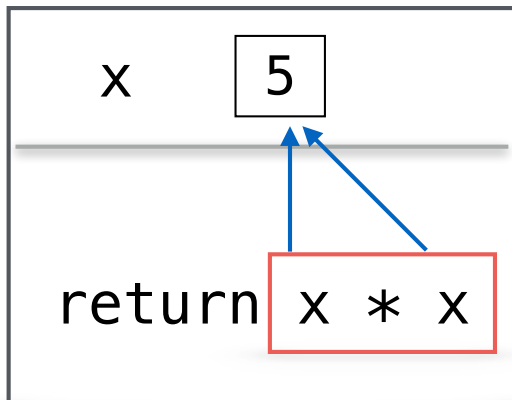
```
>>> sumSquare(5,3)
```



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



```
square(5)
```

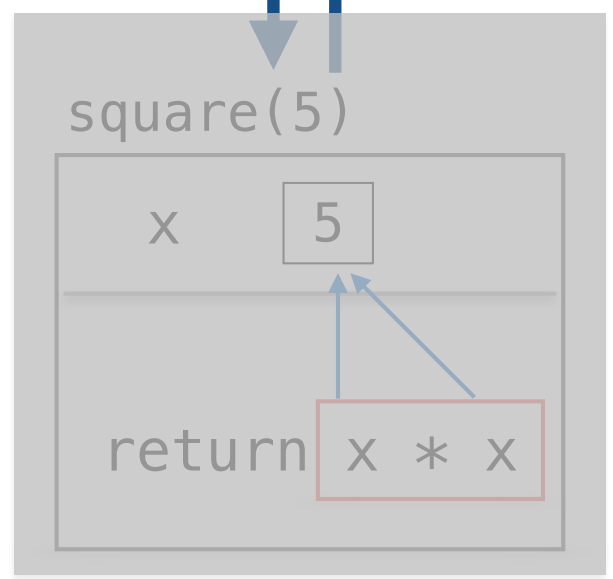
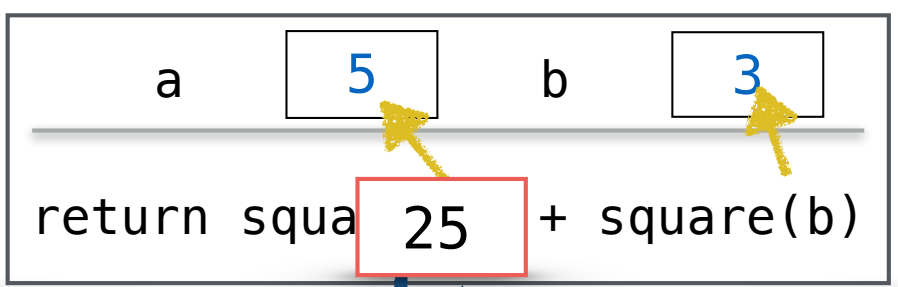


```
def sumSquare(a, b):  
    return square(a) + square(b)
```

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



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

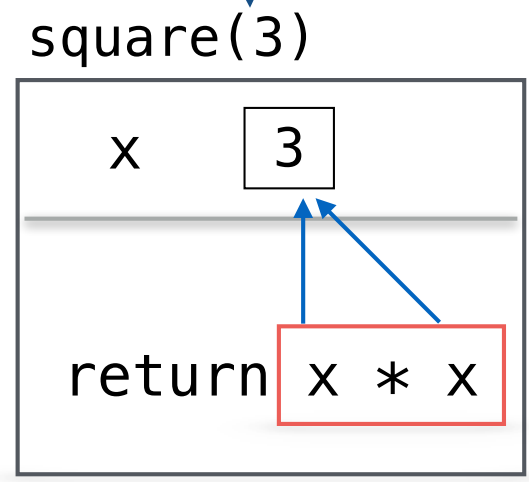
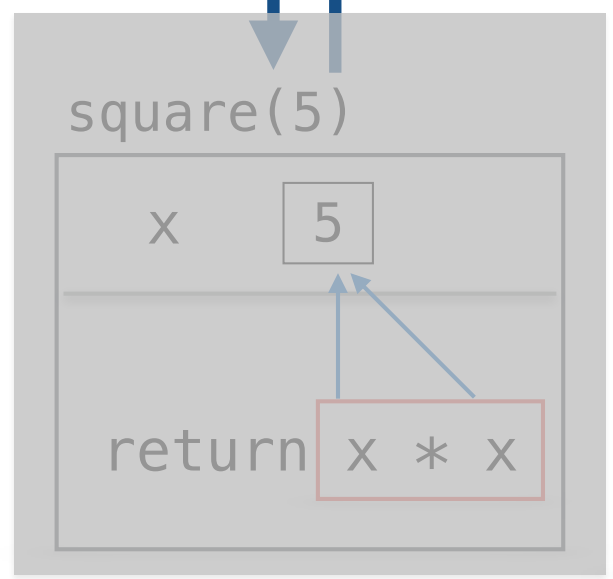
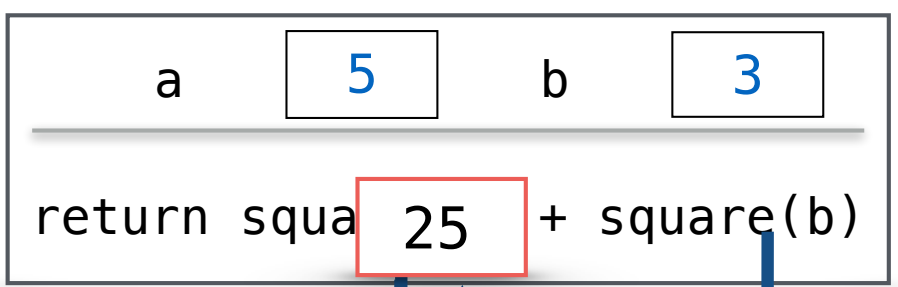


```
def sumSquare(a, b):  
    return square(a) + square(b)
```

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



sumSquare(5, 3)

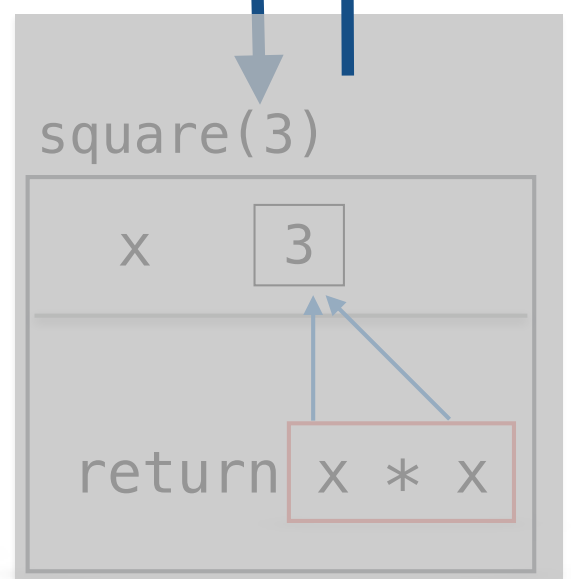
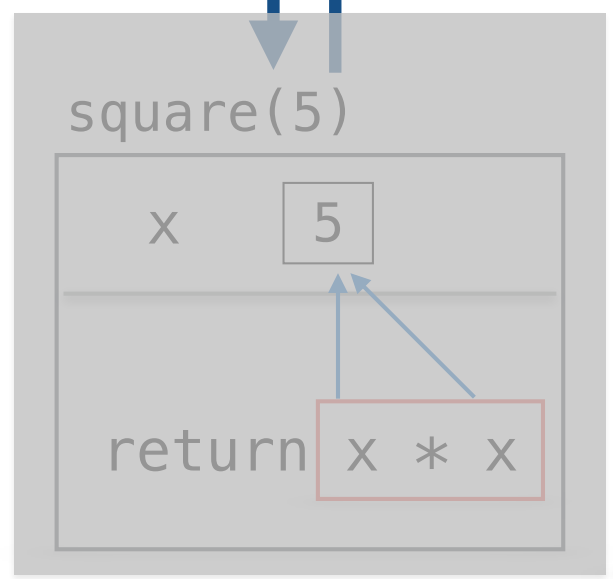
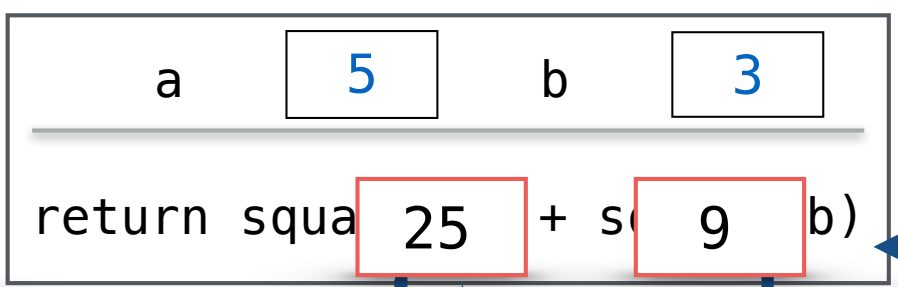


```
def sumSquare(a, b):  
    return square(a) + square(b)
```

>>> sumSquare(5,3)



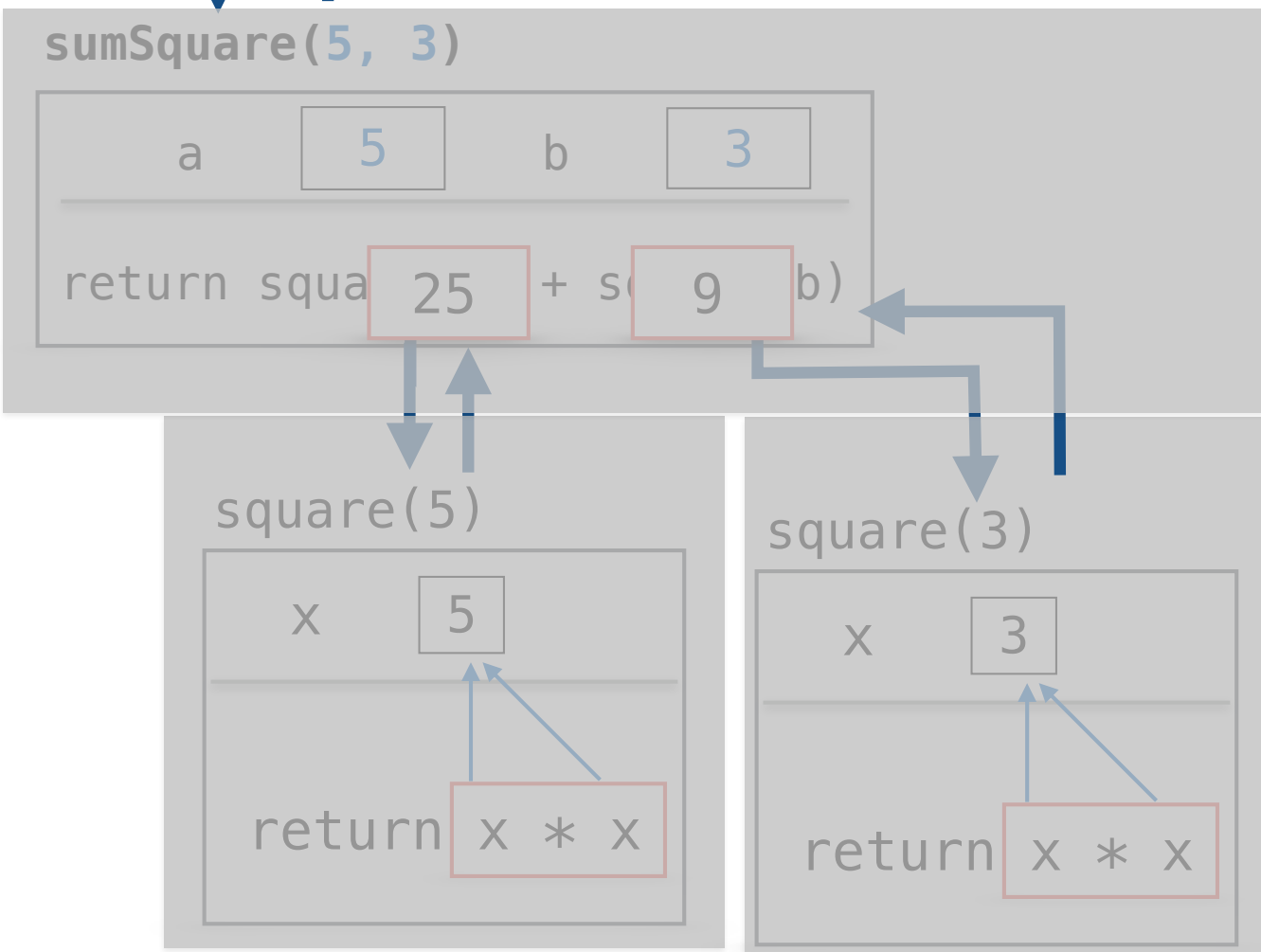
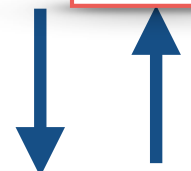
sumSquare(5, 3)



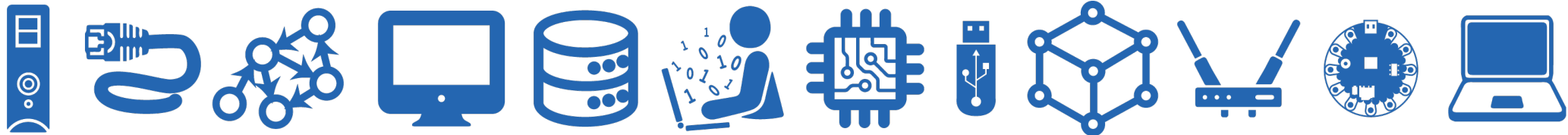
```
def sumSquare(a, b):  
    return square(a) + square(b)
```

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

34



Function Frame Model to Understand `countDown`



```
def countDown(n):  
    '''Prints ints from n down to 1'''  
    if n < 1:  
        return 0  
    else:  
        print(n)  
        return countDown(n-1)
```

```
>>> val = countDown(5)
```

```
5  
4  
3  
2  
1
```

```
>>> val = countDown(4)
```

```
4  
3  
2  
1
```

countDown(3)

```
n 3


---


if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

countDown(2)

```
n 2


---


if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

countDown(1)

```
n 1


---


if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

Base case reached!

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

countDown(0)

```
n 0


---


if n < 1:
    return 0
else:
    print(n)
    return countDown(n-1)
```

countDown(3)

```
n 3
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

countDown(2)

```
n 2
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

countDown(1)

```
n 1
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

Base case reached!

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

countDown(0)

```
n 0
-----
if n < 1:
    return 0
else:
    print(n)
    return countDown(n-1)
```

countDown(3)

```
n 3
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

countDown(2)

```
n 2
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

countDown(1)

```
n 1
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

Base case reached!

```
>>> val = countDown(3)
```

```
3
2
1
```

countDown(0)

```
n 0
-----
if n < 1:
    return 0
else:
    print(n)
    return countDown(n-1)
```

countDown(3)

```
n 3
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

countDown(2)

```
n 2
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

countDown(1)

```
n 1
-----
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

Base case reached!

```
>>> val = countDown(3)
```

```
3
2
1
```

countDown(0)

```
n 0
-----
if n < 1:
    return 0
else:
    print(n)
    return countDown(n-1)
```

```
countDown(3)
n 3
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

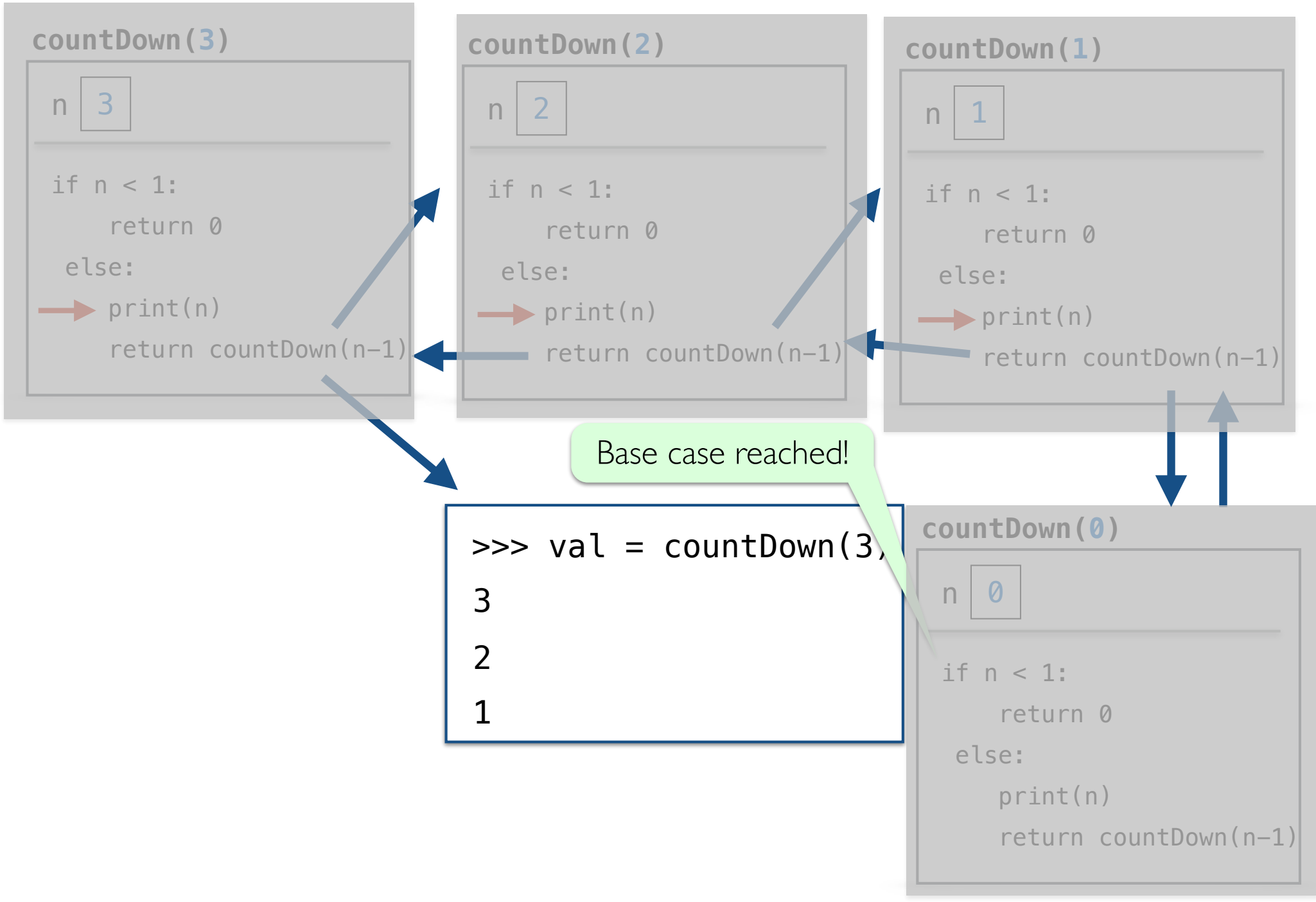
```
countDown(2)
n 2
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

```
countDown(1)
n 1
if n < 1:
    return 0
else:
    → print(n)
    return countDown(n-1)
```

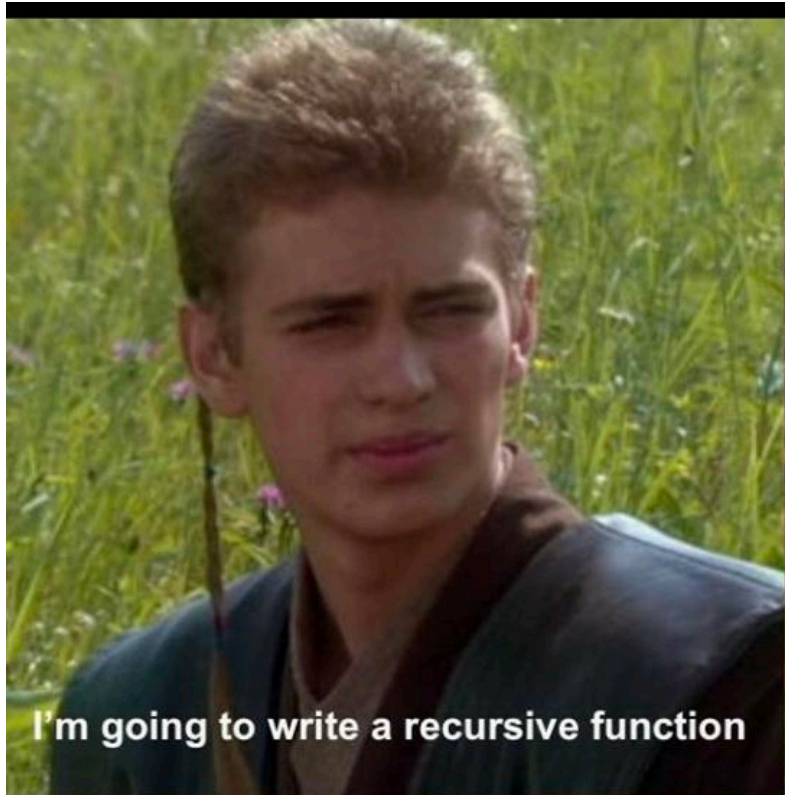
Base case reached!

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

```
countDown(0)
n 0
if n < 1:
    return 0
else:
    print(n)
    return countDown(n-1)
```



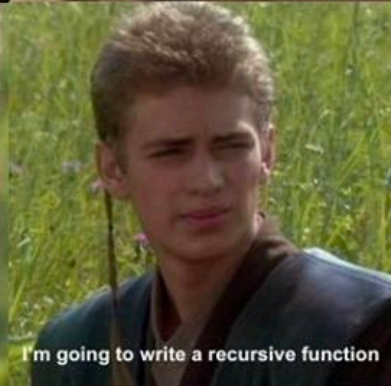




I'm going to write a recursive function



With a base case, right?



I'm going to write a recursive function



With a base case, right?

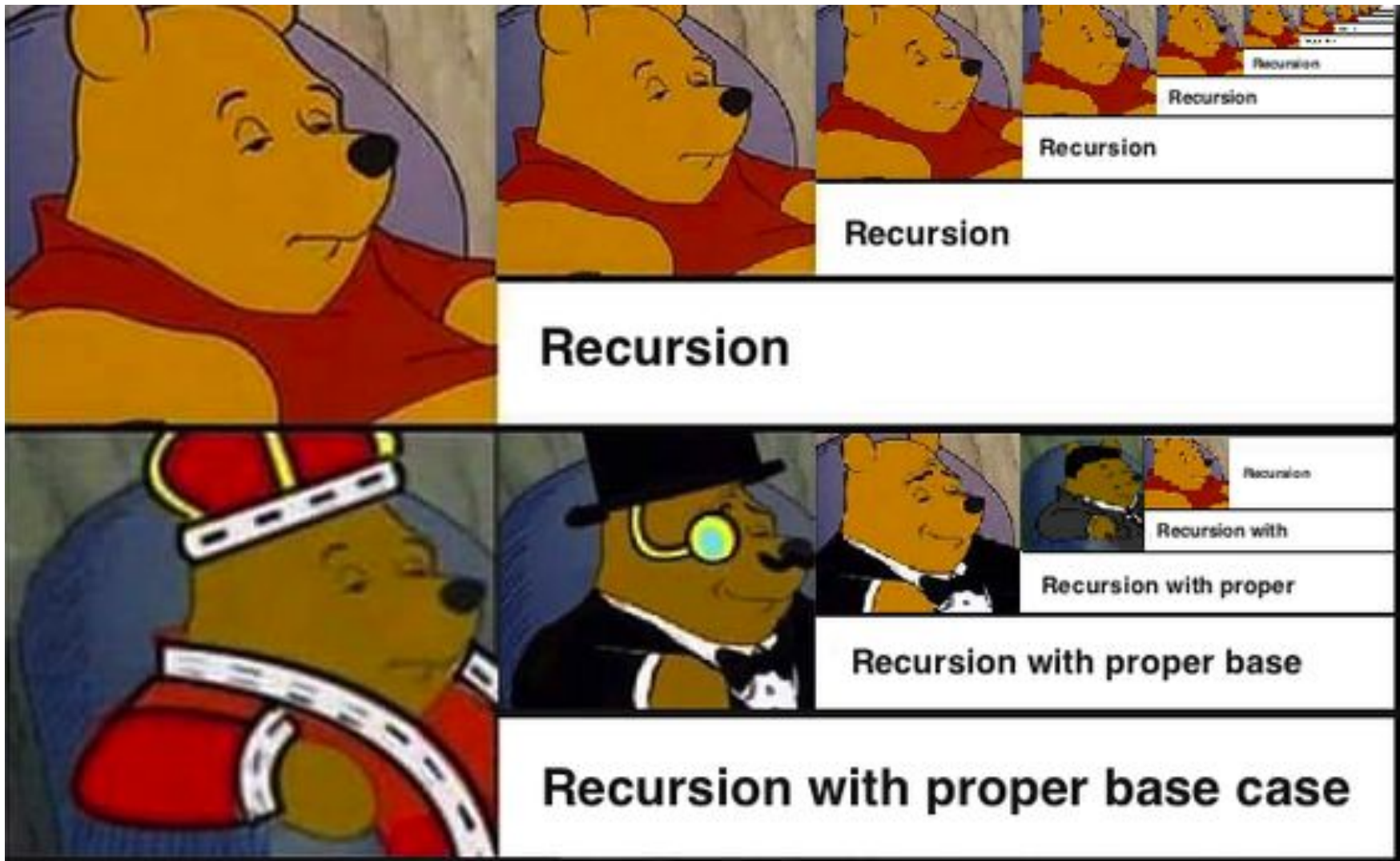


I'm going to write a recursive function



With a base case, right?





Recursion

Recursion

Recursion

Recursion

Recursion

Recursion with proper base case

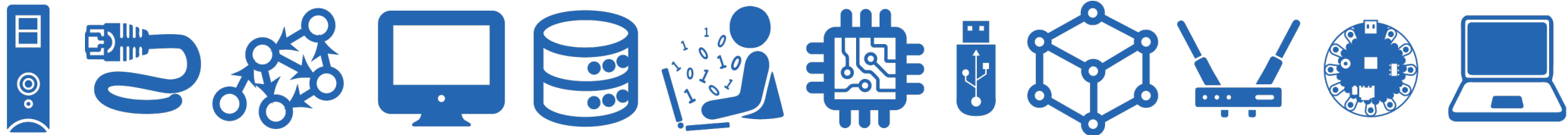
Recursion with proper base

Recursion with proper

Recursion with

Recursion

More Recursion: countUp



countUp(n)

- Write a recursive function that prints integers from **1** up to **n**
- Recursive definition of countUp:
 - **Base case:** $n = 0$, return 0
 - **Recursive rule:** call `countUp(n-1)`, `print(n)`, return

We swapped the order of recursing (calling countUp) and printing

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

```
1  
2  
3  
4  
5
```

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

```
1  
2  
3  
4
```

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

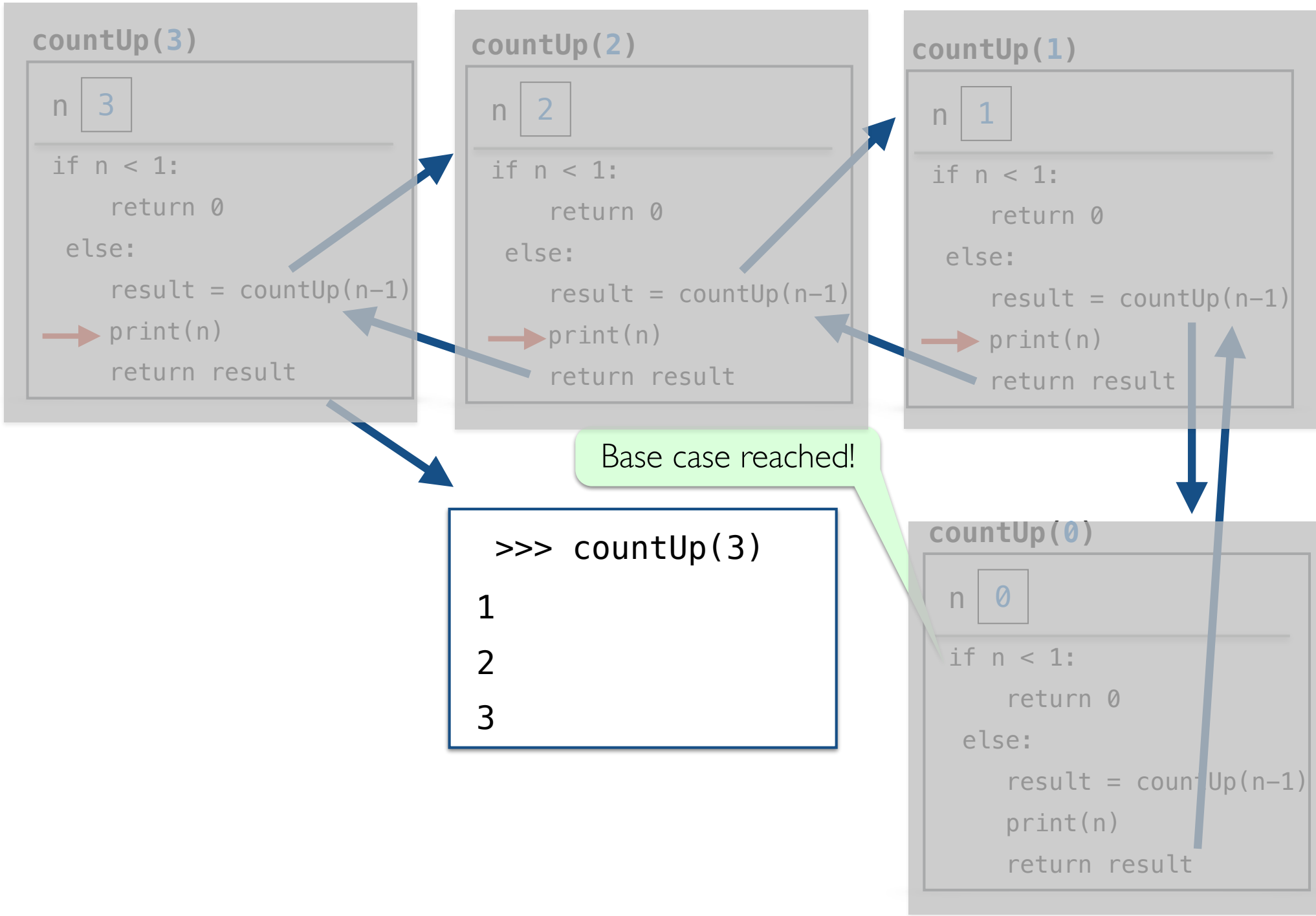
```
1  
2  
3
```

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:  
        return 0  
    else:  
        result = countUp(n-1)  
        print(n)  
        return result
```

Function Frame Model to Understand `countUp`



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 countUpGotcha(n):  
    '''Prints ints from 1 up to n'''  
    if n <= 0: # Base case  
        return 0  
    else:      # Recursive case  
        print(n)  
        return countDownGotcha(n)
```

Subproblem not getting smaller!

GOTCHA #2

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

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

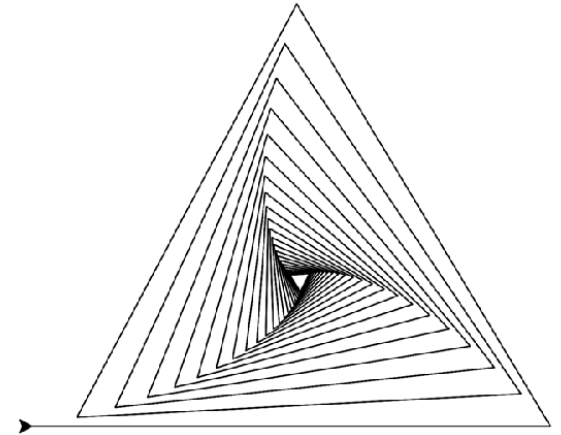
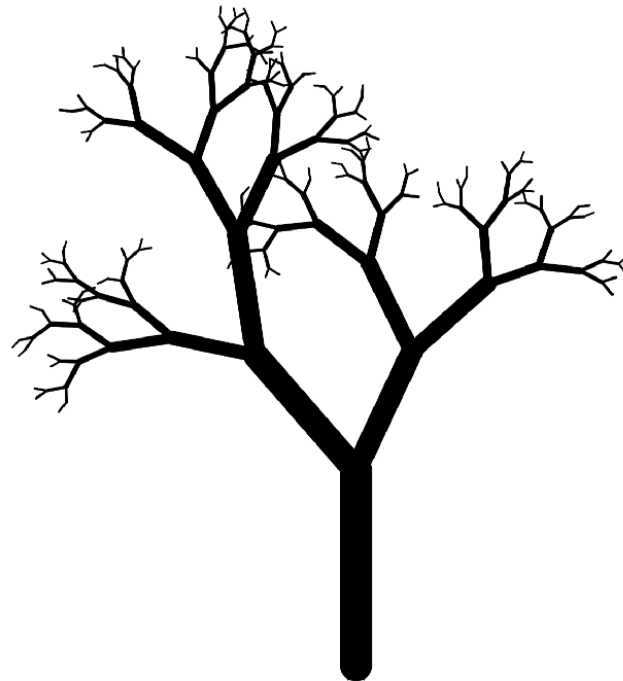
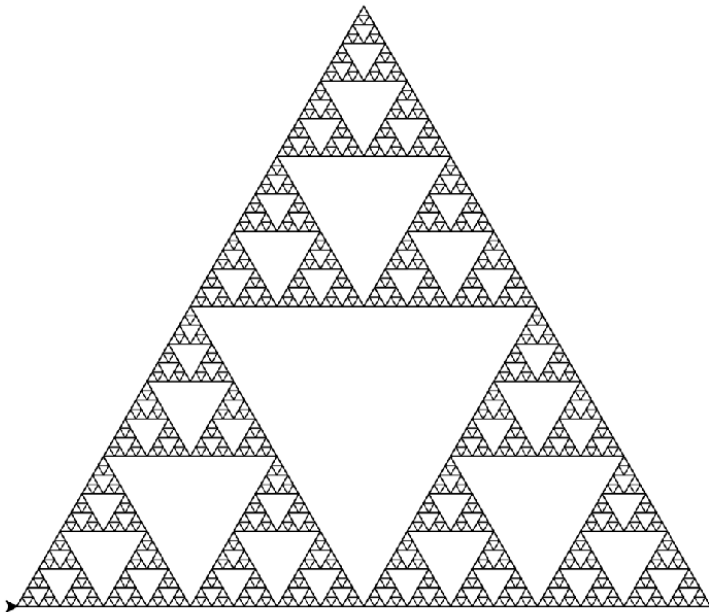


"Maximum recursion depth exceeded"

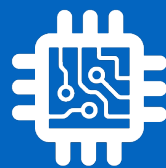
- In practice, the infinite recursion examples will terminate when Python runs out of resources for creating function call frames, leads to a "maximum recursion depth exceeded" error message

Next Lectures

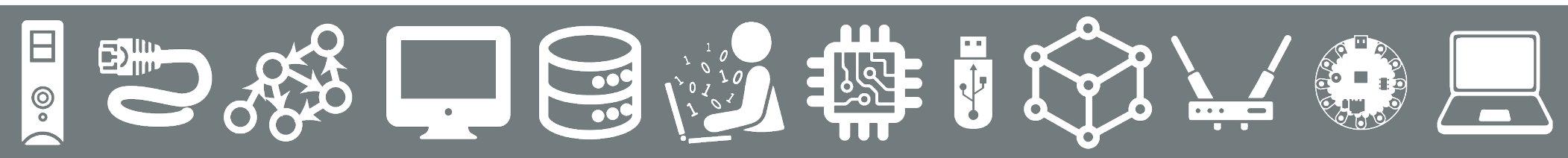
- Intro to **turtle** module and graphical recursion
- Comparing iterative and recursive programs



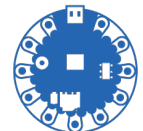
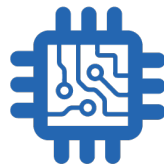
The end!



Leftover Slides



Recursive Story



Steps for Recursion

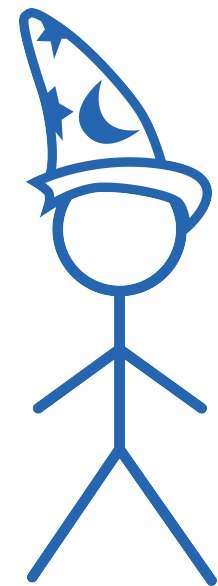
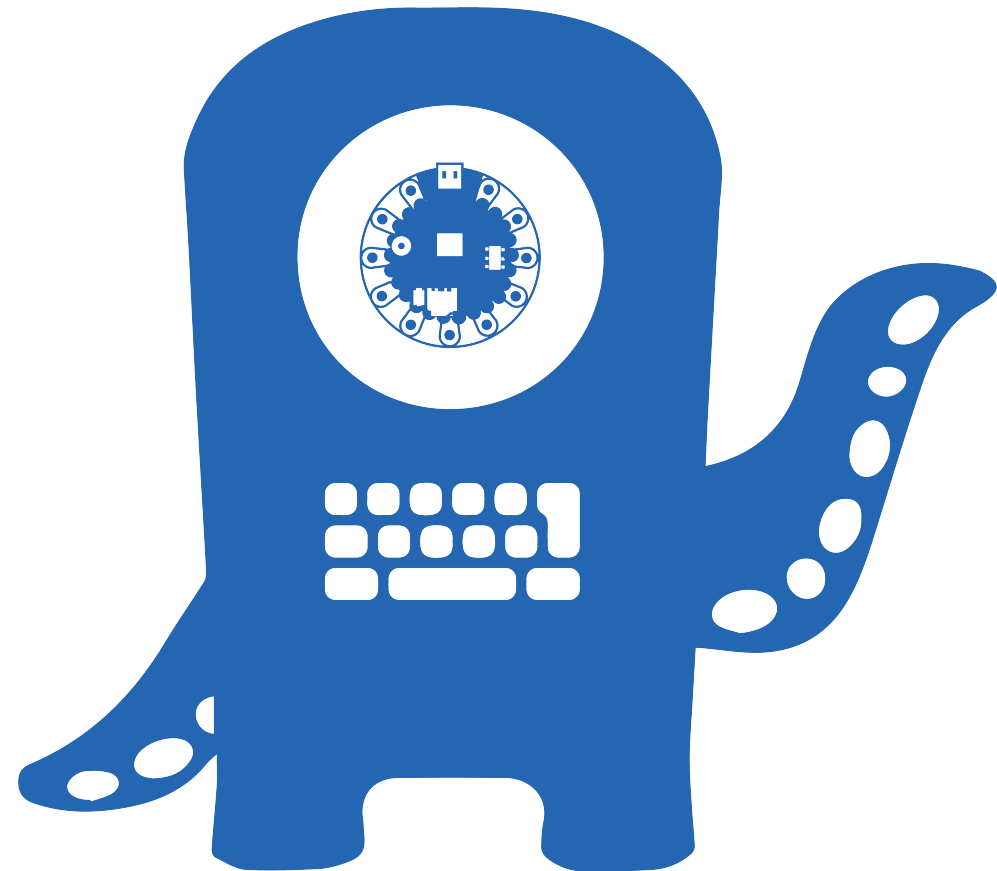
- Know when to stop
- Decide how to take one step
- Break the journey down into that step plus a smaller journey

Once upon a time, there was a very surly monster being kept by a sorcerer.

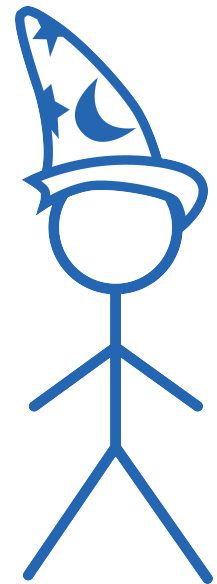
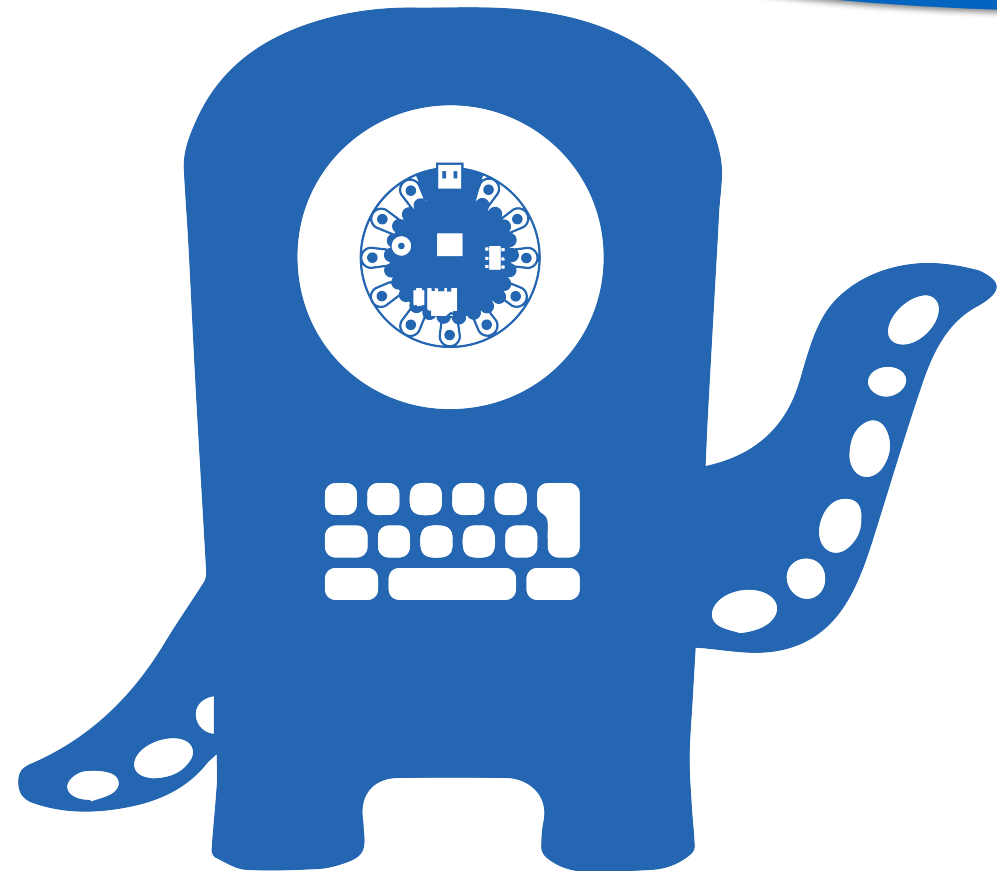
And there was a sorcerer's apprentice, Sam.

One day, the sorcerer sent Sam down to the basement to ask the surly monster with some help on a task.

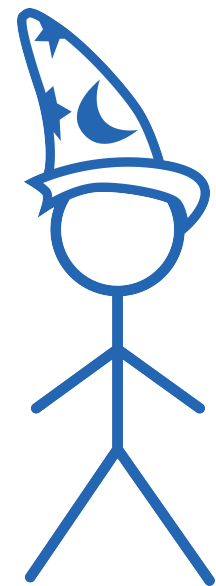
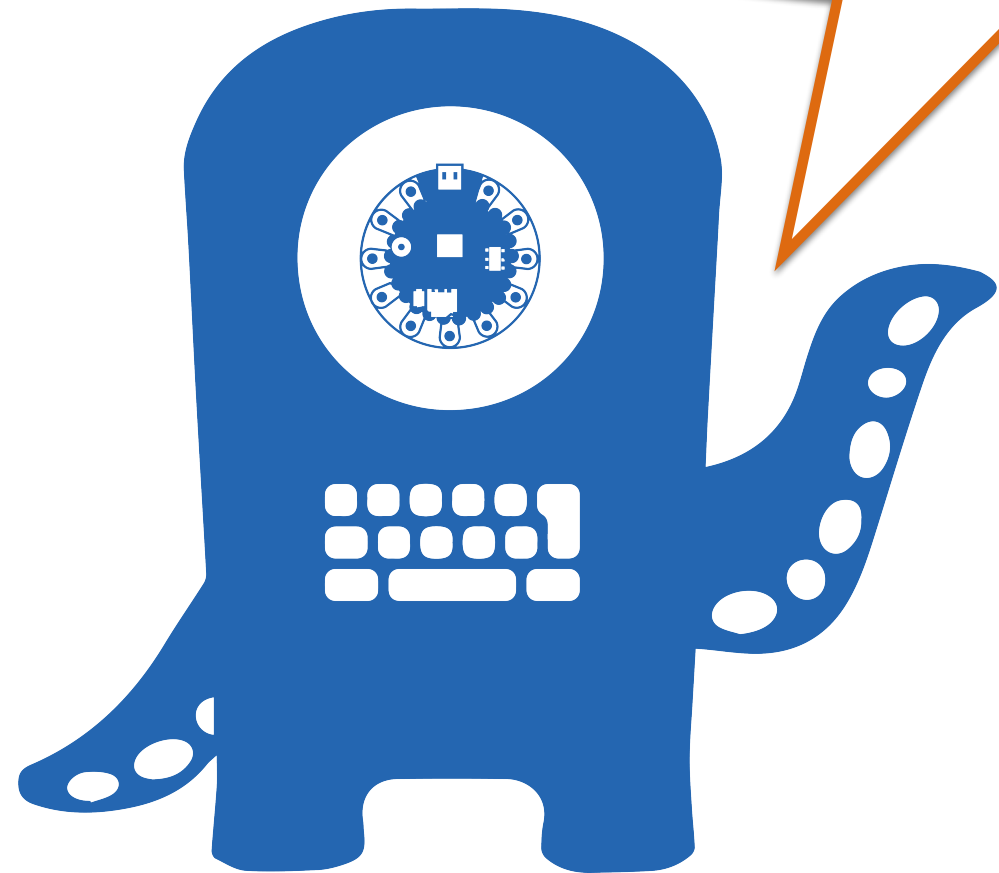
So Sam climbed down the stairs, and asked his question.



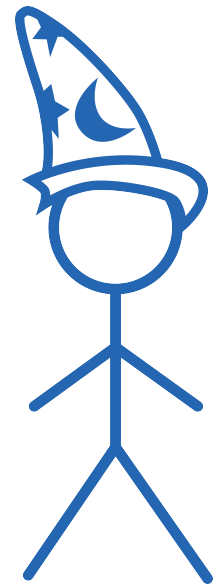
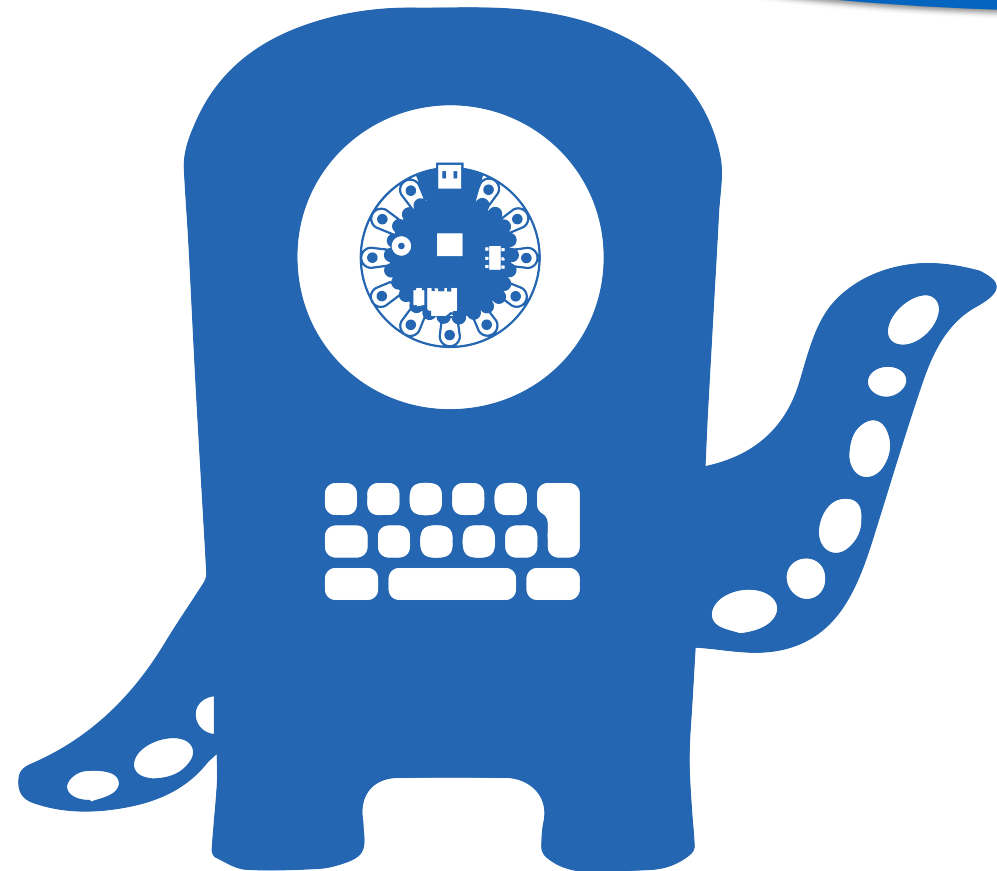
Monster, I need to know if
any of the numbers in this list are odd:
[3142, 5798, 6550, 8914]



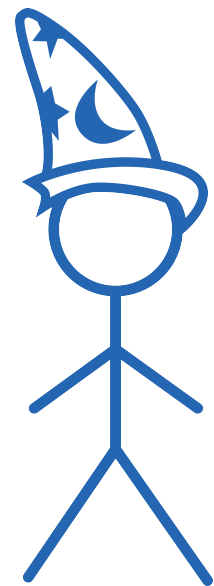
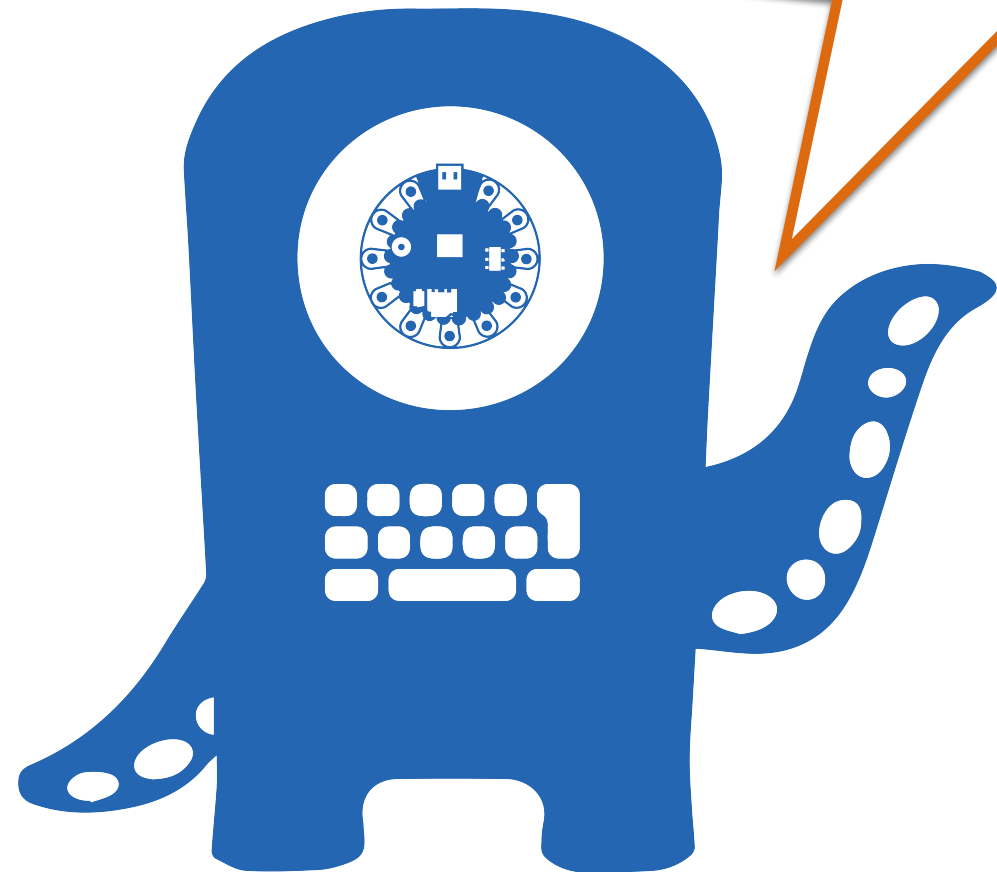
Sorry, I can only tell you if the first number of the list is odd.



But I need to know if any number in the list is odd, not just the first!



I'll only look at the first number, but
I'll look at as many lists as you like.

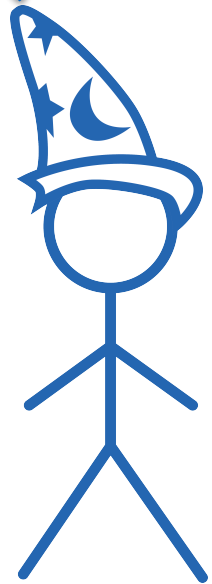
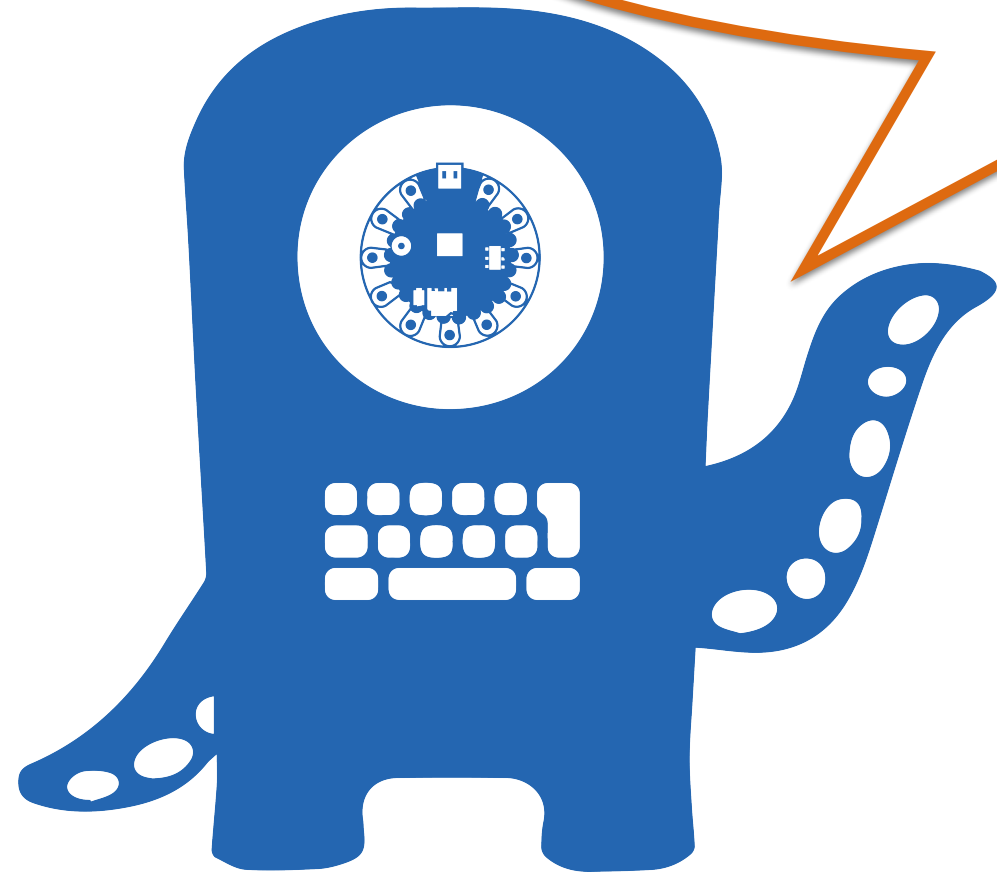




What should Sam do?

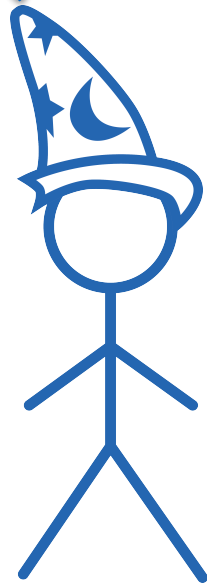
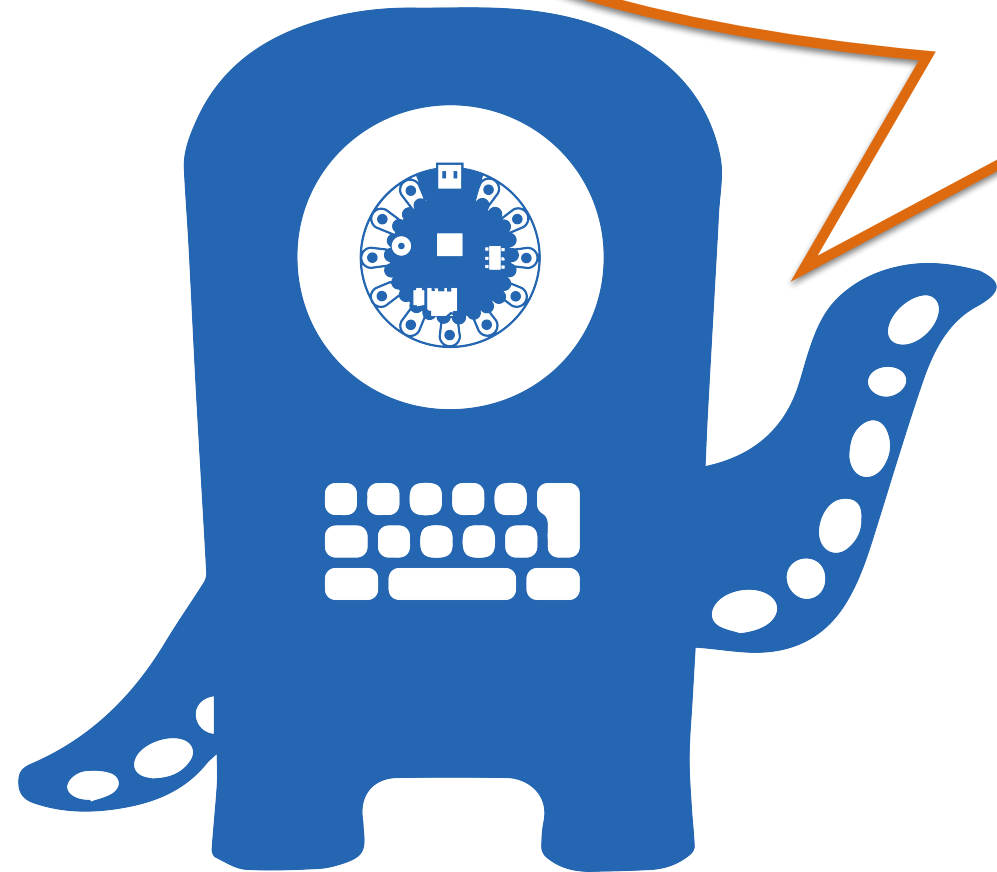
[3142, 5798, 6550, 8914]

The first number is not odd.



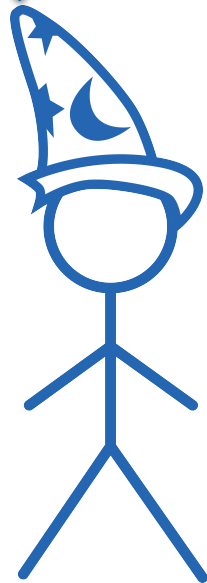
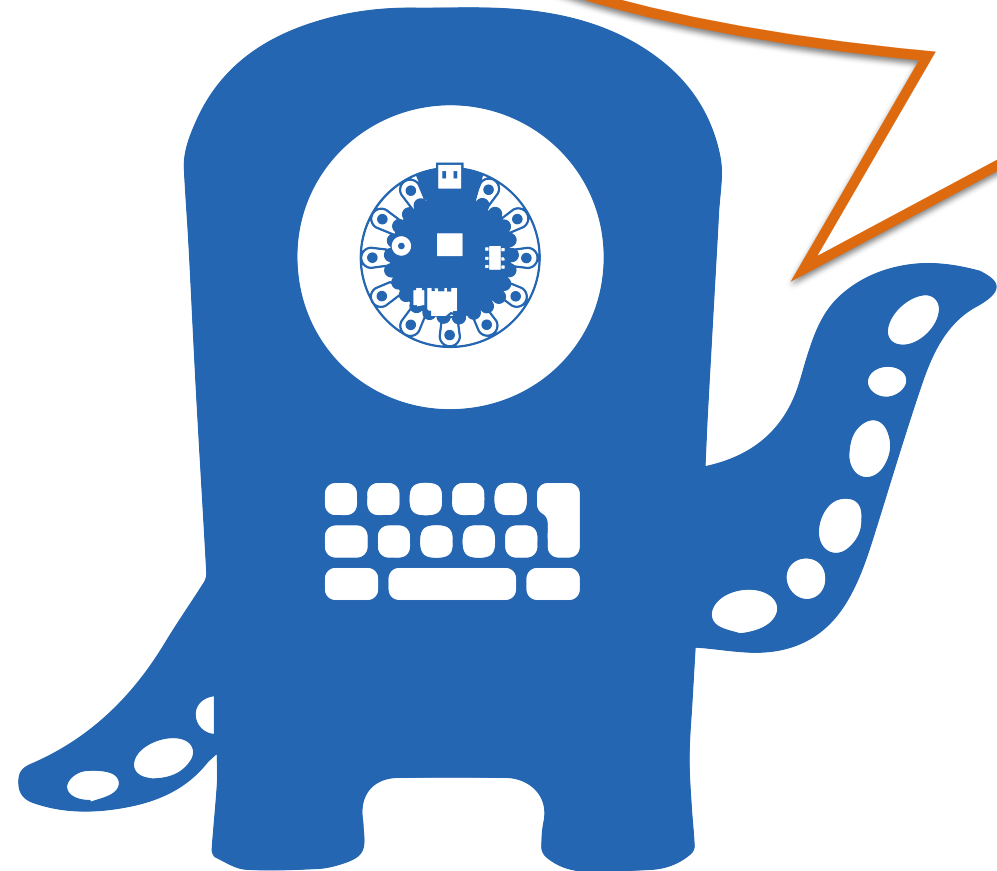
[~~3142~~, 5798, 6550, 8914]

The first number is not odd.



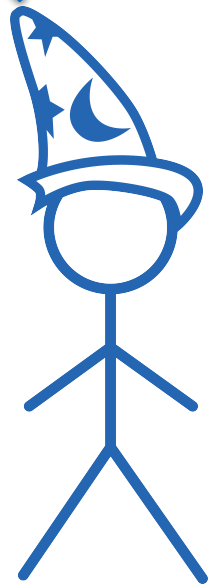
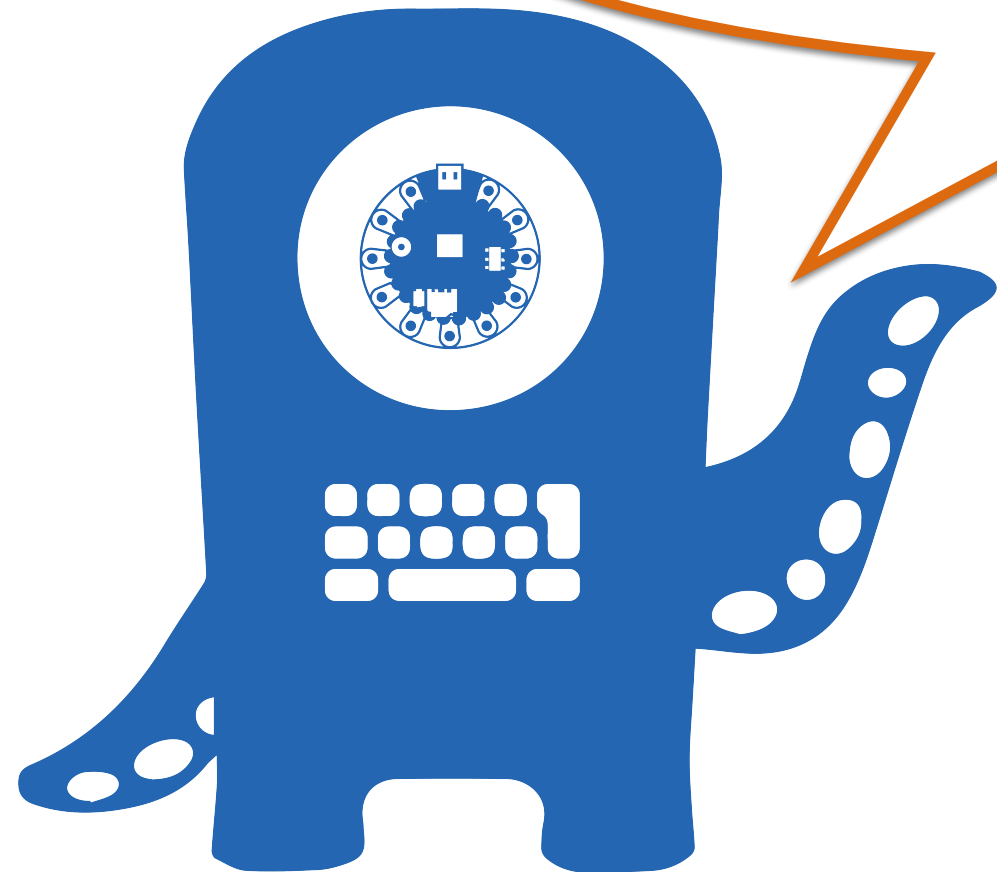
~~[3142, 5798, 6550, 8914]~~

The first number is not odd.



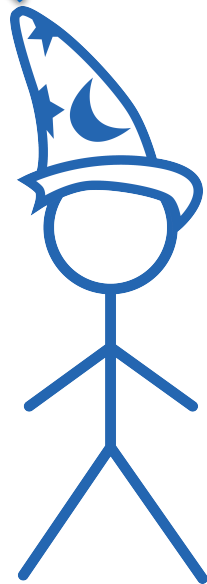
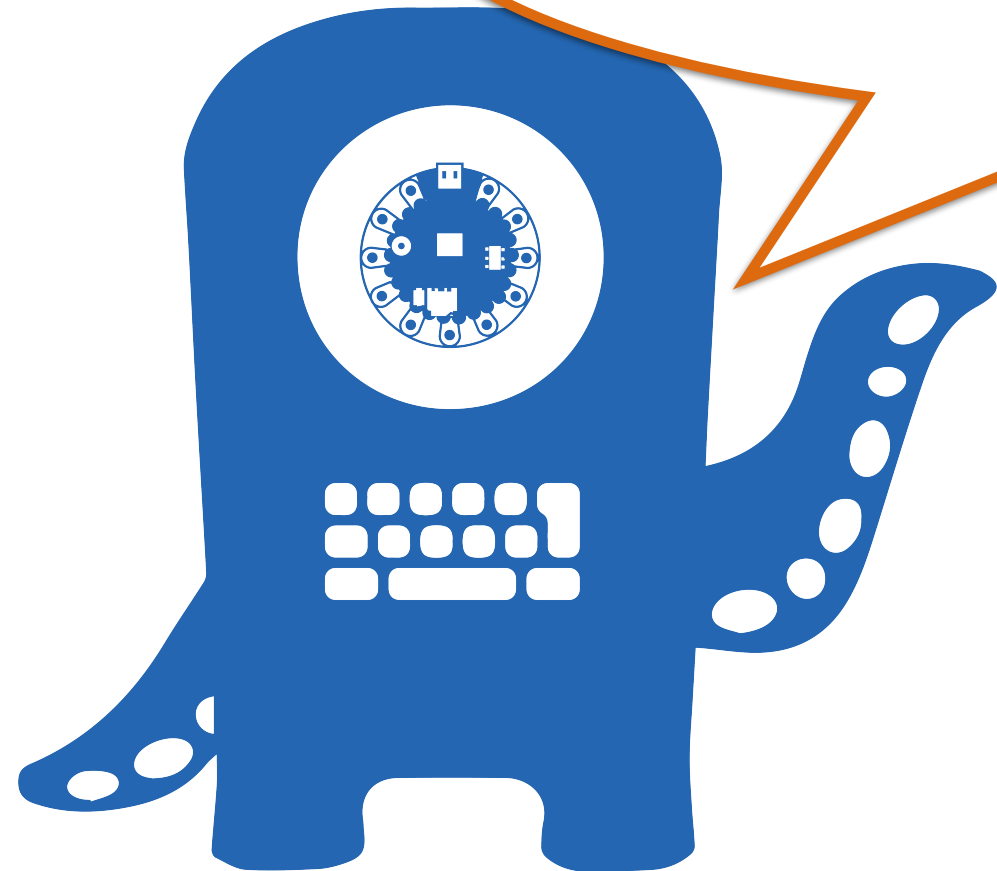
~~[3142, 5798, 6550, 8914]~~

The first number is not odd.

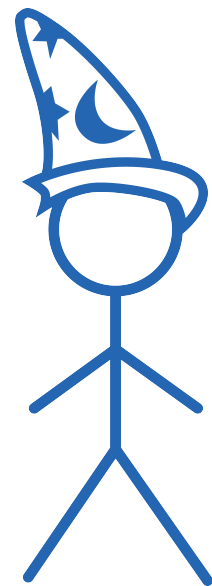
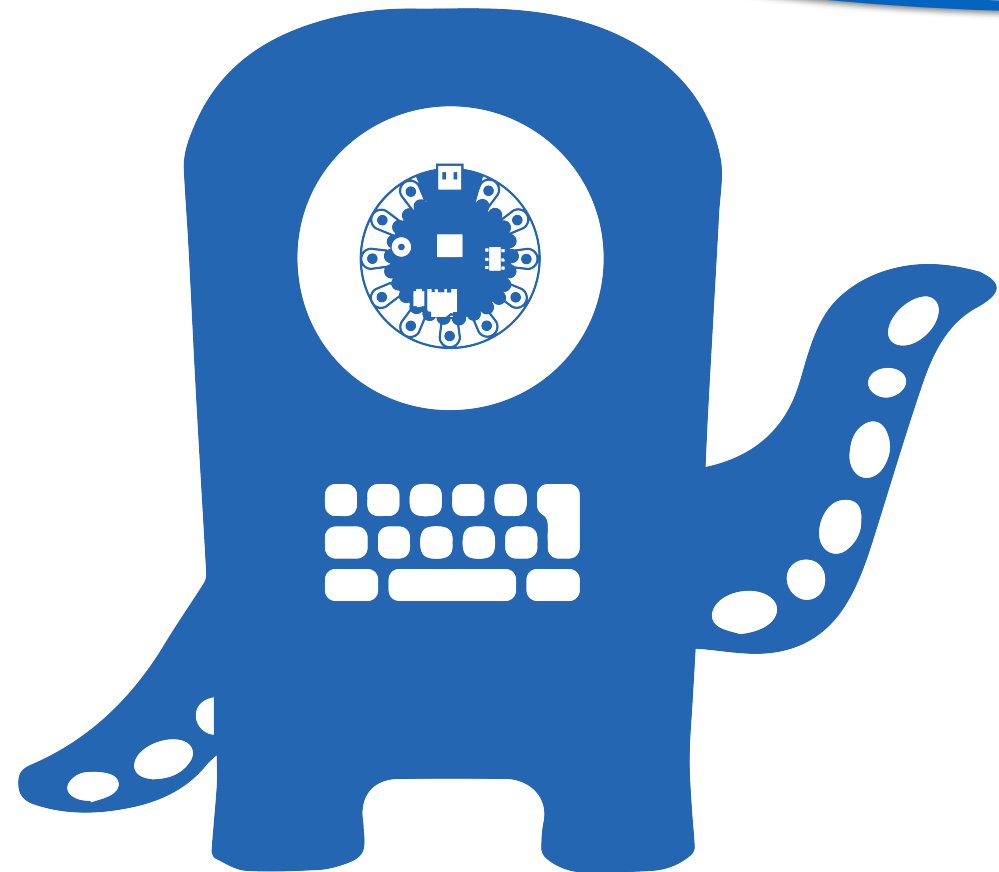


~~[3142, 5798, 6550, 8914]~~

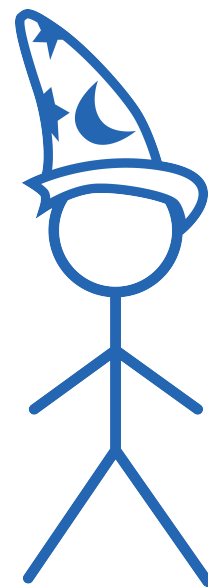
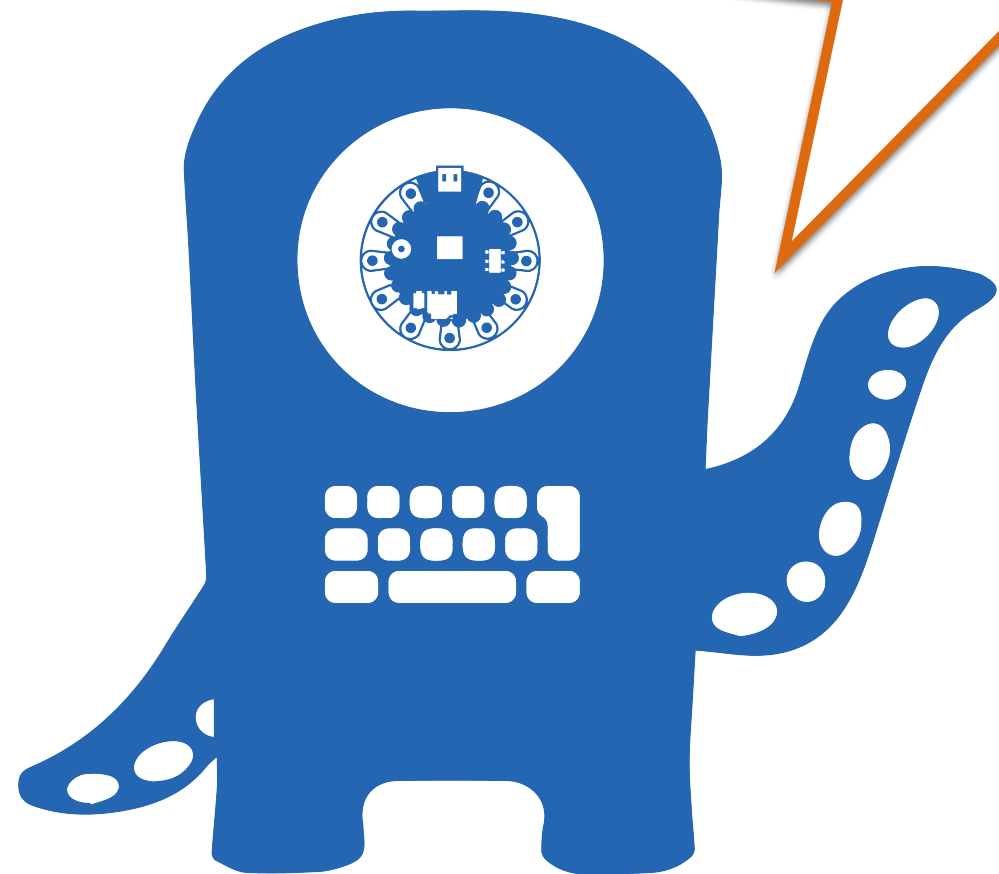
That's an empty list!
It can't be odd!



None of the numbers the Sorcerer
gave me were odd, thank you!



How can you know that? I only told
you about the first number!



The lists I gave you were:

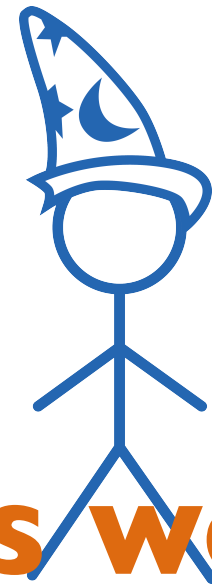
```
[3142, 5798, 6550, 8914]
```

```
[5798, 6550, 8914]
```

```
[6550, 8914]
```

```
[8914]
```

```
[]
```



Why did this work?

The lists I gave you were:

[3142, 5798, 6550, 8914]

[5798, 6550, 8914]

[6550, 8914]

[8914]

Base Case

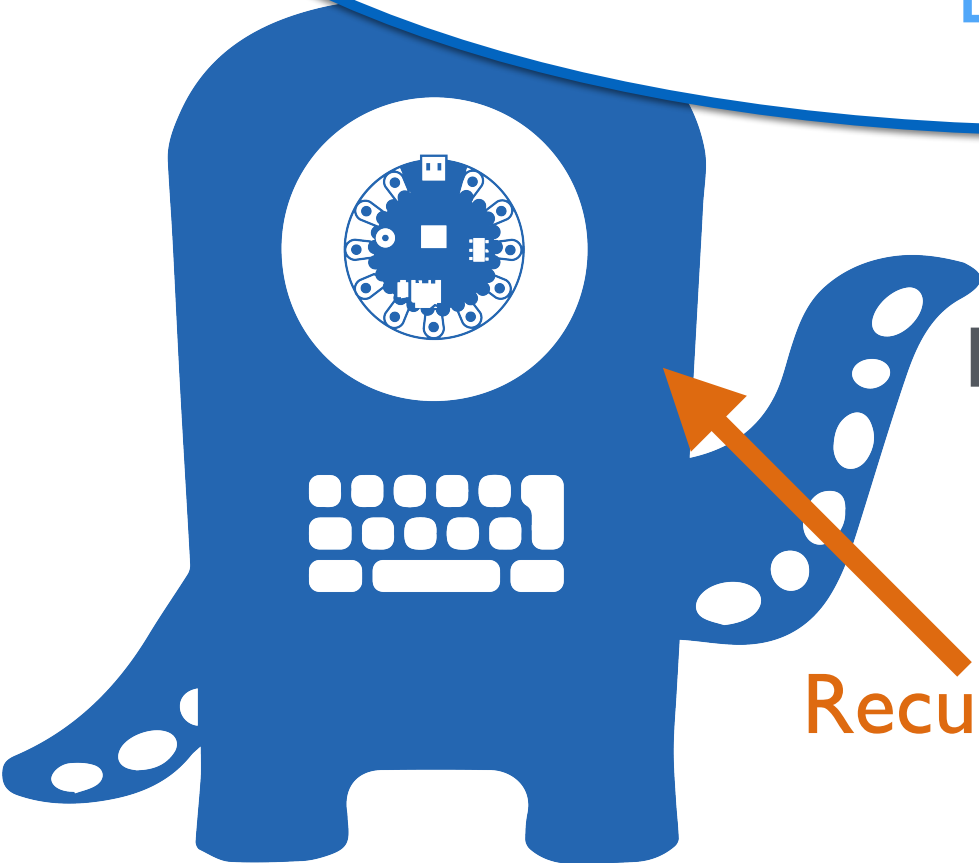
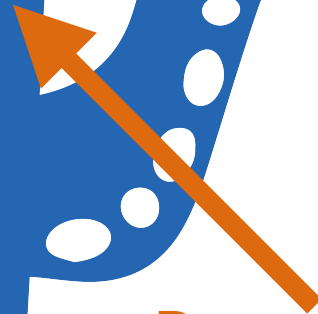


[]

Function Call,
List Mover



Recursive Function



Steps for Recursion

- Know when to stop
- Decide how to take one step
- Break the journey down into that step plus a smaller journey

Steps for Recursion

- When to stop?
- What is the one step?
- How to break the journey down?

Steps for Recursion

- When to stop?
 - When list is empty
- What is the one step?
 - Check the first list item
- How to break the journey down?
 - Look at rest of list

Pseudocode

lst = [3142, 5798, 6550, 8914]

Find firstOdd(lst)

```
def firstOdd(lst):
```

```
    is lst not empty?
```

```
        is first item odd? → return True
```

```
    is lst empty?
```

```
        empty list can't be odd! return False
```



Find firstOdd(rest of lst)

Python

```
lst = [3142, 5798, 6550, 8914]  
firstOdd(lst)
```

```
def firstOdd(lst):  
    if len(lst) > 0:  
        if lst[0] % 2 != 0:  
            return True  
    else:  
        return False
```

```
return firstOdd(lst[1:])
```



Keep going!

Until you see "The End"!

