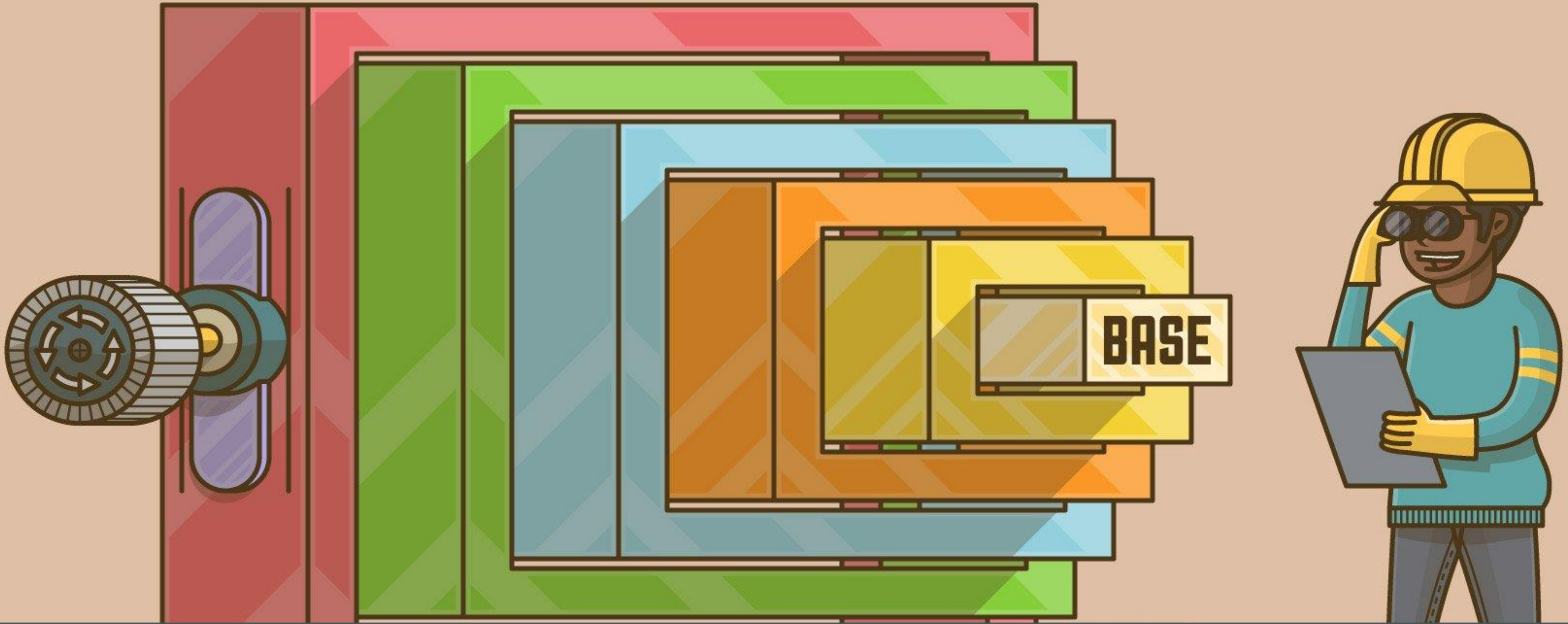
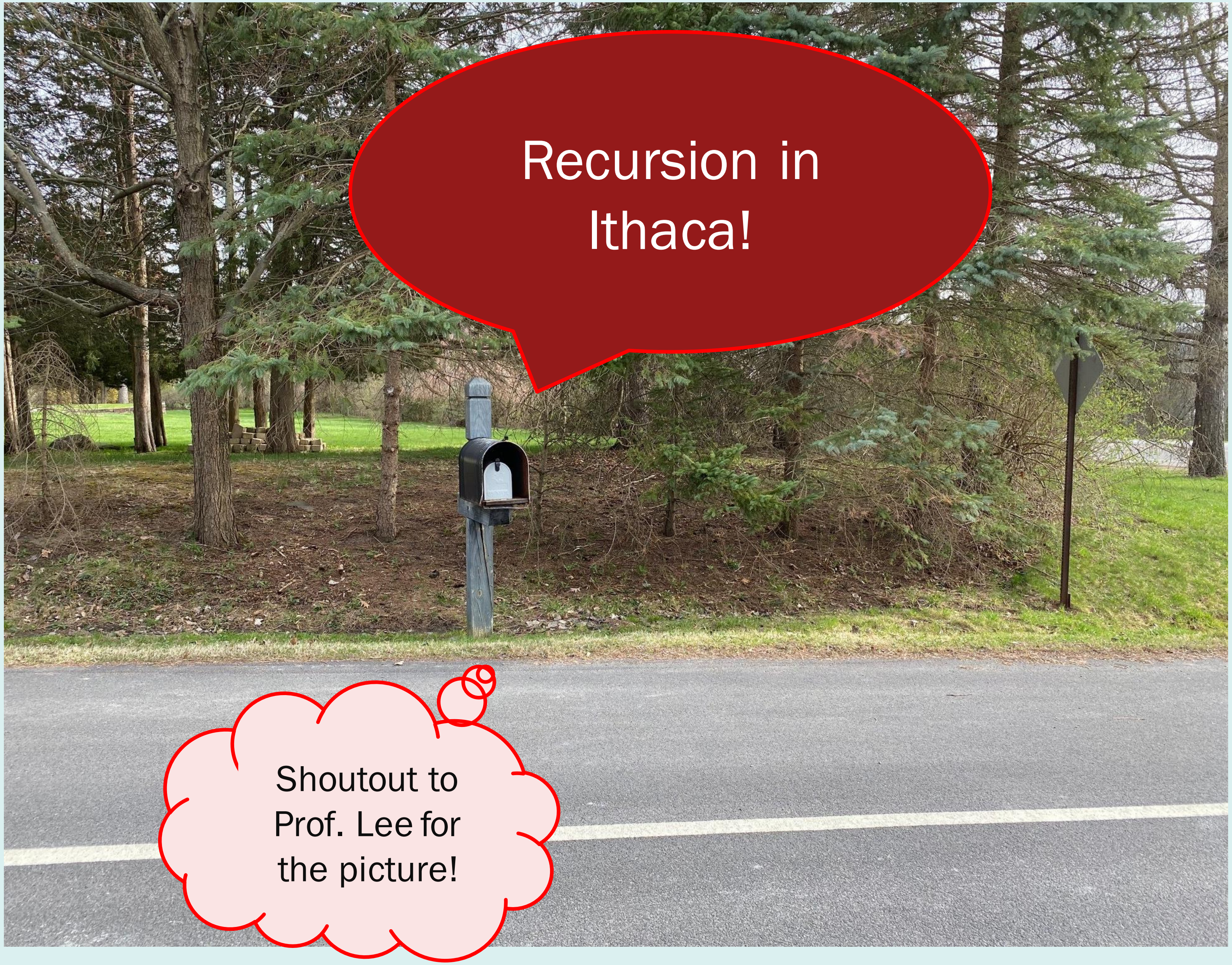


RECURSION REVIEW

PRESENTED BY: NATALIE ISAK

DEFINITION: WHAT IS RECURSION?

- Recursion is using a recursive function.
- What is a recursive function:
  - A function that calls itself
- Main idea: want to break the problem into two cases:
  - A simple case (our base case)
  - A complex case (recursive case), which we will make simpler and then call function



```python
def sum_to_num(n):
    """ Returns the value of the sum of numbers from 1 to up and including n
    Returns (1 + 2 + ... + n-1 + n).

    Precondition: n is a positive integer
    """
```

EXAMPLE ONE: RECURSION WITH INTEGERS

LET'S CODE THIS TOGETHER

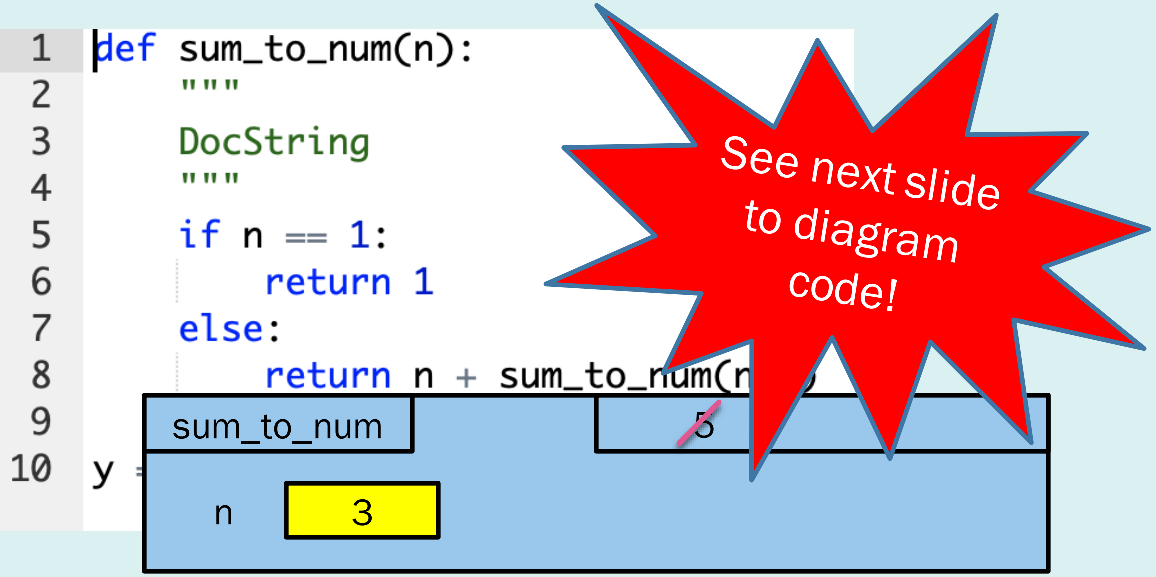
```python
def num_dolls(doll):
    """Returns: number of nesting dolls this doll contains, including itself.
    Example: if `doll` that contains one Doll in it, but that inner
    doll does not contain any Dolls, then this function returns 2.

    Precondition: doll is a Doll object (not None).
    """
```

EXAMPLE TWO: RECURSION WITH DOLLS

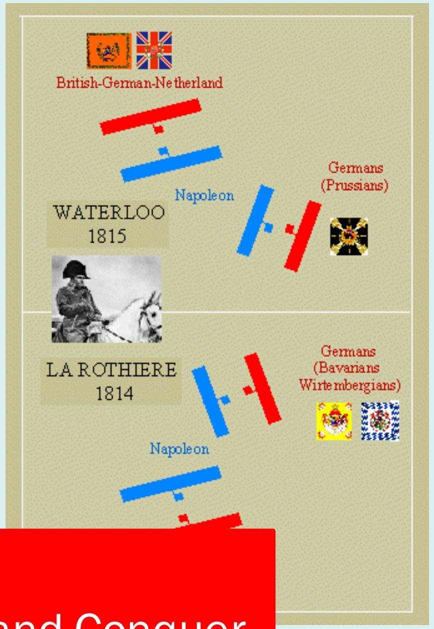
IF YOU WANT ANOTHER EXAMPLE, LET'S CODE THIS TOGETHER

IN THIS PRESENTATION

How Recursion Works (Call Frames)



How to Develop Recursive Function



Divide and Conquer

All students learn in different ways; you may find one or both of these explanations helpful!

CALL FRAMES WITH RECURSION

Let's use call frames to see how sum_to_num() runs to completion.

Code	Global Space	Call Stack

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Python Tutor Link

https://pythontutor.com/visualize.html#code=def%20sum_to_num%28n%29%3A%0A%20%20%20%20%22%22%22%20%0A%20%20%20%20DocString%0A%20%20%20%20%22%22%22%0A%20%20%20%20if%20n%20%3D%3D%201%3A%0A%20%20%20%20%20%20%20%20return%201%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20return%20n%20%2B%20sum_to_num%28n-1%29%0A%20%20%20%20%20%20%20%20%0Ay%20%3D%20sum_to_num%283%29&cumulative=false&heapPrimitives=nevernest&mode=edit&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false

Global Space

Call Stack

Notice: we don't draw the heap because we would draw nothing inside

CALL FRAMES WITH RECURSION

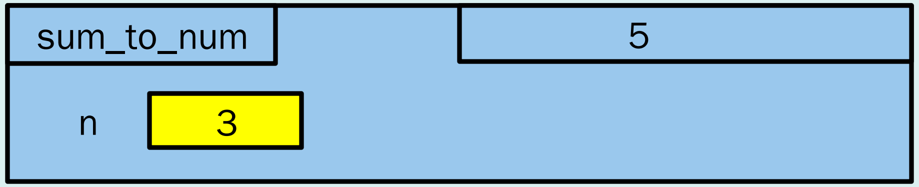
Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack



CALL FRAMES WITH RECURSION

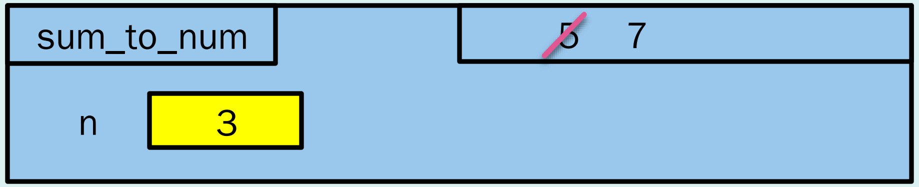
Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack



CALL FRAMES WITH RECURSION

Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack

sum_to_num ~~5~~ ~~7~~ 8

n 3

CALL FRAMES WITH RECURSION

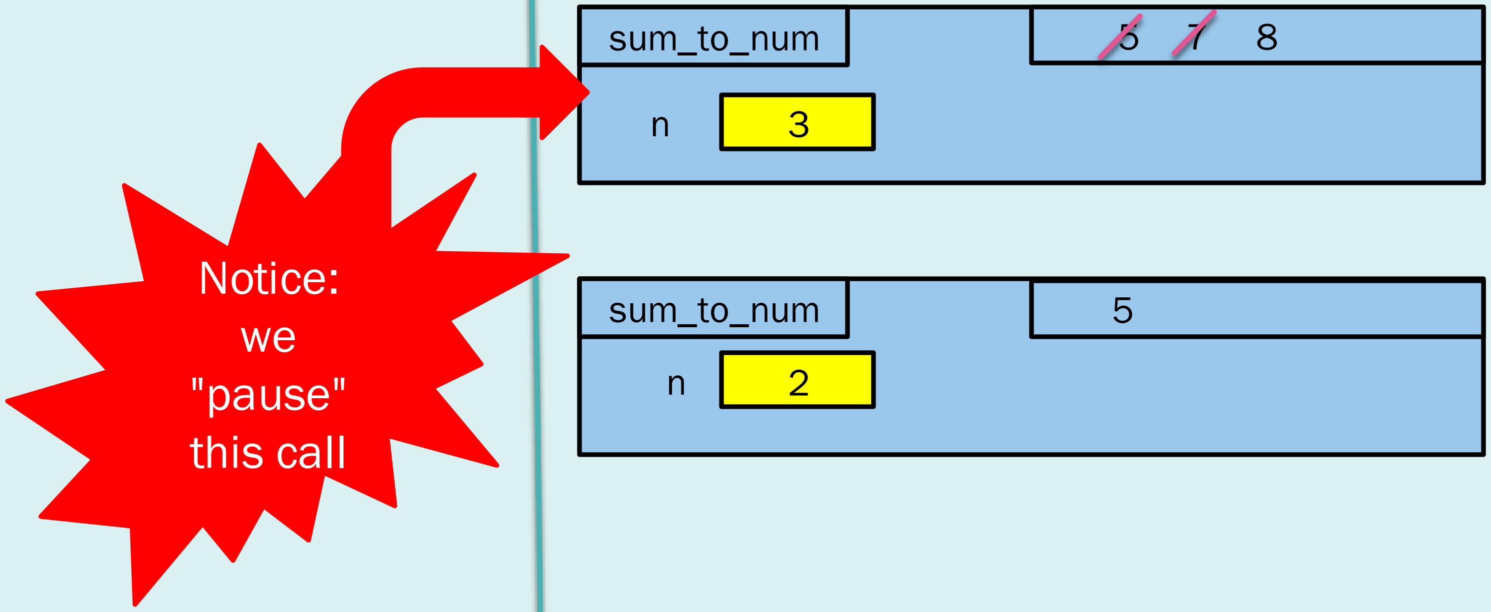
Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack



CALL FRAMES WITH RECURSION

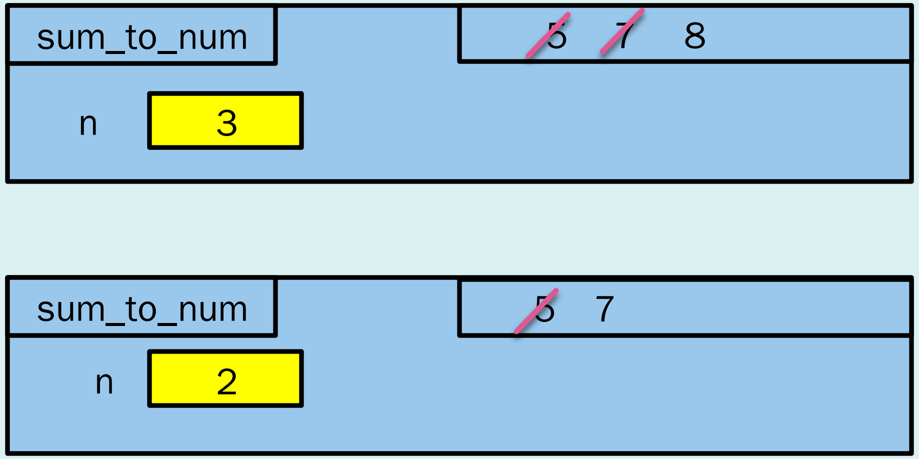
Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack



CALL FRAMES WITH RECURSION

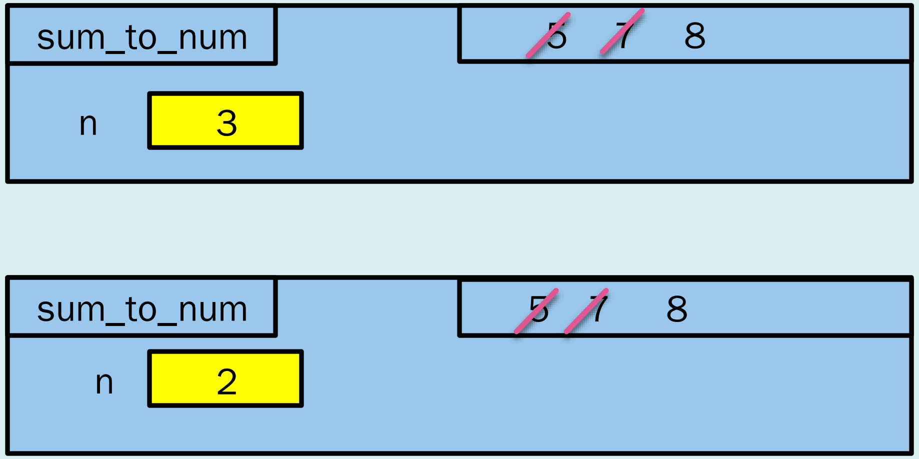
Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack



CALL FRAMES WITH RECURSION

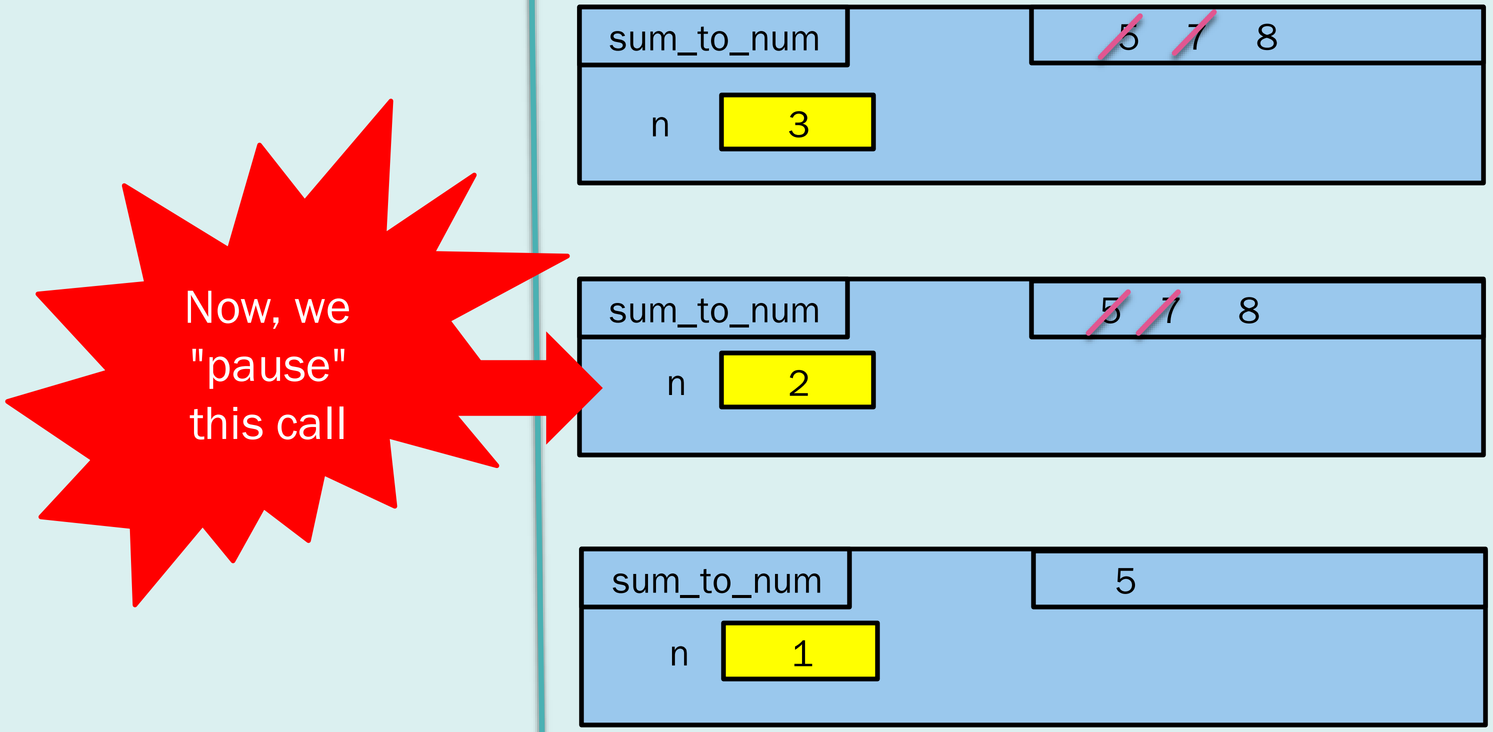
Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack



CALL FRAMES WITH RECURSION

Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack

sum_to_num ~~5~~ ~~7~~ 8

n 3

sum_to_num ~~5~~ ~~7~~ 8

n 2

sum_to_num ~~5~~ 6

n 1

CALL FRAMES WITH RECURSION

Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack

sum_to_num ~~5~~ ~~7~~ 8

n 3

sum_to_num ~~5~~ ~~7~~ 8

n 2

sum_to_num ~~5~~ ~~6~~

n 1 RETURN 1

CALL FRAMES WITH RECURSION

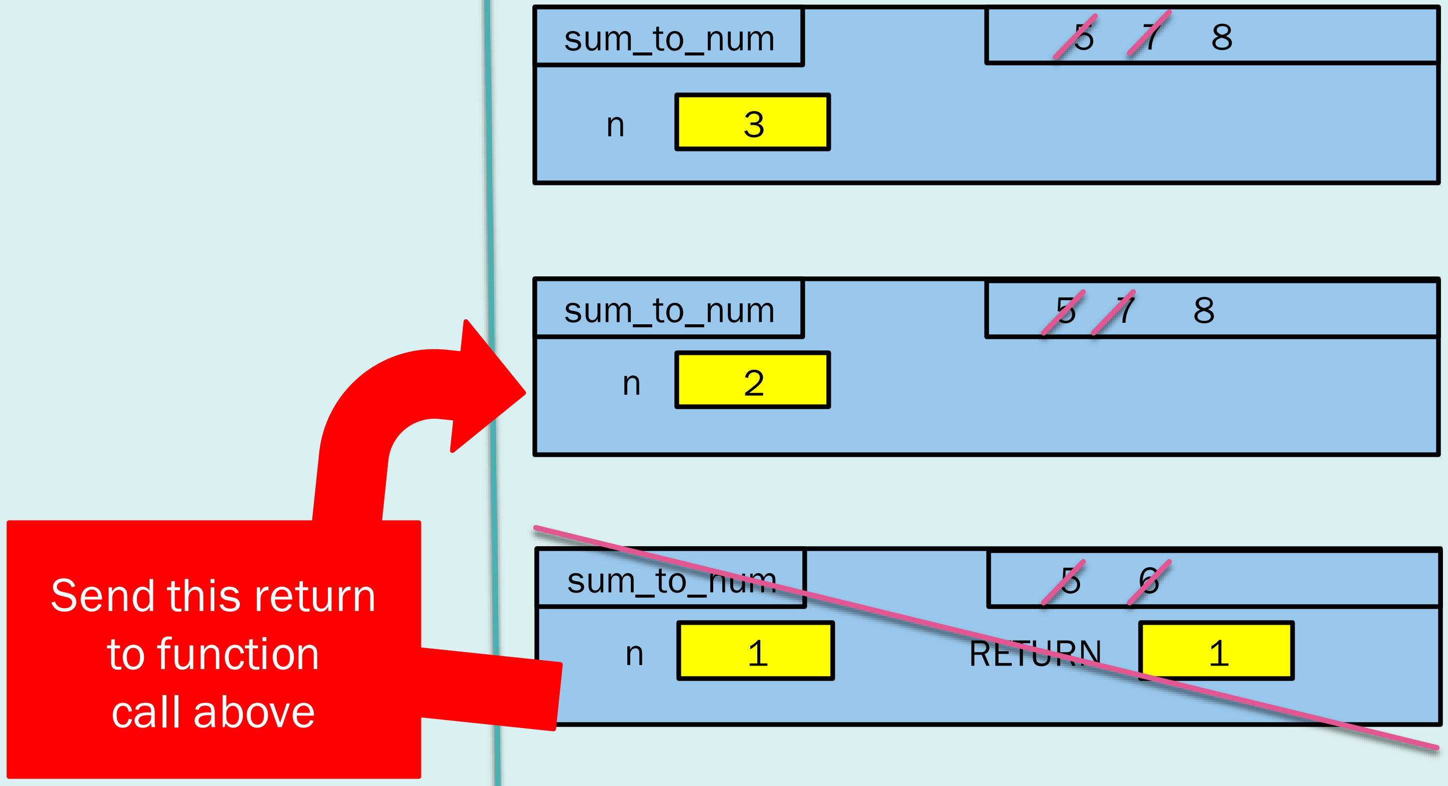
Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

Call Stack



CALL FRAMES WITH RECURSION

Let's use call frames to see how sum_to_num() runs to completion.

Code

```
1  def sum_to_num(n):
2      """
3      DocString
4      """
5      if n == 1:
6          return 1
7      else:
8          return n + sum_to_num(n-1)
9
10 y = sum_to_num(3)
```

Global Space

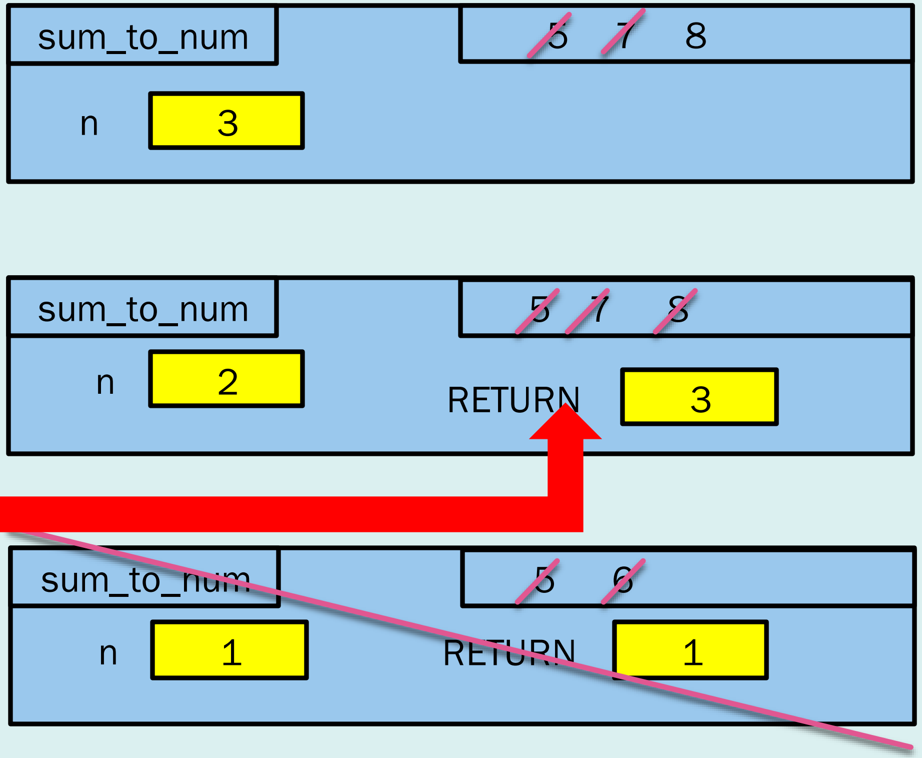
Note: line 8 says to return 2 + sum_to_num(1).

We found sum_to_num(1) is 1

So, return 2 + 1

So, return 3

Call Stack



CALL FRAMES WITH RECURSION

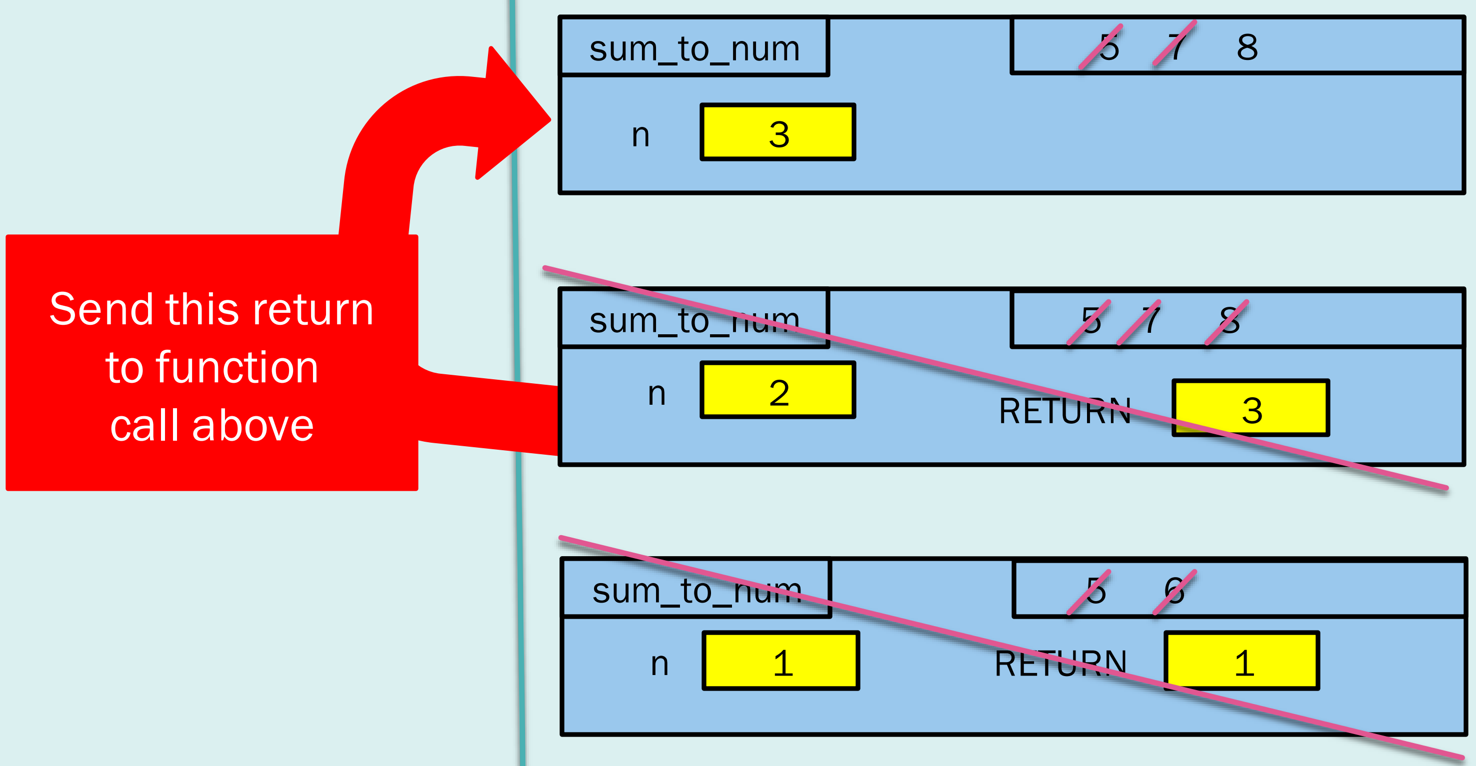
Let's use call frames to see how sum_to_num() runs to completion.

Code

```
1  def sum_to_num(n):
2      """
3      DocString
4      """
5      if n == 1:
6          return 1
7      else:
8          return n + sum_to_num(n-1)
9
10 y = sum_to_num(3)
```

Global Space

Call Stack



CALL FRAMES WITH RECURSION

Let's use call frames to see how sum_to_num() runs to completion.

Code

```
1  def sum_to_num(n):
2      """
3      DocString
4      """
5      if n == 1:
6          return 1
7      else:
8          return n + sum_to_num(n-1)
9
10 y = sum_to_num(3)
```

Global Space

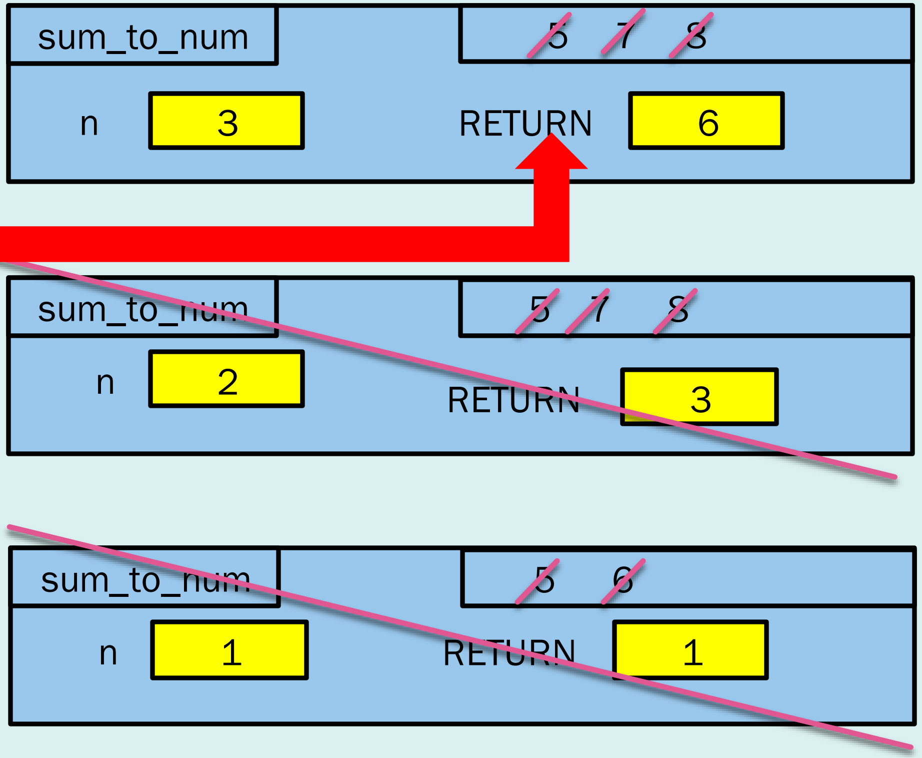
Note: line 8 says to return 3 + sum_to_num(2).

We found sum_to_num(1) is 3

So, return 3 + 3

So, return 6

Call Stack



CALL FRAMES WITH RECURSION

Let's use call frames to see how sum_to_num() runs to completion.

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Global Space

y: 6

Call Stack

~~sum_to_num~~ ~~5~~ ~~7~~ ~~8~~

~~n: 3 RETURN: 6~~

~~sum_to_num~~ ~~5~~ ~~7~~ ~~8~~

~~n: 2 RETURN: 3~~

~~sum_to_num~~ ~~5~~ ~~6~~

~~n: 1 RETURN: 1~~

ANY QUESTIONS ABOUT DIAGRAMING RECURSION?

NEXT TOPIC: DEVELOPING RECURSION

DEVELOPING RECURSION: THREE STEPS (DIVIDE AND CONQUER)

- Step 1: Decide and code your base case(s)
  - This is your simplest case(s)
- Step 2: Develop your recursive part
  - Break up data into two "parts"
  - Multiple ways to do this!
  - Both "parts" should be smaller than original input
  - Call function on these "parts"'
- Step 3: Combine these outputs
  - Must assume smaller answers are correct

Assuming smaller answers are correct helps up develop the function. See "Recursion Fairy"

STEP 1: WHAT'S OUR BASE CASE?

- First step of any recursive call = decide on a base case
- Ways to do this = ask yourself:
  - What is the simplest input I can get?
  - How can I handle this simple input by myself (most likely with just a return or simple calculation)?
  - Are there more than one simple cases? Note: sometimes there are.

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

In this function, our base case was here!

STEP 1: WHAT'S OUR BASE CASE?

But, why do I need a base case?

- If you don't have a base case, your function will never finish!
- When we drew the call frames for sum_to_num(3), the call ran until we reached our base case; then we started returning
- If we have no base case, the function will repeat forever because you don't tell it when to stop.
- Then Python gets mad…

VISUALIZE: RECURSION WITHOUT BASE CASE

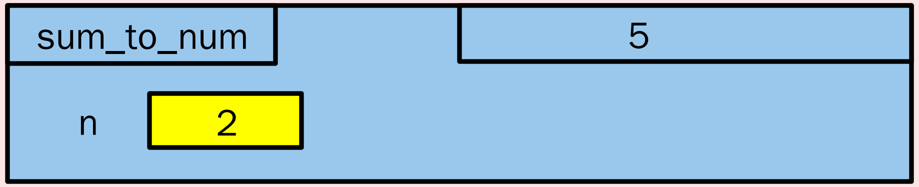
Let's see what happens when we remove the base case to sum_to_num

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    return n + sum_to_num(n-1)

y = sum_to_num(2)
```

Global Space

Call Stack



VISUALIZE: RECURSION WITHOUT BASE CASE

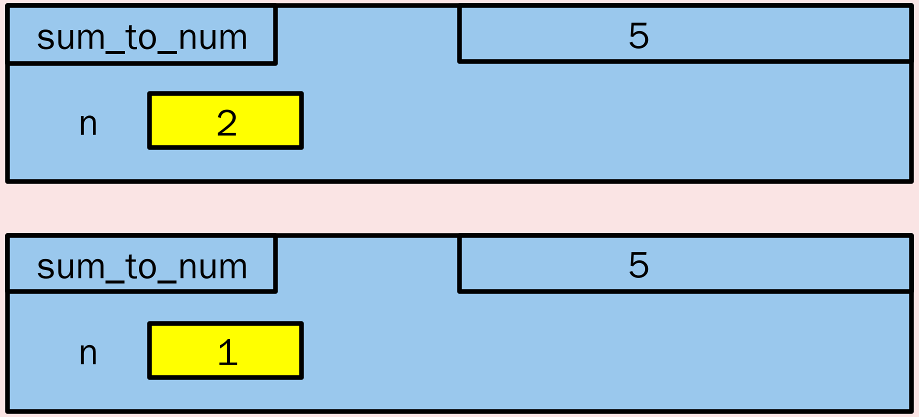
Let's see what happens when we remove the base case to sum_to_num

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    return n + sum_to_num(n-1)

y = sum_to_num(2)
```

Global Space

Call Stack



VISUALIZE: RECURSION WITHOUT BASE CASE

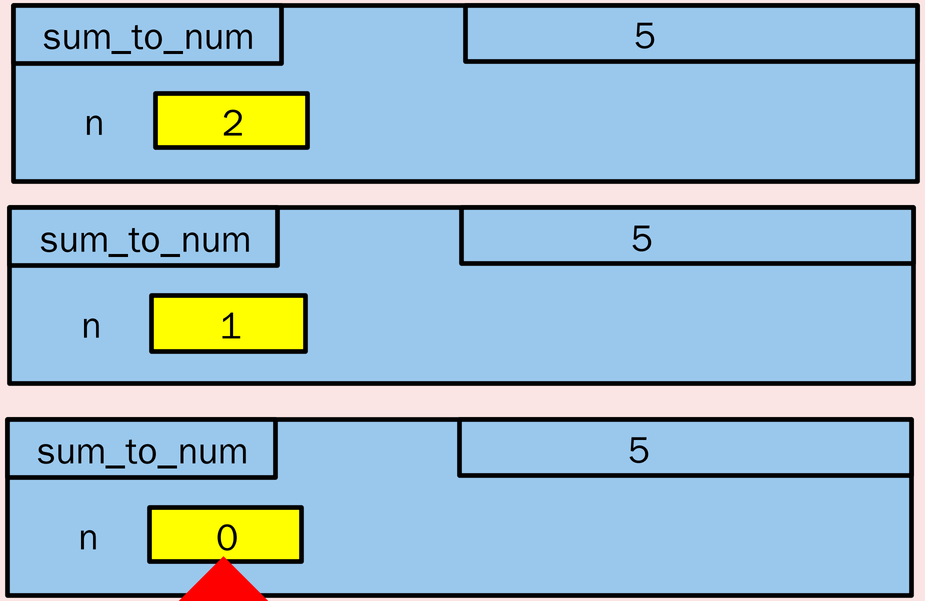
Let's see what happens when we remove the base case to sum_to_num

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    return n + sum_to_num(n-1)

y = sum_to_num(2)
```

Global Space

Call Stack



What you're thinking:
"Uh...but we said n should always be positive, Python..."

VISUALIZE: RECURSION WITHOUT BASE CASE

Let's see what happens when we remove the base case to sum_to_num

Code

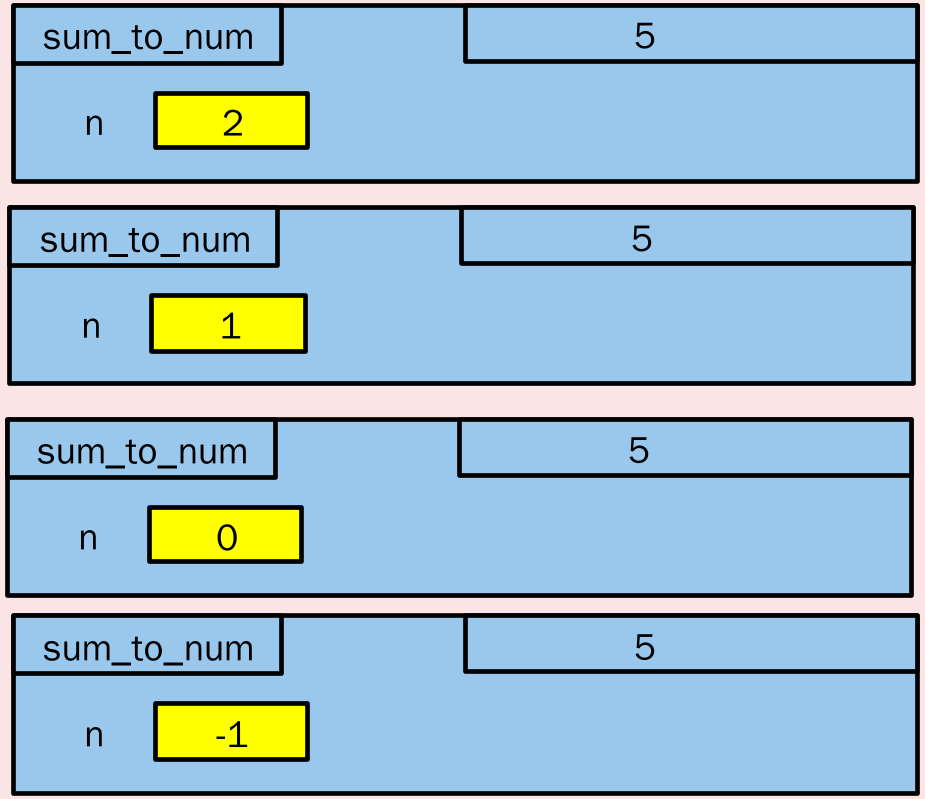
```python
def sum_to_num(n):
    """
    DocString
    """
    return n + sum_to_num(n-1)

y = sum_to_num(2)
```

Global Space

What you're thinking:

"Wait, Python stop!"

Call Stack



VISUALIZE: RECURSION WITHOUT BASE CASE

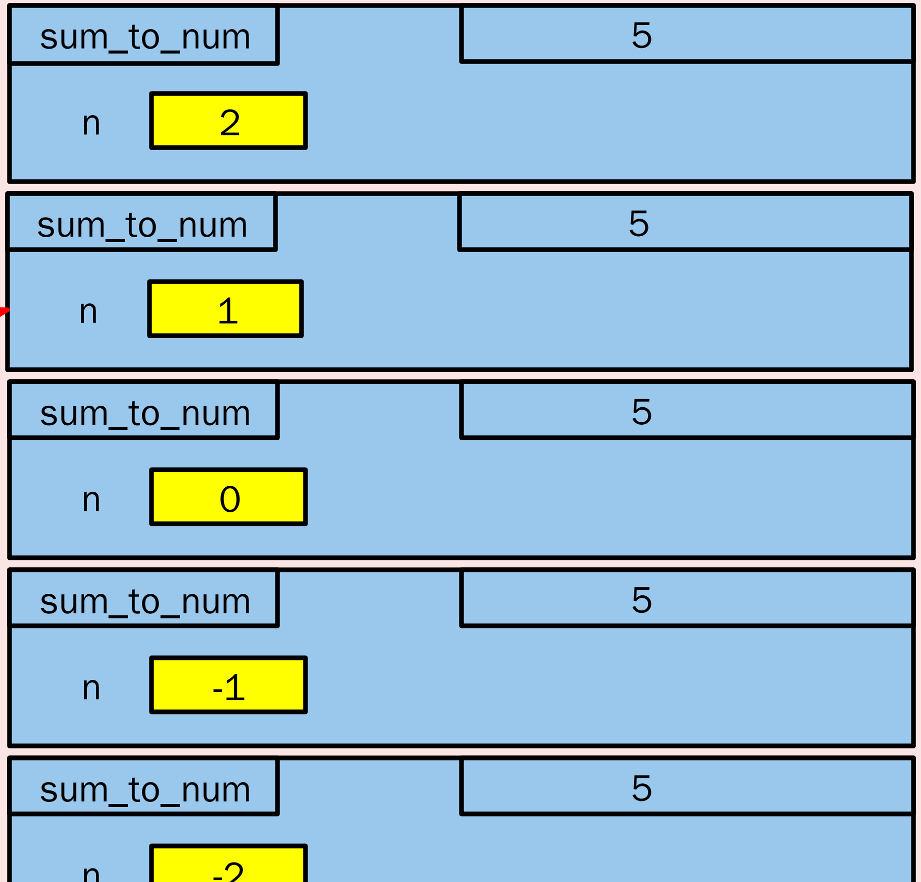
Let's see what happens when we remove the base case to sum_to_num

Code

```python
def sum_to_num(n):
    """
    DocString
    """
    return n + sum_to_num(n-1)

y = sum_to_num(2)
```

Global Space

Call Stack



STEP 1: WHAT'S OUR BASE CASE?

Please don't make us recurse forever

We'll tell you "Maximum Recursion Depth Exceeded" if you do.

- But, Python can't stop
- You didn't give it a base case
- So, Python doesn't know when to stop.
- So, keep Python happy: include a base case

STEP 2: DEVELOP RECURSIVE PART

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

Part 2: Notice we call sum_to_num on a smaller input than n (we call it on n-1), meaning our input gets closer to the base case.

Part 1: just the original number. This is smaller because we want to add all numbers from 1 to n. n is just one of these number. However, we require that the piece in sum_to_num is smaller than n.

- In this step, we need to decide how to divide our input
- Often many ways to divide
- Sometimes type of division depends on type of the input
- Then, call the function on these parts
- The part you call the function on must ALWAYS be "smaller" than our original input!
    - Small means closer to termination, not just smaller value; see note to left for more information

STEP 2: DEVELOP RECURSIVE PART

<u>But, why do I need to call the function on a smaller input?</u>

- Like when we forget a base case, the function will not be able to terminate.
- In this case, Python has a something that tells it to "stop" (a base case) but since we just call the function on the same n over and over, it never reaches that base case.

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n)

y = sum_to_num(3)
```

- Thus, Python recurses "forever"
- And gets angry again

STEP 2: DEVELOP RECURSIVE PART

How to split different types of inputs

Objects

- Sometimes, objects contain a smaller part. For instance, each doll object may contain a doll. If it does, that doll is a "smaller doll."

```
def open_doll(d):
    """Input: a Russian Doll
     Opens the Russian Doll d """
     print("My name is "+ d.name)
     if d.hasSeam:
         inner = d.innerDoll
         open_doll(inner)
     else:
         print("That's it!")
```

Integers

- To make an integer smaller, subtracting or dividing comes to mind. This is how we make an integer smaller for recursion.

```
def blast_off(n):
   """Input: a non-negative int
    Counts down from n to Blast-Off!
    """
   if (n == 0):
         print("BLAST OFF!")
   else:
         print(n)
         blast_off(n-1)
```

Both Examples
Are From Lecture13

STEP 2: DEVELOP RECURSIVE PART

How to split different types of inputs

Strings

- Slicing Strings is a common way to split them. We can slice the string into halves or make one part really small (just the first character) and the other part really big (the rest of the string)

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])

    # 3. Combine the result
    return left+right
```

This is from Lecture 13

Lists

- Like strings, we can also slice lists. However, another common method is to use a for loop to get parts of a list. Notice, in the line of code

for item in t_list:

- item will store each part of the list. Thus, item is the "part" of the list we want to (perhaps) call the function on.
- We will do an example of this in a few slides.

STEP 3: COMBINE THE OUTPUTS

- Once we finish splitting the input and calling the function on these inputs, we must combine the outputs together.
- This is sometimes hard to do.
- Students ask many questions like:
  - How do I know what the function gives me back?
  - What is the type of the return value of the function?
  - How do I combine values when I don't know what they are

    - To do this, we usually "assume" our function works properly, reading the specification to tell us what the output will be.
    - We can also use the "Recursion Fairy"

```python
def sum_to_num(n):
    """
    DocString
    """
    if n == 1:
        return 1
    else:
        return n + sum_to_num(n-1)

y = sum_to_num(3)
```

We combined our inputs here.

STEP 3: COMBINE THE OUTPUTS

"Recursion Fairy"

- It is hard to combine the "parts" from step 2.
- We assume the function works correctly.
- Or, assume the "Recursion Fairy" takes a function call and returns the correct answer for you, meaning you can assume the answer is correct while writing your code.

Recall num_dolls():

```
def num_dolls(doll):
    """Returns: number of nesting dolls this doll contains, including itself.
    Example: if `doll` that contains one Doll in it, but that inner
    doll does not contain any Dolls, then this function returns 2.

    Precondition: doll is a Doll object (not None).
    """
```

STEP 3: COMBINE THE OUTPUTS

"Recursion Fairy"

- Let's say we had the code below.
- How do we combine 1 and num_dolls(doll.innerDoll)?

```python
def num_dolls(doll):
    """Returns: number of nesting dolls this doll contains, including itself.
    Example: if `doll` that contains one Doll in it, but that inner
    doll does not contain any Dolls, then this function returns 2.

    Precondition: doll is a Doll object (not None).
    """

    #BEGIN_REMOVE
    if  not doll.hasSeam:
        return 1
    else:
        return 1 ___ num_dolls(doll.innerDoll)
```

What goes here?

STEP 3: COMBINE THE OUTPUTS

"Recursion Fairy"

I assume this function works correct so num_dolls(doll.innerDoll) will return an integer that represents the number of dolls in doll.innerDoll plus that doll.

- Recursion Fairy swoops in and tells us what the value of num_dolls(doll.innerDoll) will be according to the spec.
- According to the Fairy, this out will be the number of dolls inside doll.innerDoll, including doll.innerDoll.
- Thus, the output will be an integer
  - We assume the function will do what we want!
- So, how should we combine 1 and num_dolls(doll.innerDoll)?

```
def num_dolls(doll):
    """Returns: number of nesting dolls
    Example: if `doll` that contains one
    doll does not contain any Dolls, the

    Precondition: doll is a Doll object
    """

    #BEGIN_REMOVE
    if  not doll.hasSeam:
        return 1
    else:
        return 1 ___ num_dolls(doll.innerDoll)
```

```python
def embed(theinput):
    """Returns: depth of embedding, or nesting, in theinput.

    Examples:
        "the dog that barked" -> 0
        ["the", "dog", "that", "barked"] -> 1
        ["the" ["dog", "that", "barked"]] -> 2
        ["the" [[["dog"]], ["that", "barked"]] -> 3
        ["the" ["dog", ["that", ["barked"]] -> 4
        [[[["the"], "dog"], "that"], "barked"] -> 4

    Precondition: theinput is a string, or a potentially nested
    list of strings. No component list can be empty"""
```

EXAMPLE THREE: RECURSION OVER LISTS

LET'S CODE THIS TOGETHER

```python
def prefix(s):
    """Returns the prefix (identical characters at the start) length of s
    Example: prefix('abc') returns 1 as the prefix is 'a'
                 prefix('xxxxxxyzx') returns 6 as the prefix is 'xxxxxx'
                 prefix('') returns 0 as the string is empty
    Precondition: s is a (possibly empty) string of lowercase letters"""
```

EXAMPLE FOUR: RECURSION OVER STRINGS

TRY THE SLICING METHOD!

YOU TRY THIS FUNCTION!!!

```python
def prefix(s):
    """Returns the prefix (identical characters at the start) length of s
    Example: prefix('abc') returns 1 as the prefix is 'a'
             prefix('xxxxxxyzx') returns 6 as the prefix is 'xxxxxx'
             prefix('') returns 0 as the string is empty
    Precondition: s is a (possibly empty) string of lowercase letters"""
```

EXAMPLE FOUR: RECURSION OVER STRINGS

LET'S GO OVER THIS FUNCTION TOGETHER NOW

```python
def decode(nlist):
    """Returns a string that represents the decoded nlist
    The nlist is a list of lists, where each element is a character and
    a number. The number is the number of times to repeat the character.

    Example: decode([['a',3],['h',1],['a',1]]) is 'aaaha'
    Example: decode([]) is ''

    Precondition: nlist is a (possibly empty) nested list of two-element lists,
    where each list inside is a pair of a character and an integer"""
```

EXAMPLE FIVE: RECURSION OVER LISTS PT. 2

YOU CAN TRY THE SLICING METHOD FOR THIS!

YOU TRY THIS FUNCTION!!!

```python
def decode(nlist):
    """Returns a string that represents the decoded nlist
    The nlist is a list of lists, where each element is a character and
    a number. The number is the number of times to repeat the character.

    Example: decode([['a',3],['h',1],['a',1]]) is 'aaaha'
    Example: decode([]) is ''

    Precondition: nlist is a (possibly empty) nested list of two-element lists,
    where each list inside is a pair of a character and an integer"""
```

EXAMPLE FIVE: RECURSION OVER LISTS PT. 2

LET'S GO OVER THIS FUNCTION TOGETHER NOW

ANY QUESTIONS?

THANK YOU ALL FOR COMING!

PHOTO CITES

- Thinking-Recursively-in-Python_Watermarked.1825397c00ea.jpg
- napoleon_defeats.gif
- hand-paint-fairy-watercolor-vector-silhouette-illustration-magic-wand-165888633.jpg

- Recursion fairy idea suggested by Jeff Erikson: https://cs.stackexchange.com/questions/30712/teaching-recursion