http://www.cs.cornell.edu/courses/cs1110/2022sp

Lecture 14:
More Recursion!

CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

Lecture Afterthoughts

- Slide 34 had a typo! Should be:

```
for parent in p.parents:
```

- Slide 30 & 35 now has folders to better understand the Person class and its attributes

Announcements

- Reminder: prelim 1 regrade requests due on Gradescope Wed 11:59pm

  "When you review your prelim, if you believe a grading error was made, you may request a regrade on Gradescope until 11:59pm Wed Mar 23. We plan to handle all the regrade requests in one pass, after the regrade-request window has closed."

Recursion

Recursive Function:

A function that calls itself (directly or indirectly)

Recursive Definition:

A definition that is defined in terms of itself

From previous lecture: Factorial

Non-recursive definition:

$n! = n \times n\text{-}1 \times \ldots \times 2 \times 1$

$= n\ (n\text{-}1 \times \ldots \times 2 \times 1)$

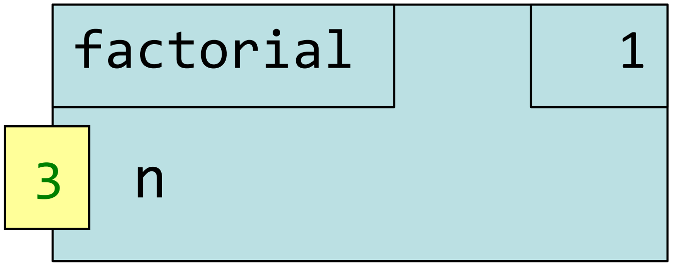
Recursive definition:

$n! = n\ (n\text{-}1)!$ for $n > 0$ Recursive case

$0! = 1$ Base case

Recursive Call Frames

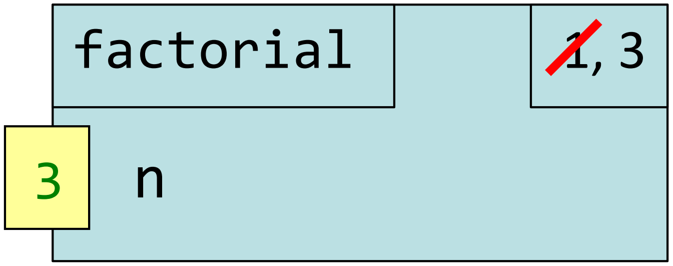
```
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int"""
    if n == 0:
        return 1
    return n*factorial(n-1)
```



```
factorial(3)
```

Recursive Call Frames

```
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int"""
    if n == 0:
        return 1
    return n*factorial(n-1)
```



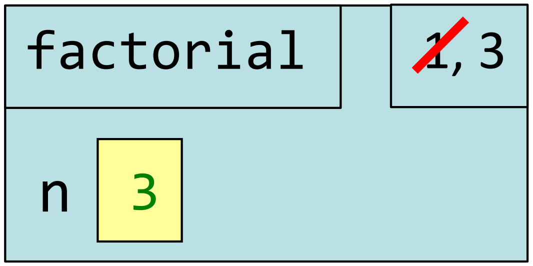
```
factorial(3)
```

Now what?

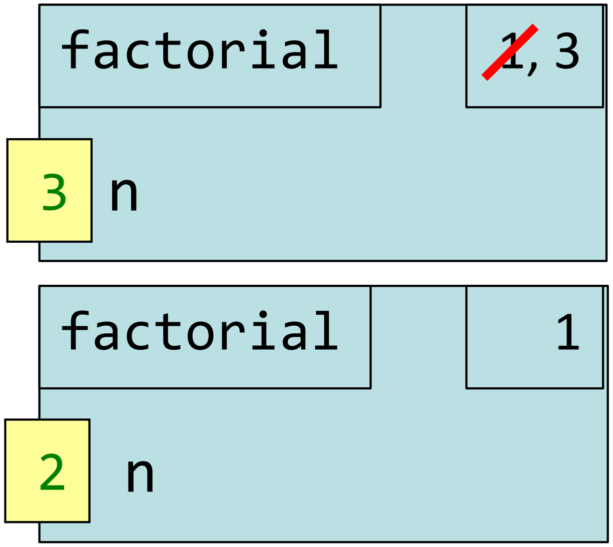
Each call is a new frame!

What happens next? (Q)

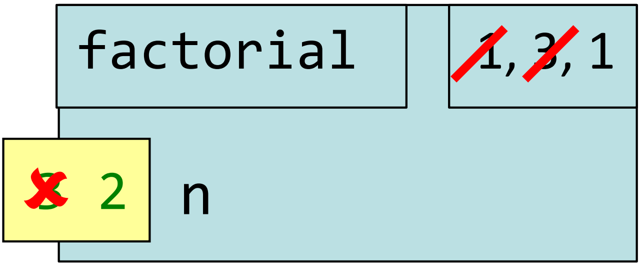
```
def factorial(n):
   """Returns: factorial o
   Precondition: n ≥ 0 an
   if n == 0:
      return 1
   return n*factorial(n-1)


factorial(3)
```



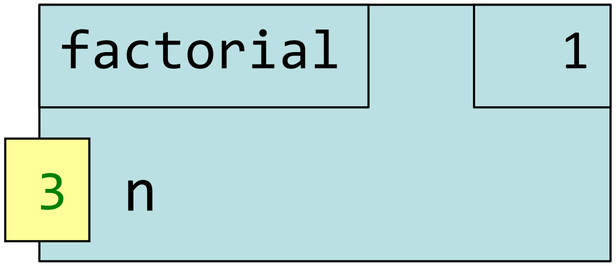
A:



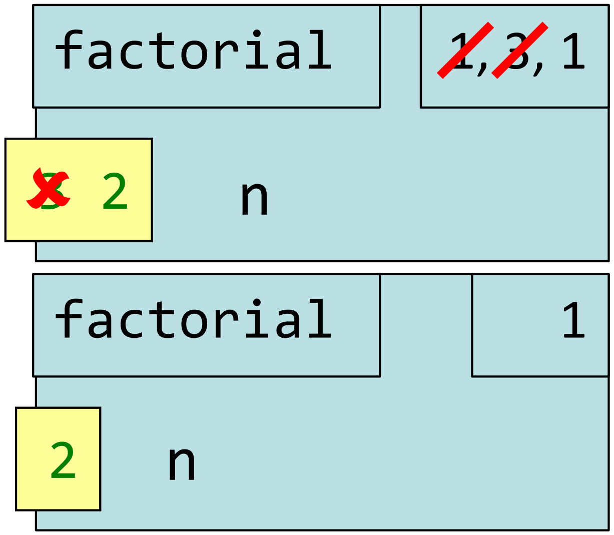
B:



C: ERASE FRAME

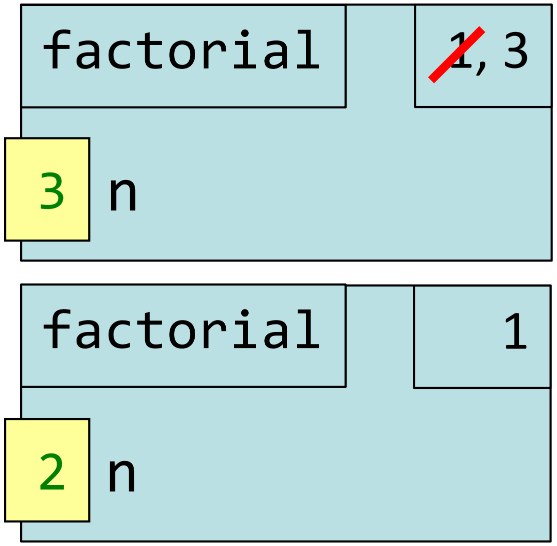


D:



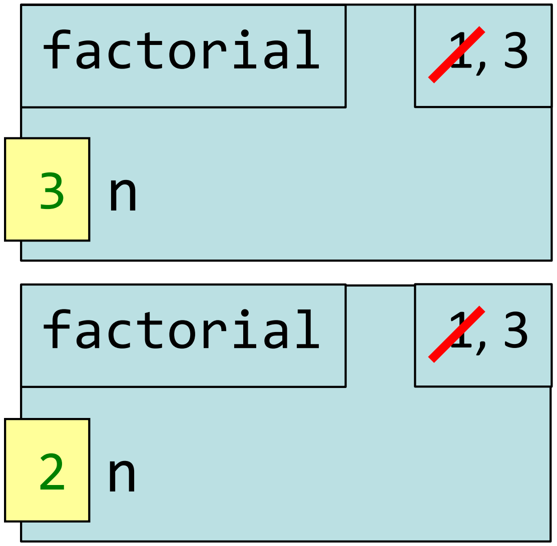
Recursive Call Frames (n==2, execute line 1)

```
  def factorial(n):
     """Returns: factorial of n.
     Precondition: n ≥ 0 an int"""
1    if n == 0:
2        return 1
3    return n*factorial(n-1)

  factorial(3)
```



Recursive Call Frames (n==2, execute line 3)

```
  def factorial(n):
      """Returns: factorial of n.
      Precondition: n ≥ 0 an int"""
1     if n == 0:
2         return 1
3     return n*factorial(n-1)

  factorial(3)
```



factorial

~~1~~, 3

3 n

factorial

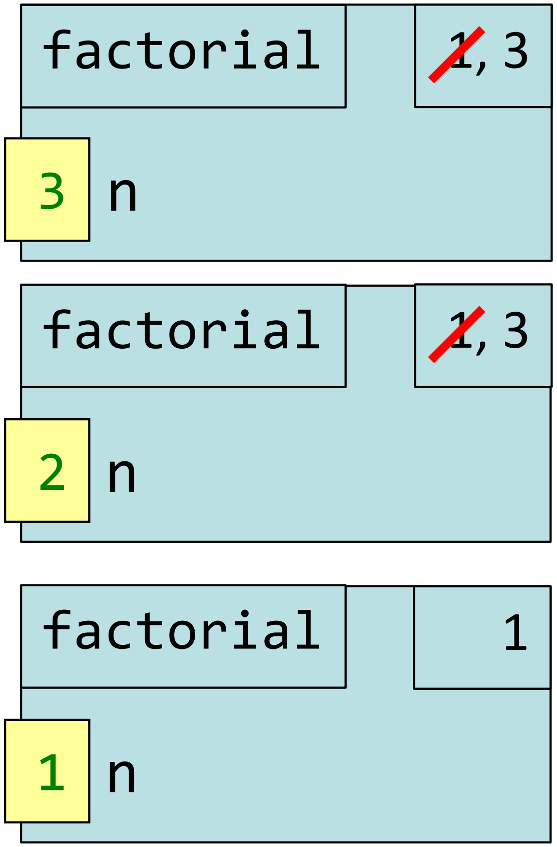
~~1~~, 3

2 n

Recursive Call Frames (n==1, execute line 1)

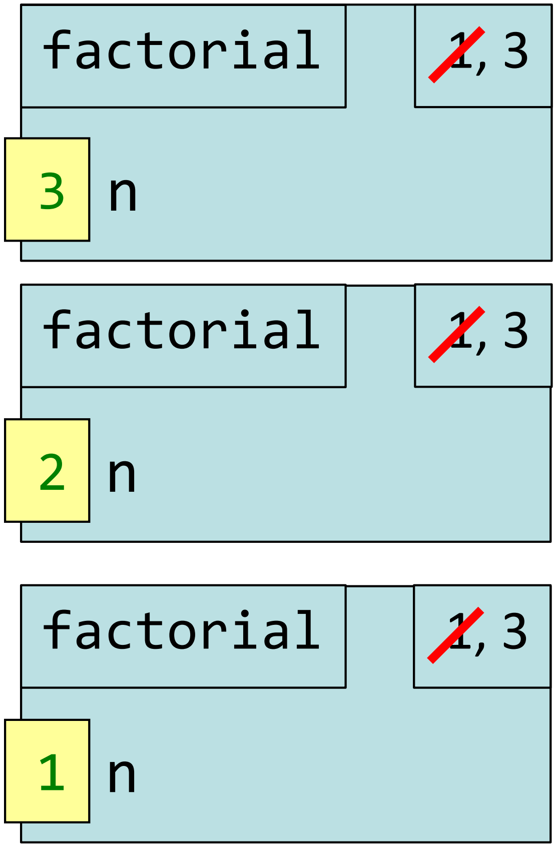
```
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int"""
1   if n == 0:
2       return 1
3   return n*factorial(n-1)
```

```
factorial(3)
```



Recursive Call Frames (n==1, execute line 3)

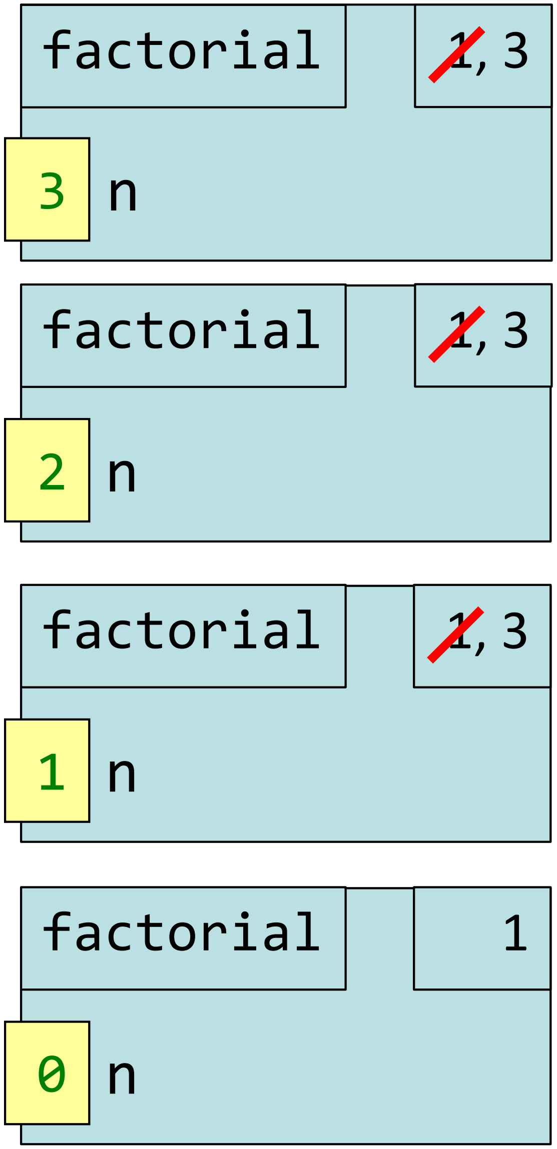
```
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int"""
1   if n == 0:
2       return 1
3   return n*factorial(n-1)


factorial(3)
```



Recursive Call Frames (n==0, execute line 1)

```
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int"""
1   if n == 0:
2       return 1
3   return n*factorial(n-1)
```

```
factorial(3)
```



factorial ~~1~~, 3

3 n

factorial ~~1~~, 3

2 n

factorial ~~1~~, 3

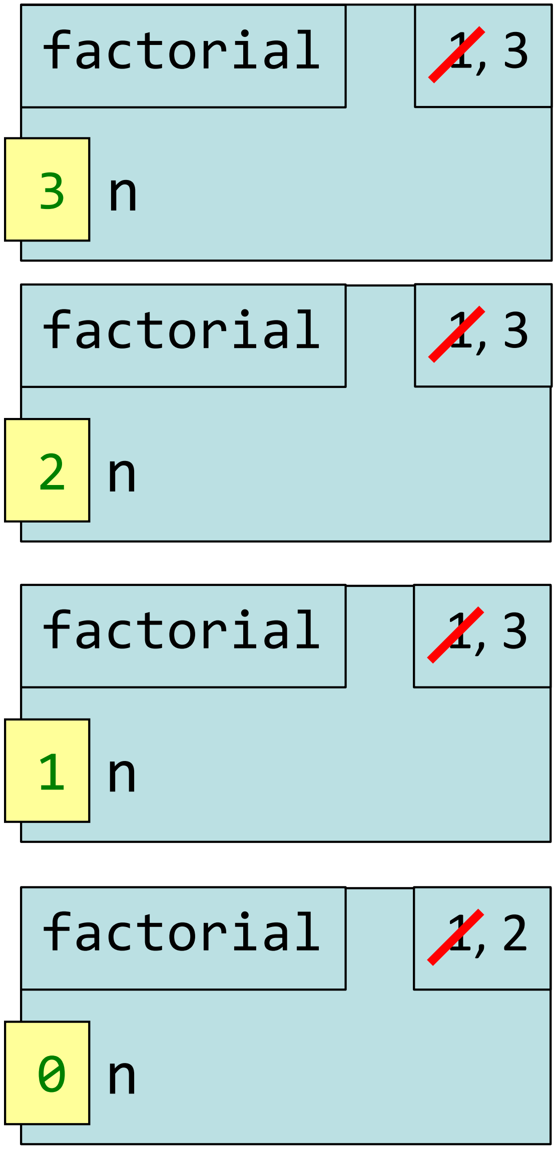
1 n

factorial 1

0 n

Recursive Call Frames (n==0, execute line 2)

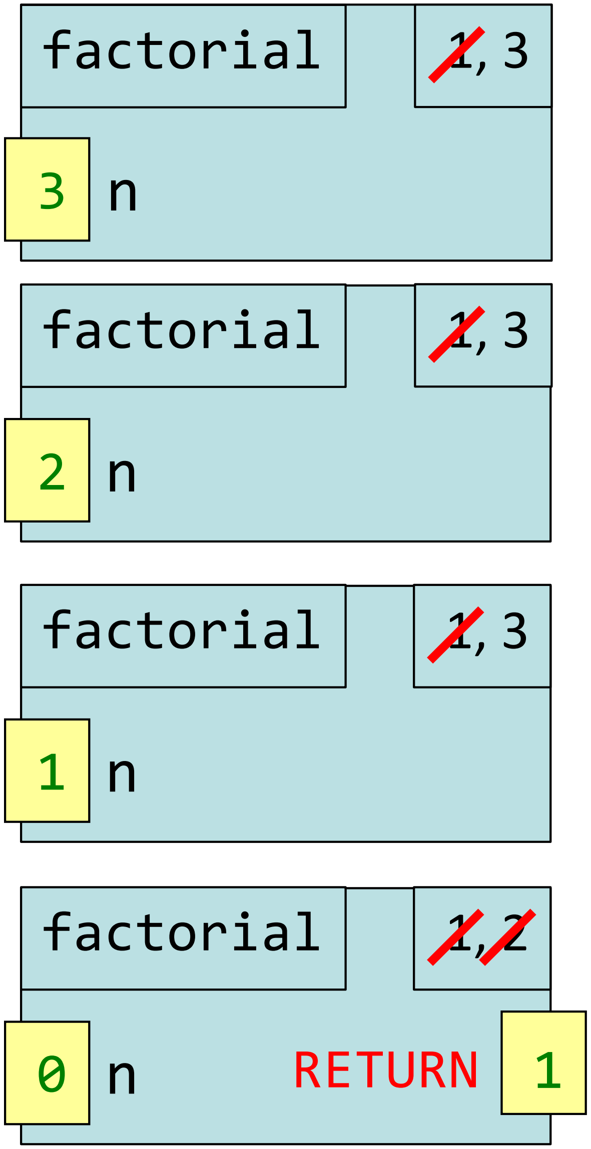
```
    def factorial(n):
        """Returns: factorial of n.
        Precondition: n ≥ 0 an int"""
1       if n == 0:
2           return 1
3       return n*factorial(n-1)


    factorial(3)
```



Recursive Call Frames (n==0, RETURN 1)

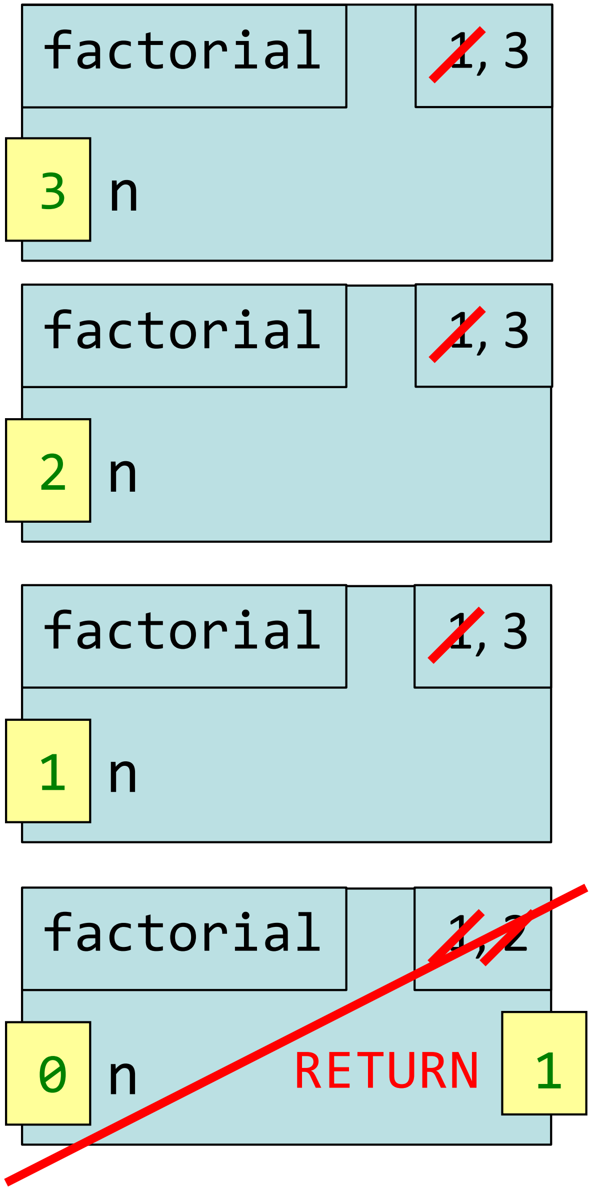
```
  def factorial(n):
     """Returns: factorial of n.
     Precondition: n ≥ 0 an int"""
1    if n == 0:
2       return 1
3    return n*factorial(n-1)
```

```
factorial(3)
```



Recursive Call Frames (n==1, finish line 3)

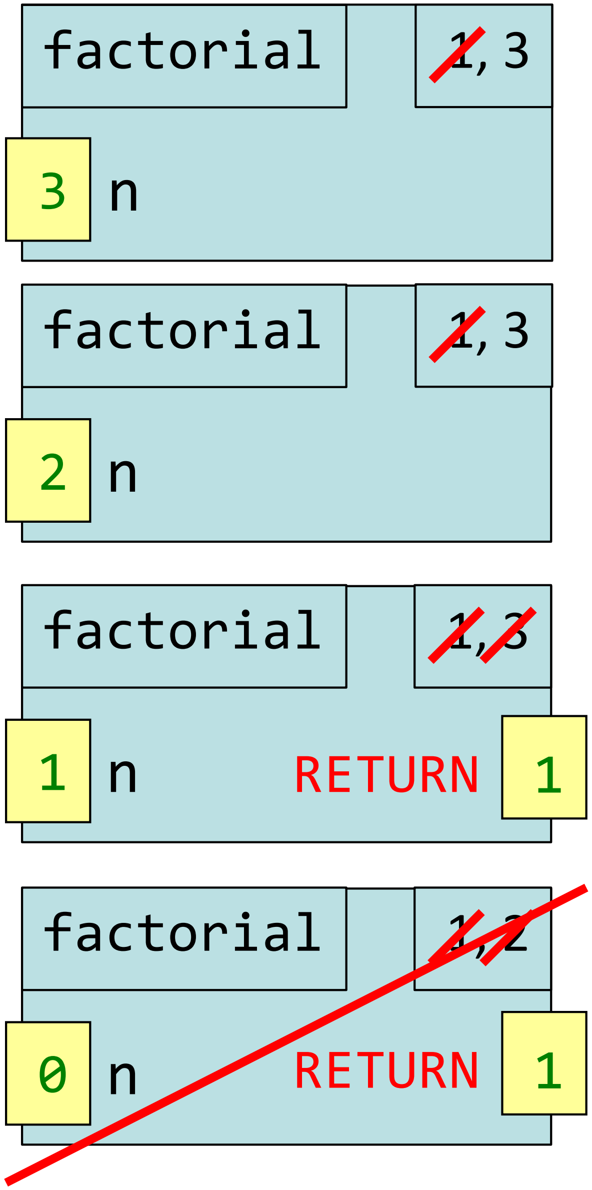
```
  def factorial(n):
      """Returns: factorial of n.
      Precondition: n ≥ 0 an int"""
1     if n == 0:
2         return 1
3     return n*factorial(n-1)


  factorial(3)
```



Recursive Call Frames (n==1, RETURN 1)

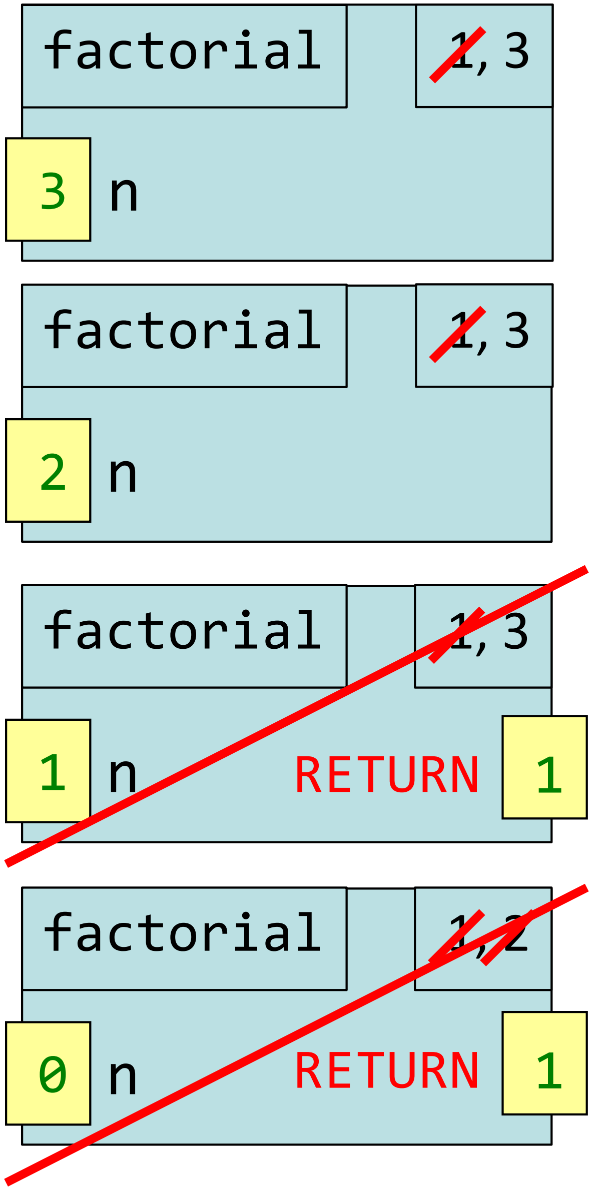
```
  def factorial(n):
      """Returns: factorial of n.
      Precondition: n ≥ 0 an int"""
1     if n == 0:
2         return 1
3     return n*factorial(n-1)
```

factorial(3)



Recursive Call Frames (n==2, finish line 3)

```
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int"""
1   if n == 0:
2       return 1
3   return n*factorial(n-1)


factorial(3)
```



factorial ~~1~~, 3

3 n

factorial ~~1~~, 3

2 n

~~factorial 1, 3~~

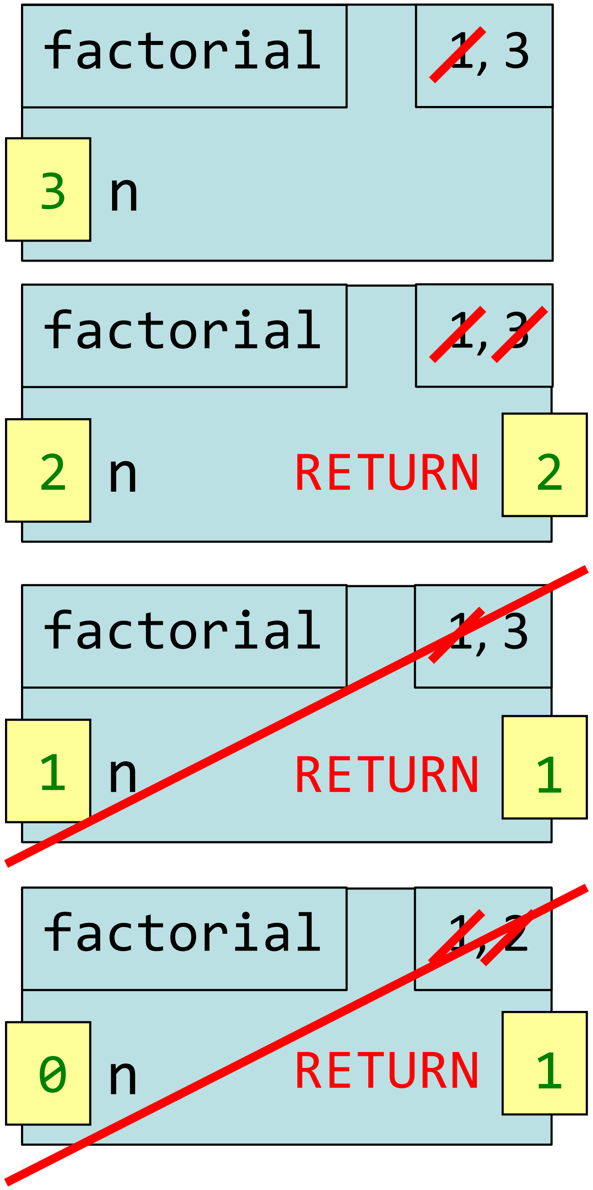
~~1 n RETURN 1~~

~~factorial 1, 2~~

~~0 n RETURN 1~~

Recursive Call Frames (n==2, RETURN 6)

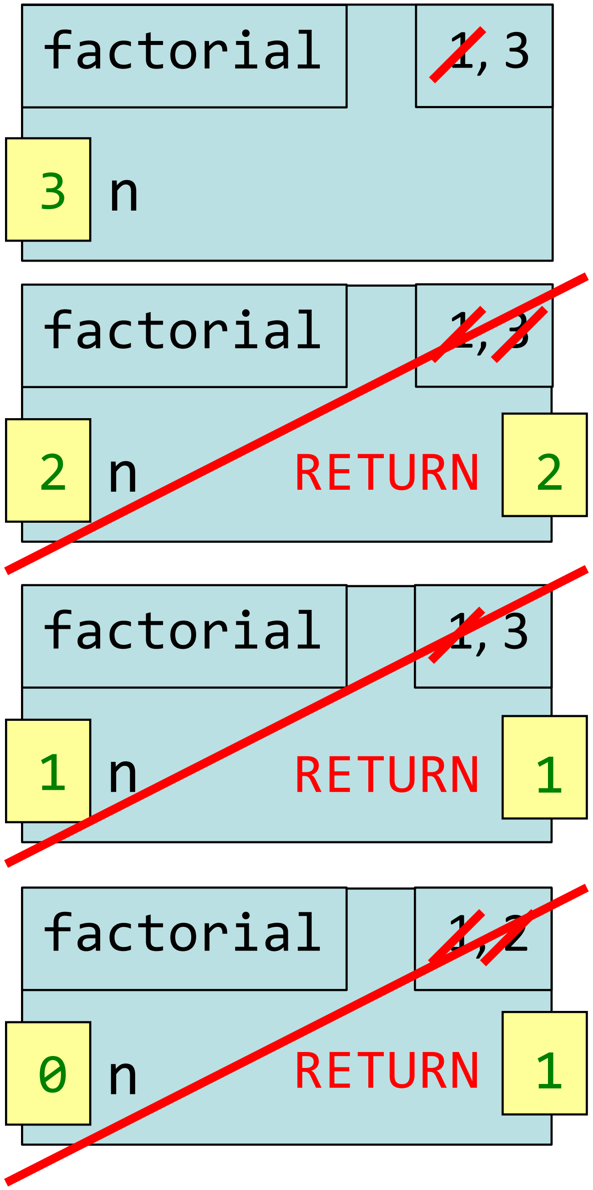
```
  def factorial(n):
      """Returns: factorial of n.
      Precondition: n ≥ 0 an int"""
1     if n == 0:
2         return 1
3     return n*factorial(n-1)


  factorial(3)
```



Recursive Call Frames (n==3, finish line 3)

```
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int"""
1   if n == 0:
2       return 1
3   return n*factorial(n-1)
```

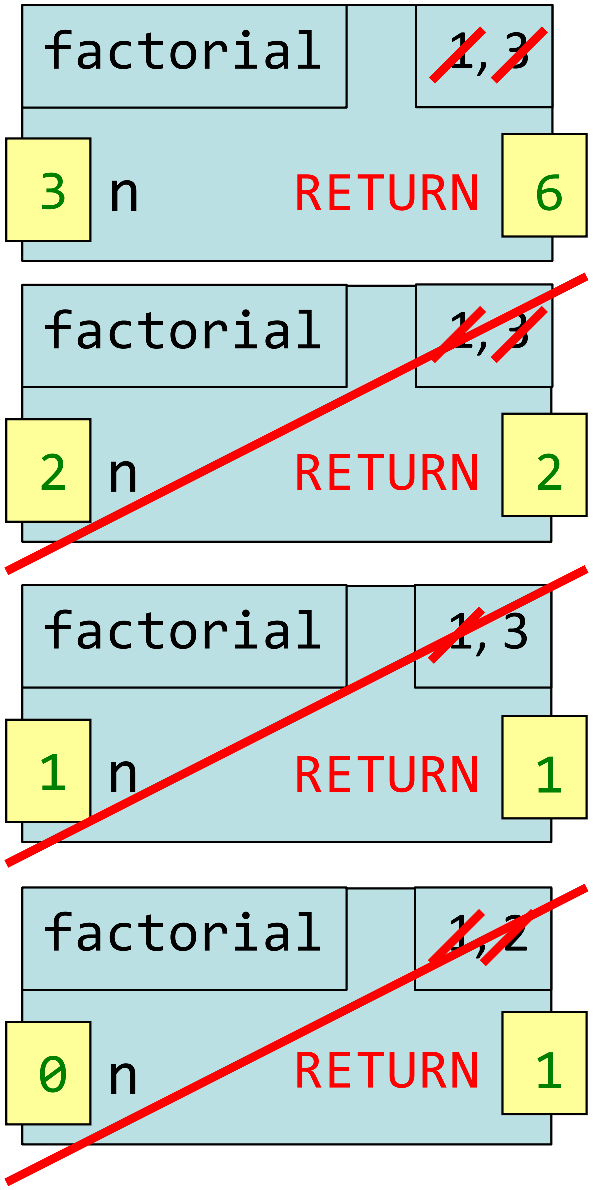
```
factorial(3)
```



Recursive Call Frames (n==3, RETURN 6)

```
  def factorial(n):
      """Returns: factorial of n.
      Precondition: n ≥ 0 an int"""
1     if n == 0:
2         return 1
3     return n*factorial(n-1)
```

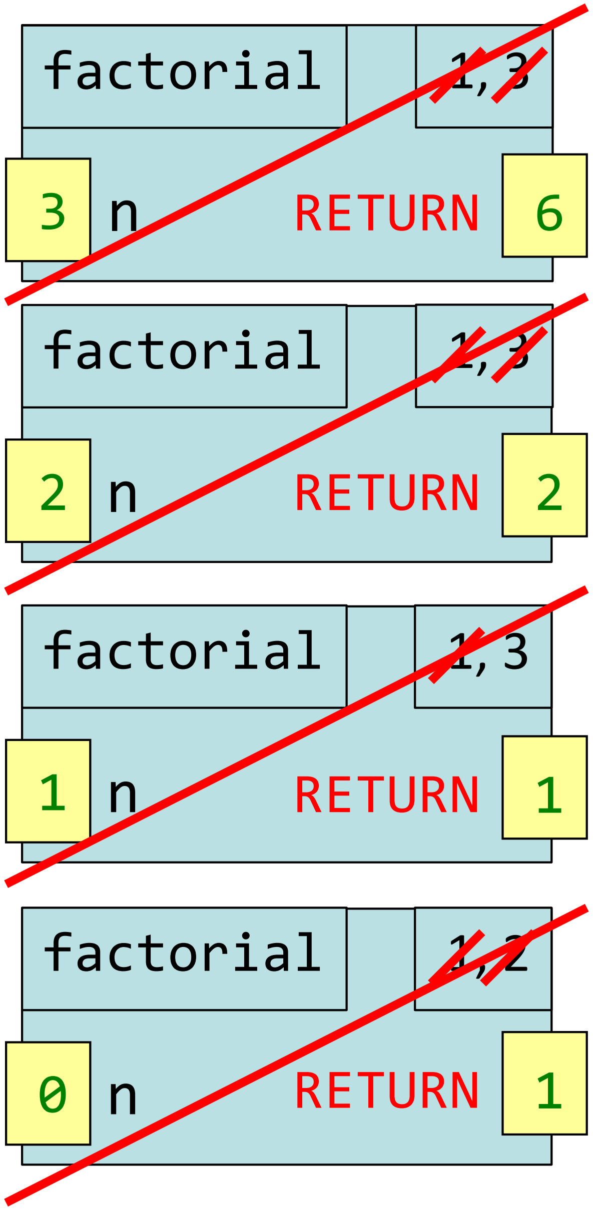
```
  factorial(3)
```



Recursive Call Frames (all calls complete!)

```
  def factorial(n):
      """Returns: factorial of n.
      Precondition: n ≥ 0 an int"""
1     if n == 0:
2         return 1
3     return n*factorial(n-1)
```

factorial(3)

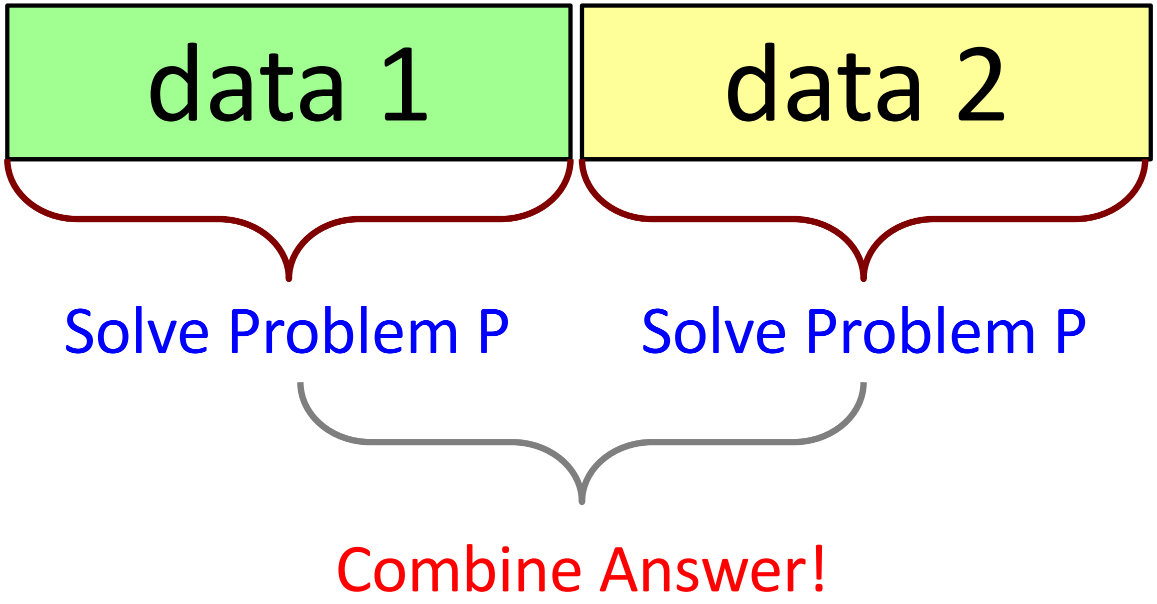


Divide and Conquer

Goal: Solve problem P on a piece of data

data

Idea: Split data into two parts and solve problem



data 1

data 2

Solve Problem P

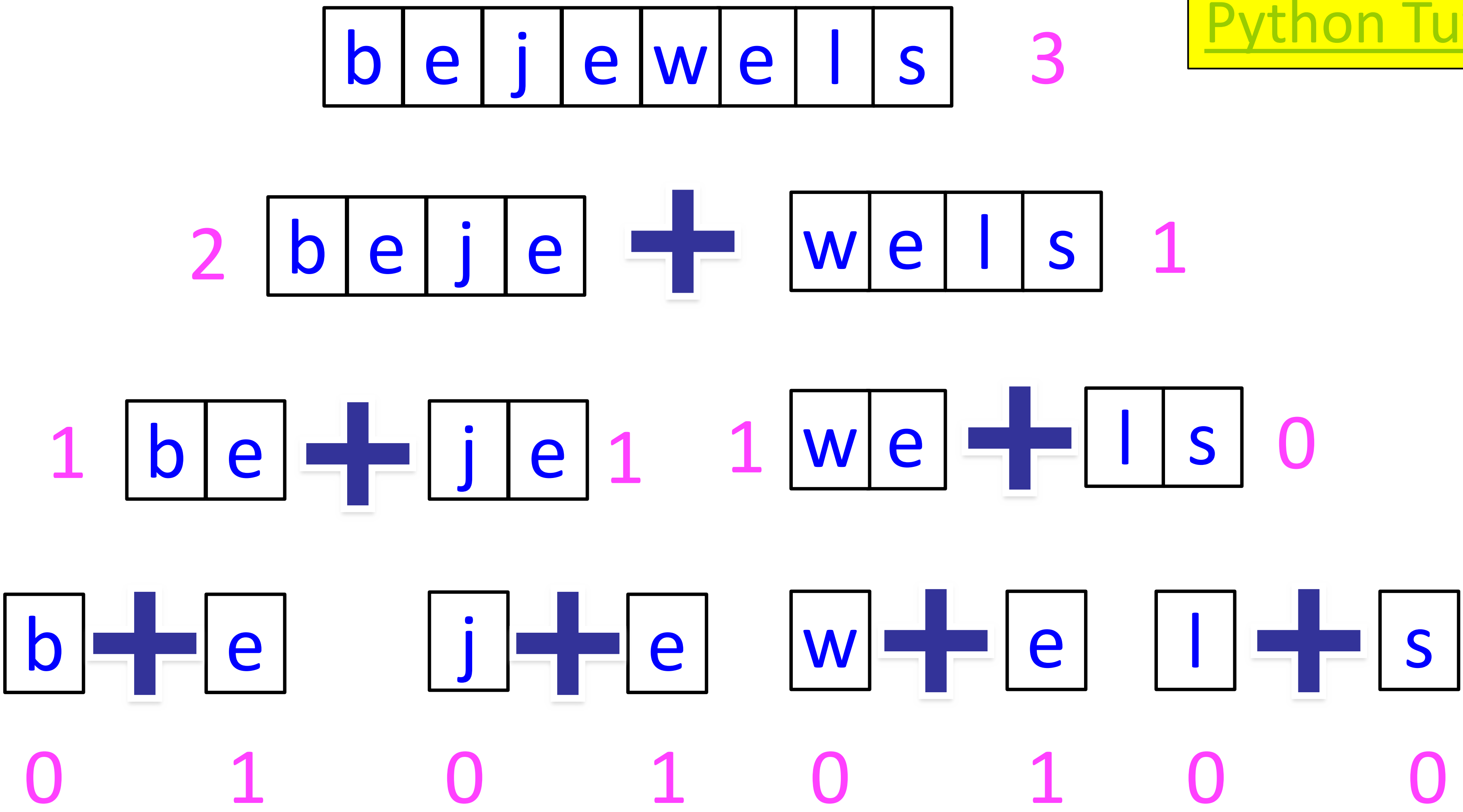
Solve Problem P

Combine Answer!

From Last Time: Divide and Conquer Example

Count the number of 'e's in a string:

Watch in the Python Tutor



b e j e w e l s 3

2 b e j e + w e l s 1

1 b e + j e 1   1 w e + l s 0

b + e   j + e   w + e   l + s

0 1 0 1 0 1 0 0

Example: Palindromes

- **Example:**

AMANAPLANACANALPANAMA

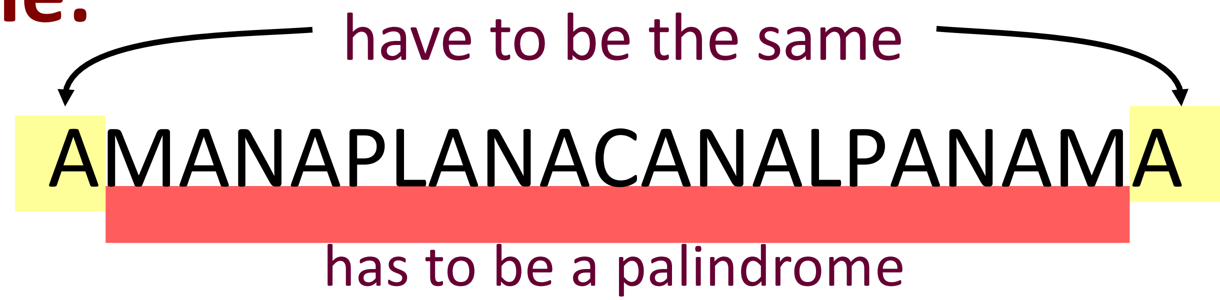
MOM

A

- Dictionary definition: “a word that reads (spells) the same backward as forward”
- Can we define recursively?

Example: Palindromes

- Strings with <= 1 character are palindromes
- String with ≥ 2 characters is a palindrome if:
  - its first and last characters are equal, and
  - the rest of the characters form a palindrome
- **Example:**



have to be the same

AMANAPLANACANALPANAMA

has to be a palindrome

- **Implement:**

```
def ispalindrome(s):
 """Returns: True if s is a palindrome"""
```

Example: Palindromes (1)

Strings with <= 1 character are palindromes

String with ≥ 2 characters is a palindrome if:

- its first and last characters are equal, and
- the rest of the characters form a palindrome

Recursive Definition!

What is the simple case? What is the complex case?

```
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
          return True

    endsAreSame = ___________________
    middleIsPali = ___________________
    return ___________________
```

Base case

Example: Palindromes (2)

Strings with <= 1 character are palindromes

String with ≥ 2 characters is a palindrome if:

- its first and last characters are equal, and
- the rest of the characters form a palindrome

Recursive Definition!

What is the simple case? What is the complex case?

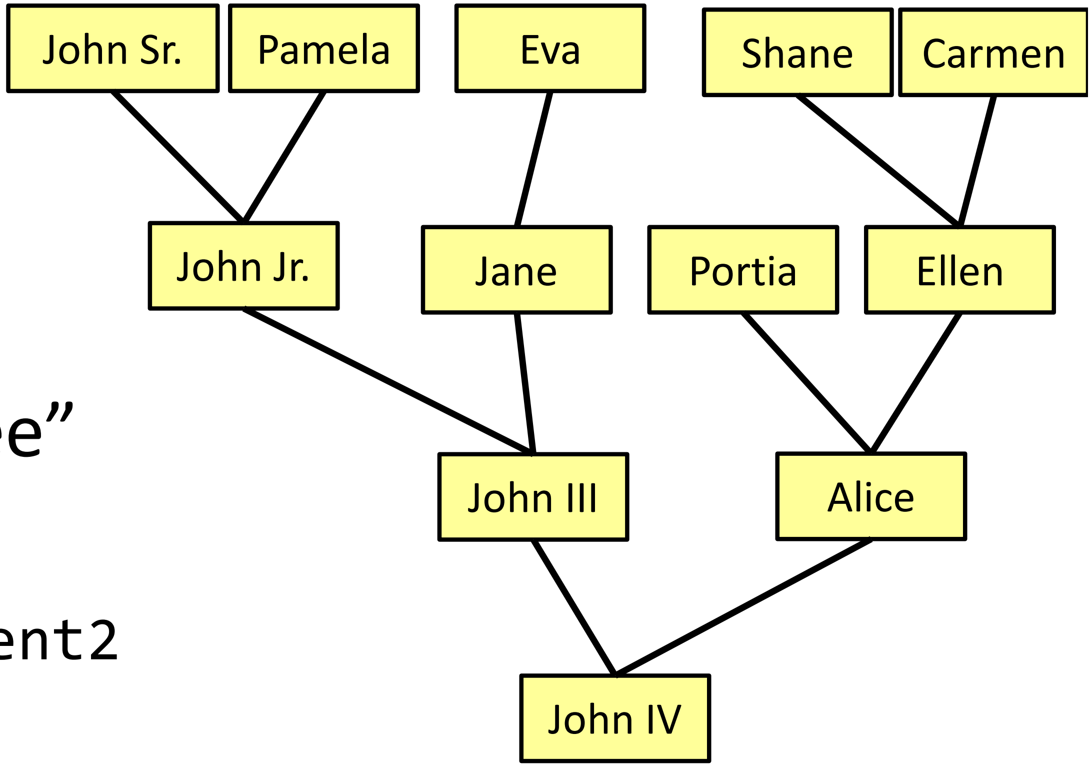
```
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
           return True

    endsAreSame = s[0] == s[-1]
    middleIsPali = ispalindrome(s[1:-1])
    return endsAreSame and middleIsPali
```

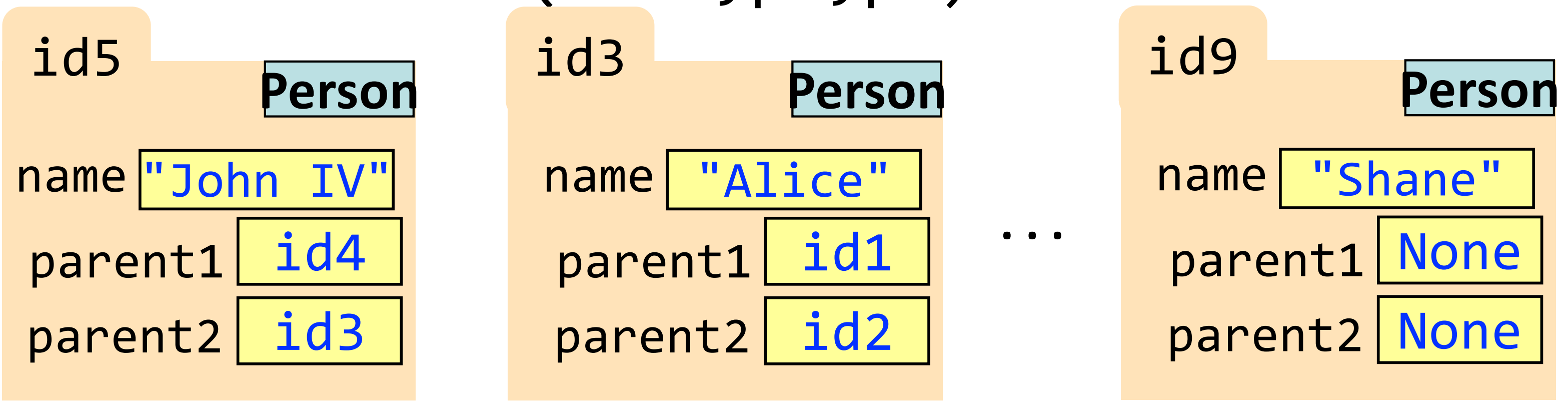
Base case

Recursive case

Recursion and Objects

- Class Person, 3 attributes
  - name: String
  - parent1: Person (or None)
  - parent2: Person (or None)
- Represents the "family tree"
  - Goes as far back as known
  - Attributes parent1 and parent2 are None if not known
- **Constructor**: Person(name,p1,p2)





id5 Person
name "John IV"
parent1 id4
parent2 id3

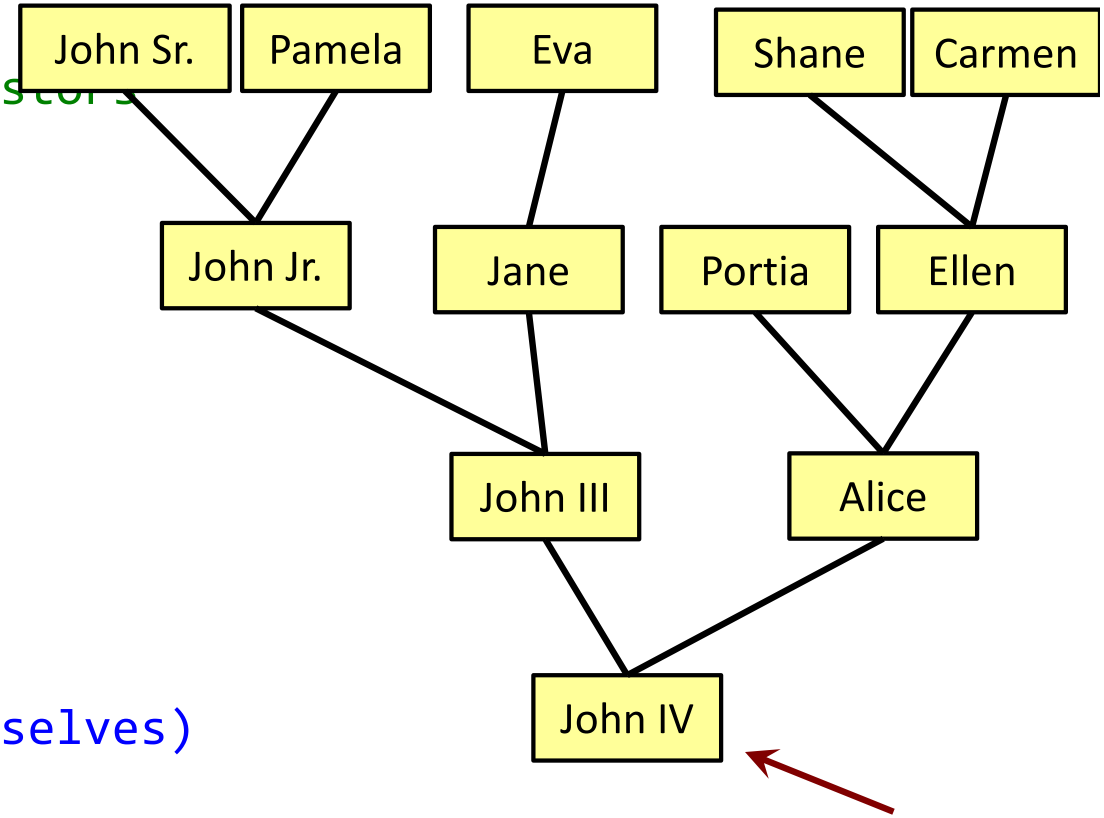
id3 Person
name "Alice"
parent1 id1
parent2 id2

…

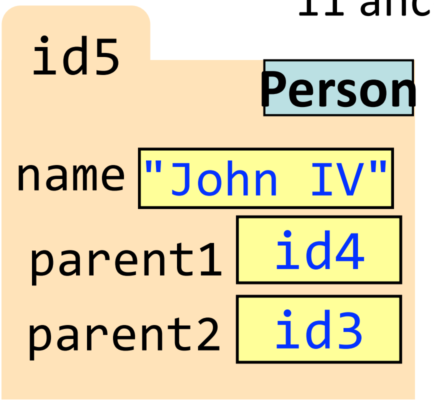
id9 Person
name "Shane"
parent1 None
parent2 None

Recursion and Objects: Setup

```
def count_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""
    # 1. Handle base case.
    # No parents (no ancestors)


    # 2. Break into two parts
    # Has parent1 or parent2
    # Count ancestors of each one
    # (plus parent1, parent2 themselves)




    # 3. Combine the result
```

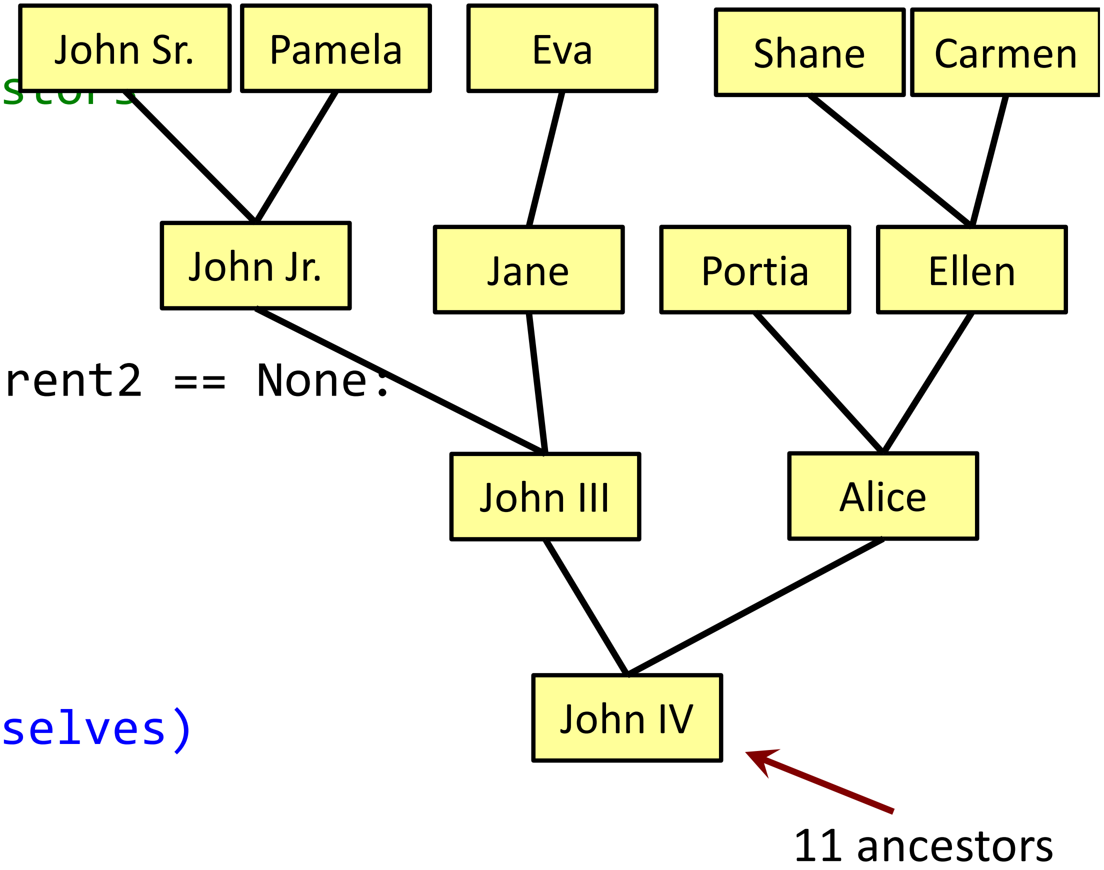


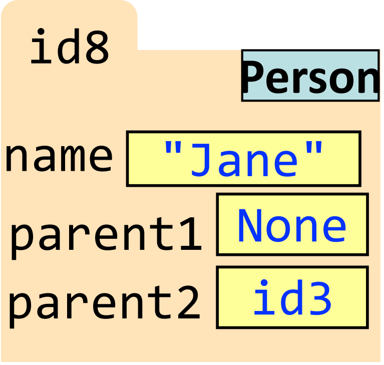
11 ancestors



Recursion and Objects: Implementation

```
def count_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""
    # 1. Handle base case.
    # No parents (no ancestors)
    if p.parent1 == None and p.parent2 == None:
        return 0
    # 2. Break into two parts
    # Has parent1 or parent2
    # Count ancestors of each one
    # (plus parent1, parent2 themselves)
    parent1s_fam = 0
    if p.parent1 != None:
        parent1s_fam = 1 + count_ancestors(p.parent1)
    parent2s_fam = 0
    if p.parent2 != None:
        parent2s_fam = 1 + count_ancestors(p.parent2)
    # 3. Combine the result
    return parent1s_fam + parent2s_fam
```





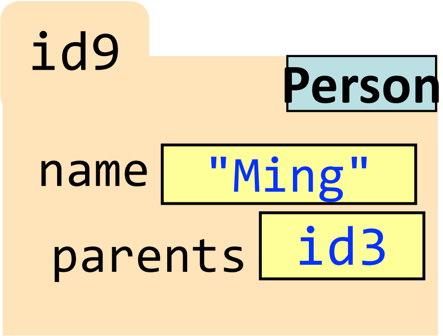
Recursion and Objects: Finishing Touches

```
def count_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"""
    # 1. Handle base case.
    # No parents (no ancestors)
    if p.parent1 == None and p.parent2 == None:
        return 0
    # 2. Break into two parts
    # Has parent1 or parent2
    # Count ancestors of each one
    # (plus parent1, parent2 themselves)
    parent1s_fam = 0
    if p.parent1 != None:
        parent1s_fam = 1 + count_ancestors(p.parent1)
    parent2s_fam = 0
    if p.parent2 != None:
        parent2s_fam = 1 + count_ancestors(p.parent2)
    # 3. Combine the result
    return parent1s_fam + parent2s_fam
```

We don’t actually need this.
It is handled by the conditionals in #2.

"It Takes a Village" Version: Lots of Parents

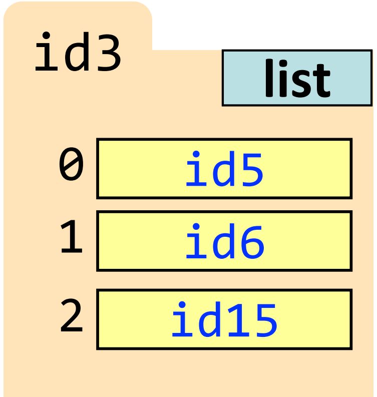
```
def count_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person with attribute parents, a list of parents """
    # 1. Handle base case. (We decided this wasn't necessary)

    # 2. Break into parts
    # For each parent, count ancestors
    # (plus parent, parent2 themselves)
    n_ancestors = 0
    for parent in p.parents:
        n_ancestors += (1 + count_ancestors(parent))

    # 3. Combine the result : FREE!
    return n_ancestors
```



id9 Person

name "Ming"

parents id3

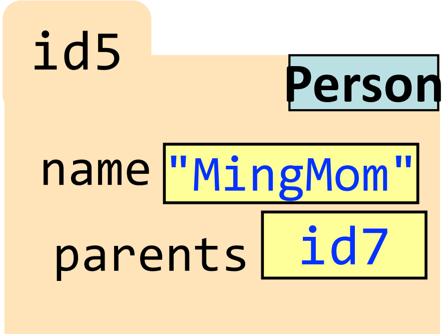


id3 list

0 id5

1 id6

2 id15



id5 Person

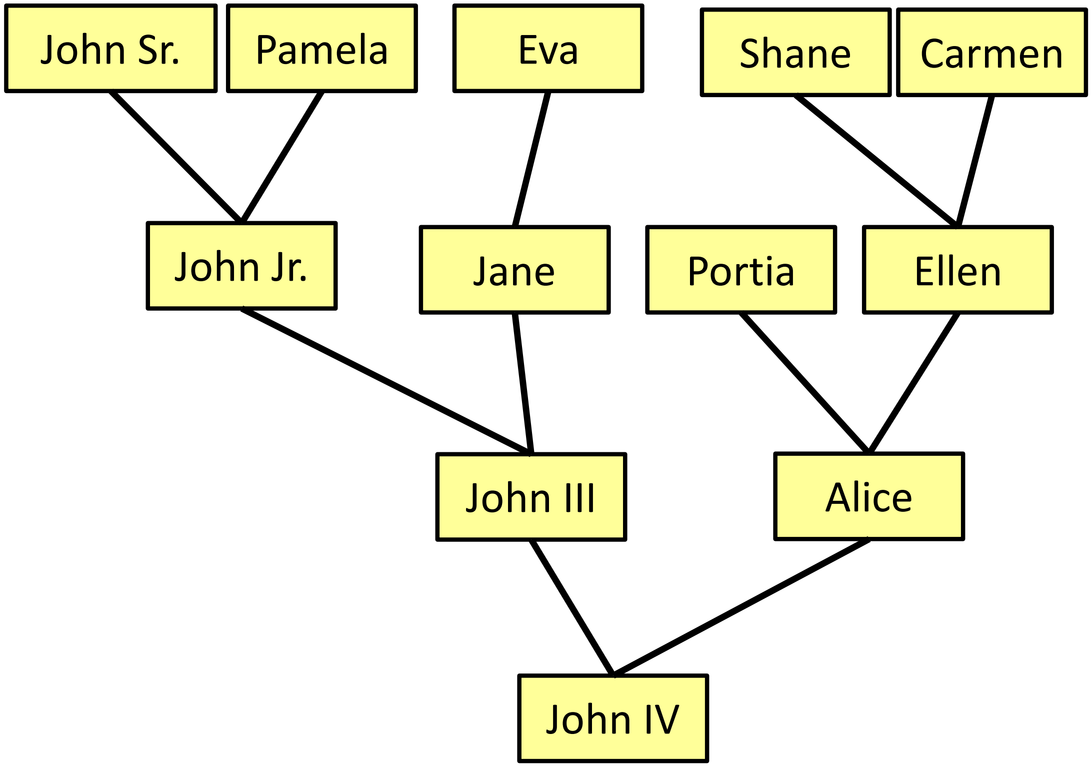
name "MingMom"

parents id7

```
# Notice when you have no parents, you return n_ancestors with the
# value 0. (the parent list is empty so you don't go in the loop)
```

Exercise: Find Ancestors

```
def list_ancestors(p):
    """Returns: list of all ancestors of p"""
    # 1. Handle base case.
    # 2. Break into parts.
    # 3. Combine answer.
```



Optional practice question. Try it after you complete this week's lab exercise.