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:
\[ n! = n \times (n-1) \times \ldots \times 2 \times 1 \]
\[ = n \times ((n-1) \times \ldots \times 2 \times 1) \]

Recursive definition:
\[ n! = n \times (n-1)! \quad \text{for } n > 0 \quad \text{Recursive case} \]
\[ 0! = 1 \quad \text{Base case} \]
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!
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 (n==2, execute line 1)

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

factorial(3)
```
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 (n==1, execute line 1)

```python
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 (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)
def factorial(n):
    
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int""

    if n == 0:
        return 1
    return n*factorial(n-1)

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

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

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

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

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

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

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

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

factorial(3)
**Goal**: Solve problem P on a piece of data

**Idea**: Split data into two parts and solve problem

- **data 1**
- **data 2**

Solve Problem P

Combine Answer!
Count the number of 'e's in a string:

```
bejewels
```

Watch in the Python Tutor
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 \( \leq 1 \) character are palindromes.

• String with \( \geq 2 \) characters is a palindrome if:
  ▪ its first and last characters are equal, and
  ▪ the rest of the characters form a palindrome.

• Example:

  ![Example](image)

• Implement:

  ```python
def is_palindrome(s):
    """Returns: True if s is a palindrome"""
```
Example: Palindromes (1)

Strings with \( \leq 1 \) character are palindromes

String with \( \geq 2 \) characters is a palindrome if:

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

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

```python
def is_palindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
        return True  # Base case

    ends_are_same = __________________________
    middle_is_pali = __________________________
    return __________________________
```
Example: Palindromes (2)

Strings with \( \leq 1 \) character are palindromes

String with \( \geq 2 \) characters is a palindrome if:

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

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
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)
Recursion and Objects: Setup

```python
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
```

![Family Tree Diagram]

id5
Person

- name: "John IV"
- parent1: id4
- parent2: id3

11 ancestors
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
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.
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

# 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)
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.