Lecture 16:
More Recursion!

CS 1110
Introduction to Computing Using Python

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

A3:

• Released Thu Mar 22
• Due Fri Mar 30

More 1-on-1’s on Friday maybe
Recursion

• **Recursive Function:**
  A function that calls itself (directly or indirectly)

• **Recursive Definition:**
  A definition that is defined in terms of itself
A Mathematical Example: Factorial

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

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

What happens if there is no base case?
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
    if n == 0:
        return 1
    return n * factorial(n - 1)

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

Call: factorial(3)
Recursion

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

Call: factorial(3)

A: factorial 1, 3
   n 3

B: factorial 1, 3, 1
   n x 2

D: factorial 1, 3, 1
   n x 2

C: ERASE FRAME
   factorial 1
   n 3
   factorial 1
   n 3
   factorial 1
   n 2
What happens next? (A)

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

Call: factorial(3)

A: CORRECT

B:

D:

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

Call: factorial(3)
Recursive Call Frames

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

**Call**: factorial(3)
Recursive Call Frames

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

Call: factorial(3)
```
Recursive Call Frames

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

Call: factorial(3)
```
Recursive Call Frames

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

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

Call: factorial(3)
def factorial(n):
    
    """Returns: factorial of n.
    Pre: n ≥ 0 an int""

    if n == 0:
        return 1

    return n * factorial(n - 1)

Call: factorial(3)
**Recursive Call Frames**

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

**Call:** factorial(3)

1. `factorial
2. n 3
3. factorial
   n 2
   factorial
   n 1
   factorial
   n 0
   RETURN

```

1, 3
1, 3
1, 3
1, 2

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

Call: factorial(3)
Recursive Call Frames

def factorial(n):
    '''Returns: factorial of n.
    Pre: n ≥ 0 an int'''

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

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

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

Call: factorial(3)
Recursive Call Frames

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

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

Call: factorial(3)
**Recall: Divide and Conquer**

**Goal:** Solve problem P on a piece of data

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

- **data**
  - **data 1**
    - Solve Problem P
  - **data 2**
    - Solve Problem P

**Combine Answer!**
Example: Reversing a String

```python
def reverse(s):
    """Returns: reverse of s"
    """Precondition: s a string"
    # 1. Handle base case
    # 2. Break into two parts
    # 3. Combine the result
```
Example: Reversing a String

```python
def reverse(s):
    
    # 1. Handle base case

    # 2. Break into two parts

    # 3. Combine the result
```

// Diagram showing the reversal process
def reverse(s):
    """Returns: reverse of s
    Precondition: s a string""
    # 1. Handle base case
    # 2. Break into two parts
    # 3. Combine the result
    return A: left + right B: right + left C: left D: right
def reverse(s):
    """Returns: reverse of s
    Precondition: s a string"
    # 1. Handle base case
    # 2. Break into two parts
    # 3. Combine the result
    return A: left + right

    CORRECT

    B: right + left
    C: left
    D: right
Example: Reversing a String

def reverse(s):
    """Returns: reverse of s
    Precondition: s a string""
    # 1. Handle base case

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

    # 3. Combine the result
    return right+left
What is the Base Case? (Q)

def reverse(s):
    """Returns: reverse of s
    Precondition: s a string""
    # 1. Handle base case
    A: if s == "":
        return s
    B: if len(s) <= 2:
        return s
    C: if len(s) <= 1:
        return s
    # 2. Break into two parts
    left = reverse(s[0])
    right = reverse(s[1:])
    # 3. Combine the result
    return right+left

D: Either A or C would work
E: A, B, and C would all work
def reverse(s):
    # 1. Handle base case
A: if s == "":
    return s
B: if len(s) <= 2:
    return s
C: if len(s) <= 1:
    return s

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

# 3. Combine the result
return right+left

CORRECT

D: Either A or C would work
E: A, B, and C would all work
Example: Reversing a String

```python
def reverse(s):
    """Returns: reverse of s
    Precondition: s a string""
    # 1. Handle base case
    if len(s) <= 1:
        return s

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

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

Base Case

Recursive Case
```python
def reverse(s):
    """Returns: reverse of s
    Precondition: s a string"""
    # 1. Handle base case
    if len(s) <= 1:
        return s

    # 2. Break into two parts
    half = len(s)//2
    left = reverse(s[:half])
    right = reverse(s[half:])

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

Does this work?

A: YES
B: NO
Alternate Implementation (A)

```python
def reverse(s):
    """Returns: reverse of s
    Precondition: s a string""
    # 1. Handle base case
    if len(s) <= 1:
        return s

    # 2. Break into two parts
    half = len(s) // 2
    left = reverse(s[:half])
    right = reverse(s[half:])

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

Does this work?

CORRECT

A: YES
B: NO
Alternate Implementation

Hello!

reverse(s[:half])

Hello!

reverse(s[half:])

reverse(s[half:])

Hello!

reverse(s[half:])

Hello!

reverse(s[half:])

Hello!

reverse(s[half:])

Hello!
Alternate Implementation

reverse(s[:half]) reverse(s[half:])

reverse(s[:half]) reverse(s[half:]) reverse(s[:half]) reverse(s[half:])

reverse(s[:half]) reverse(s[half:])

reverse(s[:half]) reverse(s[half:])

reverse(s[:half])

reverse(s[:half])

reverse(s[:half])
Example: Palindromes

- Example:

  AMANAPLANACANALPANAMA

- Can we define recursively?
Example: 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:

  has to be the same

  AMANAPLANACANALPANAMA

  has to be a palindrome

• Implement: `def ispalindrome(s):

  """Returns: True if s is a palindrome""""
Example: 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

```python
def ispalindrome(s):
    """Returns: True if s is a palindrome"""
    if len(s) < 2:
        return True
    ends = s[0] == s[-1]
    middle = ispalindrome(s[1:-1])
    return ends and middle
```
Recursion and Objects

- **Class Person**
  - Objects have 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)`
  - Or `Person(n)` if no parents known
def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"
    # 1. Handle base case.
    # No parent1 or parent2
    #(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

```python
def num_ancestors(p):
    """Returns: num of known ancestors
    Pre: p is a Person"
    # 1. Handle base case.
    if p.parent1 == None and p.parent2 == None:
        return 0

    # 2. Break into two parts
    parent1s = 0
    if p.parent1 != None:
        parent1s = 1 + num_ancestors(p.parent1)
    parent2s = 0
    if p.parent2 != None:
        parent2s = 1 + num_ancestors(p.parent2)

    # 3. Combine the result
    return parent1s + parent2s
```

11 ancestors
Recursion and Objects

```python
def num_ancestors(p):
    """Returns: num of known ancestors
Pre: p is a Person"""
    # 1. Handle base case.
    if p.parent1 == None and p.parent2 == None:
        return 0

    # 2. Break into two parts
    parent1s = 0
    if p.parent1 != None:
        parent1s = 1+num_ancestors(p.parent1s)
    parent2s = 0
    if p.parent2 != None:
        parent2s = 1+num_ancestors(p.parent2s)

    # 3. Combine the result
    return parent1s+parent2s
```

We don’t actually need this.
It is handled by the conditionals in #2.
def all_ancestors(p):
    """Returns: list of all ancestors of p"""

    # 1. Handle base case.
    # 2. Break into parts.
    # 3. Combine answer.