Lecture 18:
More Recursion!

CS 1110
Introduction to Computing Using Python

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

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

\[ 0! = 1 \quad \text{Base case} \]
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)
Recursion

```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)

Now what?
Each call is a new frame.
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)

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)

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

factorial(3)
**Recursive Call Frames**

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

`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)

factorial(3)
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
    else:
        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)

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

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

factorial(3)
Recursive Call Frames

```python
def factorial(n):
    """Returns: factorial of n.
    Pre: n \geq 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)
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)
```

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)
```

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

factorial(3)
[Start next video: ways to divide (and conquer)]
Divide and Conquer

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

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

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

Solve Problem P  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):
    """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 left + right
```

If this is how we break it up….

How do we combine it?
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 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
    left = reverse(s[0])
    right = reverse(s[1:])
    # 3. Combine the result
    return right+left
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
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
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
def reverse(s):
    if len(s) <= 1:
        return s
    half = len(s)//2
    left = reverse(s[:half])
    right = reverse(s[half:])
    return right+left

Execute the function call reverse('Hello!')

Result: ‘!olleh’
```python
def reverse(s):
    if len(s) <= 1:
        return s
    half = len(s) // 2
    left = reverse(s[:half])
    right = reverse(s[half:]):
    return right + left
```

Execute the function call `reverse('Hello!')`

Result: ‘!olleh’
Example: Palindromes

- Example:

  AMANAPLANACANALPANAMA

  MOM

- Dictionary definition: “a word that reads (spells) the same backward as forward”

- Can we define recursively?
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

• Example:
  
  ![AMANAPLANACANALPANAMA]

  have to be the same

  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
```
[Start next video: recursion and objects]
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)
```
def num_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
```
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
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

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.