Lecture 17: Recursion
(Re-introduction of this topic first discussed over 3 weeks ago!)
(Sections 5.8-5.10)

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

An example in mathematics: factorial
• Non-recursive definition:
  \[ n! = n \times (n-1) \times \ldots \times 2 \times 1 \]

• Recursive definition:
  \[ n! = n \times (n-1)! \]
  \[ 0! = 1 \]
Recursion

Recursive Function:

A function that calls itself

Two parts to every recursive function:

1. A simple case: can be solved easily
2. A complex case: can be made simpler (and simpler, and simpler… until it looks like the simple case)
Russian Dolls!
Think about opening a set of Russian dolls as a “problem.” Which is the simpler case,

the case where the doll has a seam and another doll inside of it, or

the case where the doll has no seam and no doll inside of it?
import russian

Russian Dolls!
import russian
d1 = russian.Doll("Dmitry", None)
d2 = russian.Doll("Catherine", d1)
def open_doll(d):
    """Input: a Russian Doll
    Opens the Russian Doll d """
    print("My name is "+ d.name)
    if d.hasSeam:
        # open inner doll
        open_doll2(d.innerDoll)
    else:
        print("That's it!")
def open_doll2(d):
    """Input: a Russian Doll
    Opens the Russian Doll d ""
    print("My name is "+ d.name)
    if d.hasSeam:
        # open inner doll
        open_doll3(d.innerDoll)
    else:
        print("That's it!")

What would this function look like?

<table>
<thead>
<tr>
<th>idx</th>
<th>Doll</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>hasSeam</td>
<td></td>
</tr>
<tr>
<td>innerDoll</td>
<td></td>
</tr>
</tbody>
</table>
def open_doll3(d):
    """Input: a Russian Doll
    Opens the Russian Doll d """
    print("My name is "+ d.name)
    if d.hasSeam:
        # open inner doll
        open_doll4(d.innerDoll)
    else:
        print("That's it!")

This function should look just like the others!
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!")
Play with the code

- Download modules `russian.py`, `playWithDolls.py`
- Read `playWithDolls.py`; then run it as a script.
- Modify last statement and run script again:
  - `open_doll(d3)`
- Modify last statement again and run script again:
  - `open_doll(d1)`
- Do you understand the result? Visit virtual office/consulting hours if you have any questions.
[Start next video: more examples]
Recursion: Examples

- Russian Dolls
- **Blast Off!**
- Factorial
- Count number of ‘e’s
- Deblank – removing spaces from a string
Blast Off!

- `blast_off(5)` # must be a non-negative int
- 5
- 4
- 3
- 2
- 1
- BLAST OFF!

- `blast_off(0)`
- BLAST OFF!
### Blast Off!

<table>
<thead>
<tr>
<th>blast_off(5)</th>
<th># must be a non-negative int</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>BLAST OFF!</strong></td>
<td></td>
</tr>
</tbody>
</table>

**What is the simple case that can be solved easily?**

- positive \( n > 1 \)
- \( n \) is 1
- \( n \) is 0

<table>
<thead>
<tr>
<th>blast_off(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BLAST OFF!</strong></td>
</tr>
</tbody>
</table>
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)
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?
Factorial as a Recursive Function

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

- n! = n (n-1)!
- 0! = 1

Base case(s)

Recursive case

What happens if there is no base case?
Recursion vs Iteration

• **Recursion** is *provably equivalent* to iteration
  - Iteration includes *for-loop* and *while-loop* (later)
  - Anything can do in one, can do in the other

• But some things are easier with recursion
  - And some things are easier with iteration

• Will **not** teach you when to choose recursion
  - That’s for upper level courses

• We just want you to **understand the technique**
[Start next video: divide & conquer]
Recursion is great for Divide and Conquer

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

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

- Solve Problem P
- Solve Problem P

Combine Answer!
Divide and Conquer Example

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

```
  p e n n e
```

Two 'e's

```
  p e
+  n n e
```

One 'e'

One 'e'
Divide and Conquer Example

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

Two 'e's

Zero 'e's

Two 'e's
Divide and Conquer

**Goal**: Solve really big problem $P$

**Idea**: Split into simpler problems, solve, combine

**3 Steps:**

1. Decide what to do for simple cases
2. Decide how to break up the task
3. Decide how to combine your work
Three Steps for Divide and Conquer

1. Decide what to do on “small” data
   - Some data cannot be broken up
   - Have to compute this answer directly

2. Decide how to break up your data
   - Both “halves” should be smaller than whole
   - Often no wrong way to do this (next lecture)

3. Decide how to combine your answers
   - Assume the smaller answers are correct
   - Combining them should give bigger answer
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
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
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

    """Short-cut""" for
    if s[0] == 'e':
        return 1
    else:
        return 0
Divide and Conquer Example

```python
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
```

<table>
<thead>
<tr>
<th></th>
<th>s[0]</th>
<th>s[1:]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p</td>
<td>e n</td>
</tr>
<tr>
<td>left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>right</td>
<td></td>
<td>n n e</td>
</tr>
</tbody>
</table>

0 2
Divide and Conquer Example

```python
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
```

```
s[0]  s[1:]
\[
p e n n e
\]
0 + 2
```
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
Exercise: Remove Blanks from a String

def deblank(s):
    """Returns: s but with its blanks removed"""

1. Decide what to do on “small” data
   - If it is the empty string, nothing to do
     ```python
     if s == ":
         return s
     ```
   - If it is a single character, delete it if a blank
     ```python
     if s == " ":  # There is a space here
         return ""  # Empty string
     else:
         return s
     ```
def deblank(s):
    """Returns: s but with its blanks removed"""
    left = deblank(s[0])  # A string with no blanks
    right = deblank(s[1:])  # A string with no blanks

3. Decide how to combine the answers

    return left + right  # String concatenation
Putting it All Together

def deblank(s):
    """Returns: s w/o blanks"""
    if s == "":
        return s
    elif len(s) == 1:
        return " " if s[0] == '' else s
    left = deblank(s[0])
    right = deblank(s[1:])
    return left+right

Handle small data

Break up the data

Combine answers
def deblank(s):
    """Returns: s w/o blanks"""
    if s == "":
        return s
    elif len(s) == 1:
        return " if s[0] == " " else s
    left = deblank(s[0])
    right = deblank(s[1:])
    return left+right
Following the Recursion

You really, really, really want to **visualize a call of deblank using Python Tutor**. Pay attention to the recursive calls (call frames opening up), the completion of a call (sending the result to the call frame “above”), and the resulting accumulation of the answer.
Post-lecture exercise

• Visualize a call of `deblank` using Python Tutor
• Code in file `deblank.py`
• Pay attention to
  ▪ the recursive calls (call frames opening up),
  ▪ the completion of a call (sending the result to the call frame “above”),
  ▪ and the resulting accumulation of the answer.
• Do this exercise before next lecture. *Really!*