Lecture 13:
Recursion
(Sections 5.8-5.10)

CS 1110
Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
Announcements (1/2)

• A3: not allowed to use use dict method update()
• Prelim 1 grades: read the grade centers email/see announcement
• Gauging interest on (Ed Discussions) in catch-up/subject-review sessions:
  ▪ https://edstem.org/us/courses/19140/discussion/1290339
Announcements (2/2)

Want more practice with for loops?

- posted codingbat to course homepage (4.F = under "help, advice"), many easy-to-hard problems

- for thing in list vs for in in range(len(...)):
  - [https://edstem.org/us/courses/19140/discussion/1289599](https://edstem.org/us/courses/19140/discussion/1289599)

- Extra optional exercises added to the lab 11 frontpage: loop_practice.py, loop_practice_test.py, cornellasserts.py
Recursion

• Not new python, but a new way of organizing thinking/algorithm
• Important in CS—CS majors will see it in action all 4 years
• Introduction only in CS1110, over 2 lectures
  1. Intro, examples, “divide & conquer”
  2. Visualization, different ways to “divide”, + objects
• Hard work on understanding call frames and the call stack will now pay off!
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:
  \[ n! = n \times (n-1)! \]
  \[ 0! = 1 \]

Details in pre-lecture videos
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

\[
\begin{align*}
\textbf{d1} & = \text{russian.Doll("Dmitry", None)} \\
\textbf{d2} & = \text{russian.Doll("Catherine", d1)}
\end{align*}
\]
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?
• Use Python Tutor to visualize (more next lecture)
Recursion: Examples

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

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

blast_off(0)
BLAST OFF!
Blast Off!

blast_off(5)  # non-negative int

What is the simple case that can be solved easily?

- positive n > 1
- n is 1
- n is 0

BLAST OFF!

blast_off(0)

BLAST OFF!
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:

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

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

• Recursive definition:

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

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

Details in pre-lecture videos
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

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**
Recursion is great for **Divide and Conquer**

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

- data
- data 1
- data 2

Solve Problem P  Solve Problem P
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

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

Combine Answer!
Divide and Conquer Example

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

\[
\text{bejewels} \quad 3
\]

\[
2 \quad \text{beje} + \quad \text{wels} \quad 1
\]

\[
1 \quad \text{be} + \quad \text{je} \quad 1 \quad 1 \quad \text{we} + \quad \text{els} \quad 0
\]

\[
\text{be} + \quad \text{je} \quad \text{we} + \quad \text{els} \quad \text{es}
\]

\[
0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0
\]
Divide and Conquer Example

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

```
j ewel  2
0  e + w e l  2
1  e + w e l  1
0  e + l  0
```

Will talk about how to break-up later
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
   - Combine them to give the aggregate answer
Divide and Conquer Example

def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data

    # 2. Break into two parts

    # 3. Combine the result
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>s[0]</th>
<th>s[1:]</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>enne</td>
</tr>
</tbody>
</table>

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
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

```python
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 ''
     else:
         return s
     ```
def deblank(s):
    """Returns: s but with its blanks removed"""

2. Decide how to break it up
    left = deblank(s[0])  # str w/o blanks
    right = deblank(s[1:]) # str w/o blanks

3. Decide how to combine the answers
    return left+right     # str concatenation
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
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
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!*