http://www.cs.cornell.edu/courses/cs1110/2021sp

Lecture 15: **Recursion**

(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]

Announcements

- Assignment 2 regrade request due Friday
- New topic today—recursion—takes time to learn
  - Post-lecture activities
  - Next lab to be released a little earlier than usual so that you can think about it and ask questions during lab. Not earlier due date—just more time to think

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:

$$n! = n \times \underbrace{n\text{-}1 \times \ldots \times 2 \times 1}_{(n\text{-}1)!}$$

- Recursive definition:

  n! = n (n-1)!

  0! = 1

Details in pre-lecture videos

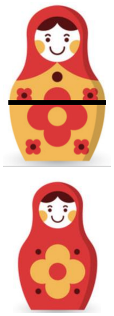
Recursion

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)

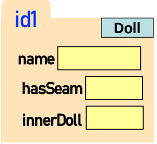


Think about opening a set of Russian dolls as a "problem." Which is the simpler case,

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?

Russian Dolls!



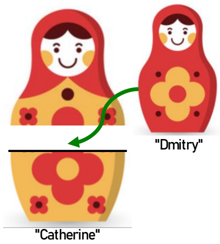
id1 Doll

name

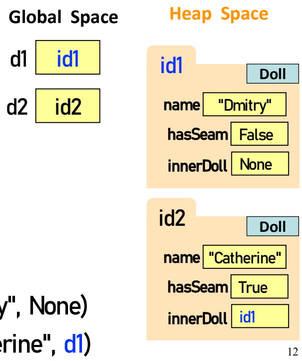
hasSeam

innerDoll

```
import russian
```



Russian Dolls!



Global Space

d1 id1

d2 id2

Heap Space

id1 Doll

name "Dmitry"

hasSeam False

innerDoll None

id2 Doll

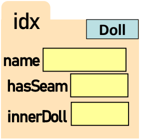
name "Catherine"

hasSeam True

innerDoll id1

```
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:
        inner = d.innerDoll
        open_doll(inner)
    else:
        print("That's it!")
```



idx Doll

name

hasSeam

innerDoll

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

What is the simple case that can be solved easily?

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

```
blast_off(0)
BLAST OFF!
```

Blast Off!

```
def blast_off(n):
  """Input: a non-negative int
   Counts down from n to Blast-Off!
   """
```

A Mathematical Example: Factorial

- Non-recursive definition:

  $n! = n \times n\text{-}1 \times \ldots \times 2 \times 1$

  $\quad = n\,(n\text{-}1 \times \ldots \times 2 \times 1)$

- Recursive definition:

  $n! = n\,(n\text{-}1)!$ for $n > 0$ **Recursive case**

  $0! = 1$ **Base case**

Details in pre-lecture videos

Factorial as a Recursive Function

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

    return n*factorial(n-1)   Recursive case
```

- $n! = n\,(n\text{-}1)!$
- $0! = 1$

What happens if there is no base case?

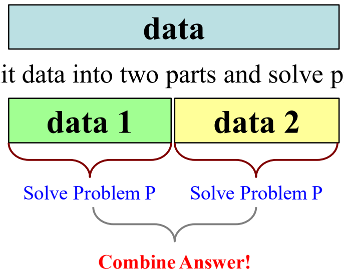
Recursion vs Iteration

- **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

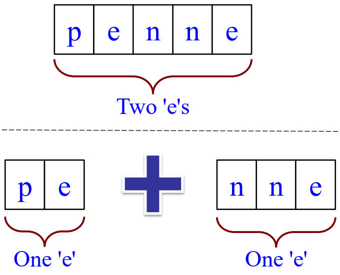
Goal: Solve problem P on a piece of data

Idea: Split data into two parts and solve problem



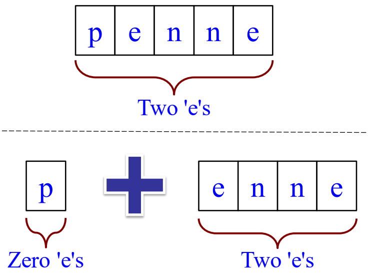
Divide and Conquer Example

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



Divide and Conquer Example

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



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
```

Divide and Conquer Example

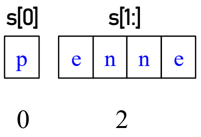
```
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

    # 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
```

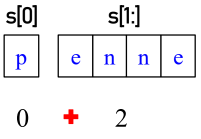
Divide and Conquer Example

```
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
```



Divide and Conquer Example

```
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
```



Divide and Conquer Example

```
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
```

Base Case

Recursive Case

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

```
if s == '':
    return s
```

   - If it is a **single character**, delete it if a blank

```
if s == ' ':    # There is a space here
    return ''   # Empty string
else:
    return s
```

Exercise: Remove Blanks from a String

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

2. Decide how to break it up

```
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

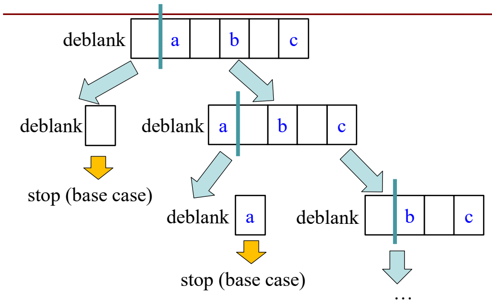
Putting it All Together

```
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
```

Base Case

Recursive Case

Following the Recursion



deblank | a | b | c

deblank | deblank a | b | c

stop (base case)

deblank a | deblank | b | c

stop (base case)

…

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!*