Announcements

• Prelim 1 feedback expected by Sunday
• Read § 5.8-5.10 (if you haven’t done so already)
Thinking about upcoming changes

• We know everyone is stressed out 😔 As situation and planning evolve we’ll keep you posted

• **Lecture**: recording is available. If you want to avoid lecture room even before break, it’s ok. You can view recording instead.

• **Labs**: exercises online. We’re trying to work out ways to provide interactive help somehow. Will need combination of technologies and platforms

• **Office/consulting hours**: ditto

• How will future **exams** work? This is difficult to deal with. In discussion inside and outside CS to come up with solution.

• Please use **Piazza**! Good way to get answers to clarification questions.
Recursion

Recursive Function:
A function that calls itself

(see also Recursive Function)

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

A: The case where the doll has a seam and another doll inside of it.
B: The case where the doll has no seam and no doll inside of it.
C: A & B are both simple
D: I do not know
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!")
Examples

- Russian Dolls
- Blast Off!
- Factorial
- Deblank
Blast Off!

blast_off(5) # must be a non-negative int
5
4
3
2
1
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:
  \[ n! = n \times n-1 \times \ldots \times 2 \times 1 \]
  \[ = n (n-1 \times \ldots \times 2 \times 1) \]

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

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)

• 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**
    - Solve Problem P
  - **data 2**
    - 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

Solve Problem P  Solve Problem P

Combine Answer!
**Divide and Conquer Example**

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

```
peenne
```

- Two 'e's

- One 'e'

- One 'e'
Divide and Conquer Example

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

```
penne
```

Two 'e's

```
p  
```
Zero 'e's

```
enne
```
Two 'e's
We're counting the number of 'e's in a string:

- 'pennenee'

We see:

- 0 'e's in 'p'
- 2 'e's in 'ennenene'

So, there are 2 'e's in total.

We'll 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
   - 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

“Short-cut” for
if s[0] == 'e':
    return 1
else:
    return 0
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
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

```python
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 answer
    return left + right    # String 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

- 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
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.
Breaking it up (1)

debblank

a b c

debblank

a b c
Breaking it up (2)

deblank

\[
\begin{array}{ccc}
  & a & b & c \\
\end{array}
\]

deblank

\[
\begin{array}{ccc}
  & a & b & c \\
\end{array}
\]

deblank

\[
\begin{array}{cc}
  a & \\
  & b & c \\
\end{array}
\]
Breaking it up (3)

deblank

```plaintext
  a  b  c
```

deblank

```plaintext
  a  b  c
```

deblank

```plaintext
  a
```

deblank

```plaintext
  b  c
```

deblank

```plaintext
  b  c
```
Breaking it up (4)

deblank

\[
\begin{array}{ccc}
  a & b & c \\
\end{array}
\]
Breaking it up (5)
Breaking it up (6)

deblank

a b c

deblank

a b c

a

deblank

b c

b

deblank

b c

b c
deblank

c

deblank

c
Combining Left+Right (1)

deblank

\[
\begin{array}{ccc}
a & b & c \\
\end{array}
\]

deblank

\[
\begin{array}{ccc}
a & b & c \\
\end{array}
\]

\[
\begin{array}{ccc}
a & b & c \\
\end{array}
\]

deblank

\[
\begin{array}{cc}
b & c \\
\end{array}
\]

\[
\begin{array}{cc}
b & c \\
\end{array}
\]

deblank

\[
\begin{array}{c}
c \\
\end{array}
\]

deblank

\[
\begin{array}{c}
c \\
\end{array}
\]

\[
\begin{array}{c}
c \\
\end{array}
\]
Combining Left+Right (2)

deblank

\[
\begin{array}{ccc}
a & b & c \\
\end{array}
\]

deblank

\[
\begin{array}{ccc}
a & b & c \\
\end{array}
\]

deblank

\[
\begin{array}{cc}
b & c \\
\end{array}
\]

deblank

\[
\begin{array}{c}
c \\
\end{array}
\]

\[
\begin{array}{c}
\text{c} \\
\end{array}
\]

\[
\begin{array}{c}
\text{c} \\
\end{array}
\]

\[
\begin{array}{c}
\text{c} \\
\end{array}
\]
Combining Left+Right (3)

deblank

\[ \begin{array}{ccc}
  & a & b & c \\
  \hline
  & a & b & c \\
  a & b & c \\
  b & c \\
  \times & c \\
  c & \\
\end{array} \]

\[ \begin{array}{cc}
  & b & c \\
  \hline
  & b & c \\
  c \\
  c \\
\end{array} \]
Combining Left+Right (4)

- deblank: a b c
- a: b c
- b: c
- c: c

Sequence:
1. a b c -> b c
2. b c -> b c
3. b c -> c
4. c: c
Combining Left+Right (5)

debblank

\[\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}\]

\[\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}\]

\[\begin{array}{ccc}
\text{b} & \text{c} \\
\end{array}\]

\[\begin{array}{ccc}
\text{b} & \text{c} \\
\end{array}\]

\[\begin{array}{ccc}
\text{c} \\
\end{array}\]

\[\begin{array}{ccc}
\text{c} \\
\end{array}\]
Combining Left+Right (6)
Combining Left+Right (7)

deblank

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{a} & \text{b} & \text{c} \\
\text{a} & \text{b} & \text{c} \\
\text{a} & \text{b} & \text{c} \\
\text{a} & \text{b} & \text{c} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{a} & \text{b} & \text{c} \\
\text{b} & \text{c} \\
\text{c} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{b} & \text{c} \\
\text{c} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{b} & \text{c} \\
\text{c} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{b} & \text{c} \\
\text{c} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{c} \\
\text{c} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{c} \\
\text{c} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\text{c} \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
\end{array}
\]