This is the last CS 5 lecture you’ll ever "need"!*  

Hw #2 due this Monday, 2/4, at 11:59 pm

**Tutoring hours: LOTS!**

*HMC’s legal counsel requires us to include these footnotes...*

饌 On Warner Brothers' insistence, we affirm that this 'C' does not stand for 'Chamber' and 'S' does not stand for 'Secrets.'

* Caution: do not take this statement too literally or it is possible find yourself in **twice** as many CS 5 lectures as you need!
Recursion example: \( \text{numis}(s) \)

Total number of \( i \)'s in \\
'\text{i<3five}'

\( \text{is} \)

Number of \( i \)'s in \\
'\text{i}'

+ \\
Number of \( i \)'s in \\
'\text{<3five}'
Recursion example: \( \text{numis}(s) \)

Total # of i's in 'alien' is

\[
\text{# of i's in 'a'} + \underbrace{\text{# of i's in 'lien'}}_{\text{default value}}
\]
Recursion example: \( numis(s) \)

```
\[
\text{total \# of i's in 'aliiien'} \\
\text{is} \\
\text{# of i's in 'a'} + \\
\text{# of i's in 'liiiien'}
\]
```
Recursion example: \( \text{numis}(s) \)

**Analysis...**

Total # of i's in 'aliiien' is

\[ \begin{align*}
\text{# of i's in 'a'} & + \\
\text{# of i's in 'liiien'} &
\end{align*} \]

... via self-similarity!
CS 5 ... Today!

Recursion

As close as CS gets to magic

a.k.a., CS's version of mathematical induction

Hw #1 due this Monday, 2/4, at 11:59 pm

This is the last CS 5 lecture you’ll ever "need"!

HMC's legal counsel requires us to include these footnotes...

On Warner Brothers' insistence, we affirm that this 'C' does not stand for 'Chamber' and 'S' does not stand for 'Secrets.'

* Caution: do not take this statement too literally or it is possible find yourself in twice as many CS 5 lectures as you need!
Picobot, Aargh!

AwkwardSelfie(Day) ...
if you attended lab and submit pr1+pr2: you get full credit for hw1pr1 and hw1pr2

else:
you should complete the two lab problems, pr1 + pr2

either way: submit pr1 + pr2
complete and submit hw2pr3

Extra Credit: Pig Latin / CodingBat

DNA transcription
This week's reading *on data*...

The End of Theory: The Data Deluge Makes the Scientific Method Obsolete

By Chris Anderson 06.23.08

The Petabyte Age:

Sensors everywhere. Infiniite storage. Clouds of processors. Our ability to capture, warehouse, and understand massive amounts of data is changing science, medicine, business, and technology. As our collection of facts and figures grows, so will the opportunity to find answers to fundamental questions. Because in the era of big data, more isn't just more. More is different.

"All models are wrong, but some are useful."

So proclaimed statistician George Box 30 years ago, and he was right. But what choice did we have? Only models, from cosmological equations to theories of human behavior, seemed to be able to consistently, if imperfectly, explain the world around us. Until now. Today companies like Google, which have grown up in an era of massively abundant data, don't have to settle for wrong models. Indeed, they don't have to settle for models at all.
Computation's Dual Identity

But what does the stuff on this side look like?
Computation's Dual Identity

accessed through *functions*…

It's no coincidence this starts with *fun*!
Functioning across disciplines

**procedure**

```python
def g(x):
    return x**100
```

**structure**

\[ g(x) = x^{100} \]

**CS's googolizer**

defined by *what it does*

+ what follows *behaviorally*

**Math's googolizer**

defined by *what it is*

+ what follows *logically*
Giving names to data helps f'ns

def flipside(s):
    """ flipside(s): swaps s's sides!
    input s: a string
    """
    x = len(s)//2
    return s[x:] + s[:x]

This idea is the key to your happiness!
Giving names to data **helps f'ns**

```python
def flipside(s):
    """ flipside(s): swaps s's sides!
    input s: a string
    """
    x = len(s) // 2
    return s[x:] + s[:x]
```

This idea is the key to **your** happiness!

'homework'

'work'

'home'

follow the data...
Use variables!

I'm happy about this, too!

OK: we humans work better with named variables.

But -- why would even computers "prefer" the top version, too?

def flipside(s):
    x = len(s)//2
    return s[x:] + s[:x]

Aargh!

def flipside(s):
    return s[len(s)//2:] + s[:len(s)//2]
Test!

```python
def flipside(s):
    """ flipside(s): swaps s's sides! 
    input s: a string 
    """
    x = len(s)/2
    return s[x:] + s[:x]
```

# Tests!
# assert flipside('homework') == 'workhome'
assert flipside('poptart') == 'tartpop'

```python
print(" petscar ~", flipside('carpets'))
print("    cs5! ~", flipside('5!cs'))
```
def convertFromSeconds(s):
    # total seconds
    """ convertFromSeconds(s): Converts an integer # of seconds into a list of [days, hours, minutes, seconds]
    input s: an int
    """

days = s // (24*60*60)  # total days
s = s % (24*60*60)     # remainder s
hours = s // (60*60)   # total hours
s = s % (60*60)        # remainder s
minutes = s // 60      # total minutes
s = s % 60             # remainder s
return [days, hours, minutes, s]
def convertFromSeconds(s):
    """ convertFromSeconds(s): Converts an integer # of seconds into a list of [days, hours, minutes, seconds]
    input s: an int
    """
    days = s // (24*60*60)  # total days
    s = s % (24*60*60)  # remainder s
    hours = s // (60*60)  # total hours
    s = s % (60*60)  # remainder s
    minutes = s // 60  # total minutes
    s = s % 60  # remainder s
    return [days, hours, minutes, s]
def dbl(x):
    """dbl x? """
    return 2*x

ans = dbl(20)

def dblPR(x):
    """dbl x? """
    print(2*x)

ans = dblPR(20)

What's the difference ?!
```
def dbl(x):
    """ dbls x? """
    return 2*x

def dblPR(x):
    """ dbls x? """
    print(2*x)

ans = dbl(20) + 2
    this is a value for further use!
    yes!

ans = dblPR(20) + 2
    this turns lightbulbs on!
    ouch!
```

**print** changes pixels on the screen...

**return** yields the function call's *value* ...

... which the shell then prints!
return > print

how software *passes information* from function to function...

changes the pixels (little *lightbulbs*) on your screen
how software *passes* information from function to function...

changes the pixels (little *lightbulbs*) on your screen
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

What is \text{demo}(15) \text{ here?}

What is \text{f}(2) \text{ here?}
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

"the stack"

they stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

15

call: demo(15)
stack frame
local variables:
x = 15
y = 5
z = ?????

they stack.
```python
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

How functions work...

```
15
```

they stack.
```python
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

How functions work...

```
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

```
call: demo(15)  
local variables:
    x = 15
    y = 5
    z = ????

returns 22

call: g(5)  
local variables:
    x = 5
    result = 22
returns 22
```

they stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

call: demo(15)  
local variables:  
x = 15
y = 5
z = 22

they stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
def demo(x):
y = x/3
z = g(y)
return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

they stack.

afterwards, the stack is empty..., but ready if another function is called

output
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

what's f(2)?
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

stack frame
x = 0
returns 12

How functions work...
How functions work...

```python
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

stack frame

```
call: f(2)
local variables: x = 2
need f(1)
```

stack frame

```
call: f(1)
local variables: x = 1
need f(0)
```

stack frame

```
call: f(0)
local variables: x = 0
returns 12
```
How functions work...

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

Call: `f(2)`
Local variables:
- `x = 2`
- Need `f(1)`

Call: `f(1)`
Local variables:
- `x = 1`
- `f(0) = 12`
- `result =`

How do we compute the result?
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

1

"the stack"

stack frame
call: f(2)
local variables:
  x = 2
  need f(1)

stack frame
call: f(1)
local variables:
  x = 1
  f(0) = 12
  result = 22

Where does that result go?
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

result = f(1)
How functions work...

def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

What's this return value?
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

2

call: f(2)
for x = 2
result = 42

the result then gets returned...

How functions work...

"the stack"
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

functions stack.

How functions work...

again, the stack is empty, but ready if another function is called...

"the stack"
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

Functions are software's **cells** ...

... each one is a **self-contained computational unit**!

again, the stack is empty, but ready if another function is called...
How functions work...

Pass these eastward!

... each one is a self-contained computational unit!

functions stack.
sequential

iteration

self-similar

recursion

problem-solving paradigms
Thinking *sequentially*

\[ \text{fac}(5) = 5 \times 4 \times 3 \times 2 \times 1 = 120 \]

\[ \text{fac}(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1 \]
Thinking *sequentially*

\[ \text{factorial} \]

\[ 5! = 120 \]

\[ \text{cs} \quad \text{fac}(5) = 5 \times 4 \times 3 \times 2 \times 1 \]

\[ \text{fac}(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1 \]
Thinking *recursively*

**factorial**

\[ \text{math} \quad 5! = 120 \]

\[ \text{cs} \quad \text{fac}(5) = 5 \times 4 \times 3 \times 2 \times 1 \]

\[ \text{cs} \quad \text{fac}(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1 \]

\[ \text{cs} \quad \text{fac}(N) = \]

Can we express \text{fac} with a smaller version of itself?
Thinking recursively...

**Recursion ~ self-similarity**

\[
\text{fac}(5) = 5 \times 4 \times 3 \times 2 \times 1
\]

\[
\text{fac}(5) = 5 \times \text{fac}(4)
\]

**can we express \text{fac} w/ a smaller version of itself?**

\[
\text{fac}(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1
\]

\[
\text{fac}(N) = N \times \text{fac}(N-1)
\]

We're done!?
Warning: this is legal!

```python
def fac(N):
    return N * fac(N-1)
```

I wonder how this code will STACK up!? 

```python
def facBad(N):
    return N * facBad(N-1)
```
unionAll resulting in StackOverflow

I've made some progress with my own question (how to load a dataframe from a python requests stream that is downloading a csv file?) on StackOverflow, but I'm receiving a StackOverflow error:

```python
import requests
import numpy as np
import pandas as pd
import sys
```
Recursion

the dizzying dangers of having no base case!

This "works" ~ but doesn't work!

def fac(N):
    return fac(N)
Defining a factorial function recursively:

\[ f(n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n-1) \times n & \text{if } n > 0 \end{cases} \]

Recursion can be the dizzying dangers of having no base case!

This "works" — but doesn't work!
Did you mean: recursion

Recursion - Wikipedia, the free encyclopedia
A visual form of recursion known as the Droste effect. The woman in this image is holding an object which contains a smaller image of her holding the same...

Recursion (computer science) - Wikipedia, the free encyclopedia
Recursion in computer science is a way of thinking about and solving problems. In fact, recursion is one of the central ideas of computer science...

Recursion -- from Wolfram MathWorld
A recursive process is one in which objects are defined in terms of other objects of the same type. Using some sort of recurrence relation, the entire class ...

recursion
Definition of recursion, possibly with links to more information and implementations.

Mastering recursive programming
calls to `facBad` will "never" stop: there's no BASE CASE

Make *sure* you have a base case

a.k.a. "escape hatch"
def fac(N):
    if N == 0:
        return 1
    else:
        return N * fac(N-1)

Thinking recursively...

Base case

Recursive case (too short?)
def fac(N):
    if N == 0:
        return 1
    else:
        return N * fac(N-1)

Thinking recursively...

Base case

Recursive case (too short?)

How can this multiply N by something that hasn't happened yet!??!
def fac(N):
    if N <= 1:
        return 1
    else:
        rest = fac(N-1)
        return N*rest

Conceptual

Acting recursively

Actual

this recursion happens first!

hooray for variables!
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)

Behind the curtain: how recursion works...

fac(5)
    5 * fac(4)
        4 * fac(3)
            3 * fac(2)
                2 * fac(1)
                    1.0
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
Behind the curtain: *how recursion works*...

```python
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

- **stack frame** with \(N = 5\)
- **stack frame** with \(N = 4\)
- **stack frame** with \(N = 3\)
- **stack frame** with \(N = 2\)

\[
\text{fac}(5) \rightarrow 5 \times \text{fac}(4) \\
5 \times \text{fac}(4) \rightarrow 4 \times \text{fac}(3) \\
4 \times \text{fac}(3) \rightarrow 3 \times \text{fac}(2) \\
3 \times \text{fac}(2) \rightarrow 2 \times 1.0
\]
Behind the curtain: *how recursion works...*

\[
\begin{align*}
\text{fac}(5) & \quad \Rightarrow \quad 5 \times \text{fac}(4) \\
5 \times \text{fac}(4) & \quad \Rightarrow \quad 4 \times \text{fac}(3) \\
4 \times \text{fac}(3) & \quad \Rightarrow \quad 3 \times 2.0
\end{align*}
\]

**stack frame with** N = 5

**stack frame with** N = 4

**stack frame with** N = 3

```python
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```
Behind the curtain: 

*how recursion works...*

def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)

fac(5)  

5 * fac(4)  

stack frame with N = 5  

4 * 6.0  

stack frame with N = 4
Behind the curtain: how recursion works...

```python
def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)
```

`fac(5)`

5 * 24.0

**stack frame** with N = 5
Behind the curtain: 
how recursion works...

\[ \text{fac}(5) \]

120.0

complete!

def fac(N):
    if N <= 1:
        return 1.0
    else:
        return N * fac(N-1)

But is recursion for real?!
Recursion's *conceptual* challenge?

You need to see **BOTH** the *self-similar pieces* AND the *whole thing* simultaneously!

... because it's completely *self-sufficient*!
Recursion

Base Case

Self-similar design

problem-solving paradigm
Recursion

Base Case

Self-similar design

Next: recursive-function DESIGN
The value of $5\times4\times3\times2\times1$ is $\text{fac}(5)$. The value of $4\times2\times3\times1$ is $\text{fac}(4)$. Base case: $\text{fac}(0)$ should return 1.
def fac(x):
    """ factorial! Recursively! """
    if x == 0:
        return 1
    else:
        return x*fac(x-1)
plusone(5) adds 1 a total of 5 times

value of 1+1+1+1+1

value of 1 + ___

Base case:
plusone(0) should return ___
plusone(5) is $1+1+1+1+1$

value of 1 plus value of $1+1+1+1+1$

plusone(4)

Base case:
plusone(0) should return ___
def plusone(n):
    
    returns n by adding 1's!

    if n == 0:
        return

    else:
        return
def plusone(n):
    """
    returns n by adding 1's!
    """
    if n == 0:
        return 0
    else:
        return 1 + plusone(n-1)
value of \( 2 \times 2 \times 2 \times 2 \times 2 \) is

\[
\text{pow}(2, 5)
\]

Base case:
\[
\text{pow}(2, 0) \text{ should return } __ ?
\]
value of \(2 \times 2 \times 2 \times 2 \times 2\) is \( \text{pow}(2, 5)\) or \(b\) to the \(p\)th power.

Base case:
\text{pow}(2, 0)\) should return \(\_\_\)?
def pow(b, p):
    """
    b**p, defined recursively!
    """

    if p == 0:
        return
    else:
        return

Extra! Can we also handle negative powers...?
def pow(b, p):
    """b**p, defined recursively!""
    if p == 0:
        return 1.0
    elif p < 0:
        return b * pow(b, p - 1)

Extra! Can we also handle negative powers...?
```python
def pow(b, p):
    """b**p, defined recursively!""

    if p == 0:
        return 1.0
    elif p < 0:
        return 1.0/pow(b, -p)
    else:
        return b*pow(b, p-1)

Extra! Can we also handle negative powers... ?
```
Recursion's advantage: It handles arbitrary structural depth – *all at once + on its own!*

As a hat, I'm recursive, too!
Recursion's advantage:

It handles arbitrary structural depth – *all at once + on its own!*
Design patterns...

Recursion's a design - not a formula, **BUT**, these pieces are common:

\[
\text{s} = 'alien'
\]

in terms of \( s \), what are these pieces? (index! slice!)
Design patterns...

Recursion's a design - not a formula, **BUT**, these pieces are common:

```python
s = 'aliiien'

's[0]'  # handle the *first* Human!

's[1:]'  # recurse the *rest* Machine!
```
Design patterns...

Recursion's a design - not a formula, **BUT**, these pieces are common:

\[ L = [3, 1, 4, 1, 5, 9] \]

- **handle the first** \( L[0] \)
- **recurse the rest** \( L[1:] \)

Human! Machine!
Design patterns...

- **Do one piece of work:** \(L[0] \text{ or } s[0]\)

- **Recurse with the rest:** \(L[1:] \text{ or } s[1:]\)

- **Combine! Make sure all types match...**

- **Handle base cases, with if ...**

Recursion's a design - not a formula, **BUT**, these pieces are common:
Base case:
numis("") should return ___ ?

def numis(s):
    """ # of i's in s ""
    if s == '':
        return
    elif s[0] == 'i':
        return
    else:
        return
```python
def numis(s):
    """ # of i's in s """
    if s == '':
        return 0
    elif s[0] == 'i':
        return 1 + numis(s[1:])
    else:
        return numis(s[1:])
```

What's really being added here?
len('yaycs')

# of chars in 'yaycs'

is

len(s)

length of s

# of chars in 'y'

# of chars in 'aycs'

Base case:
len('') should return ____ ?
def len(s):
    """
    returns the length of s
    """
    if s == '':
        return __________
    else:
        return ________________

Extra! Can we also handle LISTS... ?
def len(s):
    """
    returns the length of s
    """
    if s == '' or s == []:
        return 0
    else:
        return 1 + len(s[1:])

one, plus... ... the length of the rest of s
A brief word from our sponsor, Mother Nature...

Like broccoli, recursion is "Good for You™"
Yes... and no. Are these rules for real?
But, do only *plants* get to be recursive?
There still has to be a base case...
or else!
Leap before you look!

Try these four...
vwl('eerie')

# of vowels in 'eerie'

is

# of vowels in 'e' + # of vowels in 'erie'

Base case:
vwl('') should return ___ ?
def vwl(s):
    
    
    if s == ' ':
        return
    
    elif s[0] in 'aeiou':
        return
    
    else:
        return
Python is...  

>>> 'i' in 'team'
False

>>> 'cs' in 'physics'
True

>>> 'i' in 'alien'
True

>>> 42 in [41, 42, 43]
True

>>> 3*'i' in 'alien'
False

>>> 42 in [[42], '42']
False
keepvwl('pluto')

keep vowels in 'pluto'

keepvwl('pluto') is

keep vowels in 'p' + keep vowels in 'luto'

Base case:
keepvwl('') should return ___?
def keepvwl(s):
    """ returns ONLY the vowels in s! """
    if s == '':
        return
    elif s[0] in 'aeiou':
        return
    else:
        return
max([7,5,9,2])

max of [7,5,9,2] is either 7 or the max of [5,9,2]

Base case:
if len(L) == 1, what should max(L) return?
def max(L):
    
    returns the max of L!
    
    if len(L) == 1:
        return

    M = max(L[1:])

    if L[0] > M:
        return
    else:
        return

    The max of the REST of L
zeroest([-7, 5, 9, 2])

zeroest of
[-7, 5, 9, 2]

is

either -7

or the zeroest of [5, 9, 2]

Base case:
if len(L) == 1, what should zeroest(L) return?
def zeroest(L):
    """ returns L's element nearest 0 """

    if len(L) == 1:
        return ______________

    Z = ______________
    The zeroest of the REST of L

    if ______________:
        return ______________

    else:
        return ______________
def vwl(s):
    """ # of vowels in s """
    if s == '':
        return 0
    elif s[0] in 'aeiou':
        return 1 + vwl(s[1:])
    else:
        return vwl(s[1:])

What seven-letter s maximizes vwl(s)?
def keepvwl(s):
    """ returns ONLY the vowels in s! """

    if s == '':
        return ''

    elif s[0] in 'aeiou':
        return s[0]+keepvwl(s[1:])

    else:
        return keepvwl(s[1:])
def max(L):
    """ returns the max of L! """
    if len(L) == 1:
        return L[0]
    M = max(L[1:])
    if L[0] > M:
        return L[0]
    else:
        return M

The max of the REST of L
def zeroest(L):
    """ returns L's element nearest 0 """

    if len(L) == 1:
        return L[0]

    Z = zeroest(L[1:]): 

    if abs(L[0]) < abs(Z):
        return L[0]
    else:
        return Z

The zeroest of the REST of L
The key to understanding recursion is, first, to understand recursion.

- former CS 5 student

It's the eeriest!

Good luck with Homework #1

tutors @ LAC + 4C's   Th/F/Sa/Su/Mon.
def pow(b, p):
    if p == 0:
        return 1
    elif p < 0:
        return 1 / pow(b, -p)
    else:
        return b * pow(b, p - 1)

Extra! See if you can also handle negative powers...

pow(b, p) returns b**p, but using only multiplication times b

pow(2, 4) ~ 2*2*2*2
pow(2, 3) ~ 2*2*2
pow(2, 2) ~ 2*2
pow(2, 1) ~ 2
pow(2, 0) ~ 1

anything to the zero power is 1.0
def vwl(s):
    if s == '':
        return 0
    elif
        
else:

Extra! What 7-letter English word *maximizes* vwl(s)?
**Answer:** `pow`

def `pow(b, p)`:
    if p == 0:
        return 1.0
    else:
        return `pow(b, p-1) * b`

Extra! See if you can also handle negative powers...